

## 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 focuses on sequential HW processes. Combinational HW processes are supported but mostly not discussed 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](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, input1 is sampled when the process wakes on the first @(posedge clk) and evaluates the RHS. For the purposes of the scheduling rules, we anchor that Read(input1,.) to the most recent synchronization point. The resulting Write(output1,.) is anchored to the next synchronization point (the cycle in which the written value is treated as stable for others to observe).

Cycle-level anchoring convention (used throughout this document). For cycle-level reasoning, each signal IO is assigned an anchor cycle relative to surrounding synchronization calls

(e.g., wait() / SyncChannel / @(posedge clk)):

- Read(sig, val) is anchored to the closest preceding synchronization call, i.e. pred\_S(Read).
- Write(sig, val) is anchored to the closest succeeding synchronization call, i.e. succ\_S(Write).

This convention is only for cycle-level reasoning and does not depend on simulator update regions or delta-cycle ordering.

## 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](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. In this document, message-passing channels are assumed to be point-to-point: for each channel *c*, exactly one producer performs all Push\_*c* operations and exactly one consumer performs all Pop\_*c* operations.

*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\\_\(computer\\_science\)](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*: In this document, a blocking message-passing operation is modeled as issuing a persistent request that remains asserted (and with stable payload, if applicable) until the operation commits (the message is sent or received). A non-blocking message-passing operation returns immediately with a status indicating whether the operation committed in that cycle.

*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. Wait statements explicitly placed in the model are called *explicit wait statements* in this document and 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 issued before or in the same cycle in which the synchronization interface commits. (Here “the synchronization interface commits” refers to the commit of that synchronization call in the calling process’s trace (i.e., the cycle at which the process completes the synchronization and may proceed). For blocking message-passing operations (Push/Pop), the process cannot complete the synchronization call until any earlier issued blocking message requests in that interval have committed; therefore, any such blocking operations that occur before the synchronization call in program order necessarily commit no later than the synchronization call’s commit cycle.)
  - All message-passing operations after a call to a synchronization interface shall be issued after the cycle in which the synchronization interface call commits.
  - 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 prevents HLS from introducing deadlocks *via this specific mechanism* (reversing the program order of message-passing calls that appear in sequence in the source). However, deadlock behavior can still be affected by other transformations that change the *timing* of request/commit—most notably overlapped execution in pipelined loops and finite-capacity effects. In this document, preservation of deadlock behavior in the presence of pipelined-loop overlap relies on the additional pipelined-loop discipline described below (automatic flush / WFG-inertness).

## 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 but is not required to (and in most cases will not do so). We call these

wait statements *explicit pipeline stage wait statements*. 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.

Modeling convention for pipelined-loop overlap.

The post-HLS pipelined-loop implementation may overlap iterations and may internally accept/stage up to the finite bound  $B(c\_in)$  of input values for younger overlapped iterations before older iterations' outputs are produced. This internal accept/stage behavior is part of the back-annotated channel realization counted in  $B(c\_in)$  between the  $\Sigma$ -visible endpoints of  $c\_in$  (producer  $Push\_c\_in$  and consumer  $Pop\_c\_in$ ). These internal accept/stage movements are modeled as  $\epsilon$ -steps. They do not introduce any additional ("second")  $\Sigma$ -visible  $Pop\_c\_in$  events:  $\Sigma$  records only the committed boundary transfers at the  $\Sigma$ -visible endpoints. In particular, the consumer-side boundary  $Pop\_c\_in$  is the loop-body  $Pop\_c\_in$  operation from the pre-HLS source when that  $Pop$  is included in the chosen  $\Sigma / \Sigma\_DUT$  projection. The externally relevant effect of pipelining is therefore captured entirely by  $B(c\_in)$  (via boundary backpressure/acceptance behavior), without introducing extra  $\Sigma$ -visible boundary  $Pops$ . Once the pipeline reaches a quiescent blocked state under the automatic flush discipline below, it does not initiate new blocking request assertions for younger overlapped iterations.

Multi-input  $Pop$  points and coupler policies.

When a pipelined loop consumes from multiple input channels (e.g.,  $c1, c2$ ), back-annotation still assigns a separate finite bound  $B(c_i)$  and cycle-boundary occupancy  $occ\_c(t)$  to each channel  $c_i$ . Terminology: when the realization is elaborated (Appendix J, Remark 2), the "channels" to which  $B(\cdot)$ ,  $occ\_c(t)$ , and E4 apply are the explicit abstract channels of that elaborated model (including any introduced staged/bundle channels); throughout this section we write  $c_i$  for the chosen  $\Sigma$ -visible abstract boundary channel associated with the source-level name  $c_i$  (not for an entire downstream internal path through a join). As in the single-input case,  $occ\_c(t)$  is determined solely by the committed  $\Sigma$ -visible boundary transfers ( $Push/Pop$  commits) on that same channel; internal pipeline staging and any internal handoff within the composite realization counted in  $B(c)$  are  $\epsilon$ -steps and do not create additional  $\Sigma$ -visible events. In multi-input pipelines, the back-annotated realization may elaborate the concrete channel implementation by introducing explicit finite-capacity staged/bundle channels to represent any per-input staging or shared pipeline capacity, and may use purely combinational (stateless) coupler/join logic to enforce the intended mutual-stall behavior only at explicit internal staged/bundle/join boundaries. The coupler itself contributes no capacity state and does not redefine per-channel E4 capacity/availability at the chosen  $\Sigma$ -visible boundaries: per-channel  $B(c_i)$  and  $occ\_c_i(t)$  remain defined solely by  $\Sigma$ -visible commits on  $c_i$ . Accordingly, any cross-channel dependency (e.g., "join not met") shall be expressed only via the explicit staged/bundle channels introduced by the elaboration, i.e., as  $ChannelDisabled$  at those internal boundaries under their own E4 availability predicates, and shall not appear as cross-channel disablement at the chosen  $\Sigma$ -visible channel endpoints.

Practical note ( $\Sigma\_DUT$  projection). The loop-body  $Pop\_c\_in$  in the pre-HLS source (i.e., the consumer-side  $Pop\_c\_in$  operation, conceptually after the in-flight capacity  $B(c\_in)$ ) may be exposed in  $\Sigma\_DUT$  if desired, but is typically omitted since it is latency-insensitive, and it is often practically difficult to expose/collect in the post-HLS RTL. The liveness/deadlock reasoning still accounts for this blocking point without requiring it to be  $\Sigma\_DUT$ -visible: Appendix K defines WFG edges using the  $ChannelEnabled/ChannelDisabled$  status of the corresponding boundary  $Pop\_c\_in$  (via  $E4/occ\_c/B(c)$ ), so no additional internal interface needs to be treated as a separate  $\Sigma$ -observable channel.

Potential deadlocks are avoided by having the pipeline automatically flush.

Automatic flush (drain-only blocking discipline).

Terminology (most-upstream pending blocking op / frontier). In an  $\epsilon$ -quiescent cycle, consider the set of pending (i.e., issued but not yet committed) source-level blocking message-passing operations (Pop/Push) in the current interval between adjacent synchronization calls. By the Issue/Commit linkage / BoundaryReq definition, each such pending blocking op has its endpoint-owned boundary request asserted (Push: vld, Pop: rdy) until it commits; couplers may affect only the complementary mirror signal (Push: rdy, Pop: vld) and thus whether the op commits in that cycle after  $\epsilon$ -settling. The most-upstream pending blocking op(s) are those that are minimal with respect to source program order within that interval (if multiple minimal blocking ops are pending, they form a frontier). (This is the same “minimal pending blocking frontier” notion formalized later as  $\text{Front\_P}(\sigma)$  in Appendix K.)

A pipelined loop is said to automatically flush if, whenever the minimal pending blocking frontier is non-empty and every most-upstream pending blocking op (i.e., every member of that minimal frontier) cannot make progress because it is not channel-enabled (evaluated after combinational ready/valid/backpressure has  $\epsilon$ -settled for the cycle), then:

1. Block point. The process is blocked on message passing at that most-upstream pending blocking frontier, as in the unpipelined source.
2. No new later work. While blocked at that most-upstream frontier (i.e., while the minimal pending blocking frontier remains non-empty and all of its members are channel-disabled), the implementation does not issue any new blocking Pop/Push operations that are later in source program order (including any younger overlapped iterations) and therefore does not begin asserting their BoundaryReq.
3. No withdrawal. While blocked, the implementation does not withdraw any already-asserted blocking Pop/Push request (no “try-and-withdraw”); outstanding blocking requests remain asserted until they commit.
4. Drain-only downstream progress. Work that has already passed the blocked frontier point(s) may continue to drain, and any already-asserted resulting output-Push requests may commit when channel-enabled, provided no work advances past the blocked frontier point(s).

(Equivalently: this is the “ALL disabled  $\Rightarrow$  stall” policy—P stalls only when none of the operations in the minimal pending blocking frontier are channel-enabled; downstream work may drain while the frontier is fully disabled.)

Example ( $II=1$ , latency=4, automatic flush): a loop that performs one blocking  $\text{Pop}_{\{c\_in\}}$  and one blocking  $\text{Push}_{\{c\_out\}}$  per iteration in the pre-HLS source may, after pipelining in the post-HLS RTL, internally accept/stage up to four input values into pipeline staging (pipeline overlap) before the first output  $\text{Push}_{\{c\_out\}}$  commits, subject to the back-annotated bound  $B(c\_in)$  (E4). Values may then reside for several cycles in internal pipeline storage; such internal staging/hand-off is  $\epsilon$ -state within the back-annotated realization and does not create additional  $\Sigma$ -visible  $\text{Push}_c/\text{Pop}_c$  events beyond the boundary commits at the  $\Sigma$ -visible endpoints. The drain-only blocking behavior is as defined above under “automatic flush”; in particular, if the blocked operation is in a late pipeline stage (e.g., the final Push), there may be little or no downstream work to drain.

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 reordered relative to same-interface accesses from other iterations.

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_val` 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 terms of the Appendix G trace model, the  $\Sigma$ -visible Read(sig,val) events remain anchored to their synchronization points; any additional internal sampling made by the implementation between anchors is treated as an  $\epsilon$ -step and must not change the recorded value label.

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 on distinct interfaces (still not moving calls across synchronization interface calls). This relaxed mode may violate the default no-reverse discipline and therefore may introduce deadlocks that are not present under the default rules. 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 new deadlocks *via reversal of source order on distinct message interfaces* as described earlier. Deadlock behavior may still depend on bounded-capacity effects and on overlapped execution (e.g., loop pipelining); those cases are addressed separately in this document via the pipelined-loop discussion (automatic flush / WFG-inertness) and the later deadlock reasoning. The formal equivalence and deadlock-preservation results in Appendices G–K assume STRICT\_IO\_SCHEDULING is not set to relaxed.

## 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. This rule is stated in terms of commit cycles (the cycle the mapped memory storage is accessed); the scheduler enforces it by choosing issue cycles consistent with the fixed access latency defined by the array access mapping layer.
- 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.

## Definitions: Issue vs. Commit

The definitions immediately below are with respect to the  $\Sigma$ -visible endpoints of the chosen modeling boundary (Sys or Sys\_B)—i.e., the ready/valid interfaces at which  $\Sigma$  records message-transfer events. Ready/valid handshakes on internal micro-interfaces inside an HLS pipeline or inside a back-annotated channel realization that merely move values within the composite realization counted in B(c) (e.g., post-HLS RTL pipeline stage implementation) are modeled as  $\epsilon$  and do not constitute Issue/Commit events unless they are explicitly included in the chosen observation/projection (e.g.,  $\Sigma_{DUT}$ ).

Definition: Issue (message channel) — as an auxiliary (scheduler/witness) timestamp

Issue(q) is an auxiliary timestamp used to state scheduling constraints (R4/E3/E5). Intuitively, it is the first cycle in which the calling process begins requesting the transfer corresponding to the source-level message operation instance q at the chosen  $\Sigma$ -visible boundary endpoint.

Issue/Commit linkage (well-formedness conditions). Fix a  $\Sigma$ -visible endpoint direction, and let req(t) denote the endpoint-owned boundary request signal in cycle t (valid for Push, ready for Pop), i.e., the process-driven request level corresponding to BoundaryReq. Any  $\epsilon$ -settling of combinational channel-side structure (including couplers) may affect only the complementary mirror signal (Push: rdy; Pop: vld) and therefore whether a transfer commits; it does not alter req(t). For each source-level message operation instance q on that endpoint direction:

- Boundary-request soundness. req(Issue(q)) = 1. If q is a blocking Push/Pop, then once issued it remains the unique outstanding request on that endpoint direction until it commits: req(t)=1 in every cycle t from Issue(q) through the commit cycle of q (inclusive), with stable payload while asserted for Push.
- Stable attribution while outstanding. If req(t)=1 and no transfer commits on that endpoint direction in cycle t, then any continued assertion req(t+1)=1 is attributed to the same outstanding q (i.e., attribution may not change from q to q' without an intervening commit on that endpoint direction).
- Back-to-back issuance under continuously asserted req. If q<sub>0</sub> commits in cycle t and the next source-level operation q<sub>1</sub> on the same endpoint direction is issued back-to-back, then Issue(q<sub>1</sub>)=t+1 (even if the

request signal does not deassert between commits), provided  $q_0$  and  $q_1$  lie within the same source interval between adjacent synchronization calls.

- Sync-boundary discipline. If  $q_0 \in \text{pref\_S}(s)$  and  $q_1 \in \text{suff\_S}(s)$  for a synchronization call  $s$ , then  $\text{Issue}(q_1)$  is strictly after  $s$ 's commit in the same trace (i.e.,  $\text{Issue}(q_1) > \text{clk\_pre}(s)$  in  $\tau_{\text{pre}}$  and  $\text{Issue}(q_1) > \text{clk\_post}(s)$  in  $\tau_{\text{post}}$ ). Any boundary request assertion that persists while the process is still pending at  $s$  is attributed to the outstanding operation(s) in  $\text{pref\_S}(s)$  and does not constitute issuance of any operation in  $\text{suff\_S}(s)$ .
- Non-blocking operations. A non-blocking poll may still assert a  $\Sigma$ -visible endpoint request; issuance is attributed using the Issue/Commit linkage conditions above (stable attribution while outstanding), so that a continuously asserted endpoint request with no intervening commit is attributed to one issued request  $q \in Q$  (with  $\text{Issue}(q)$  at the first asserted cycle, or at  $t+1$  immediately following the previous commit if the request never deasserts), and only a poll that succeeds yields the corresponding committed transfer  $m(q) \in M$ . Repeated poll checks under a continuously asserted request produce no  $\Sigma$ -event and are  $\epsilon$ -steps within that same  $q$  (no additional  $q \in Q$ ).

Definition: Commit (message channel) — as observed by an external clocked observer

Commit is the cycle in which an external clocked observer sees a transfer *complete* at a  $\Sigma$ -visible boundary endpoint:

- A message commits in the cycle when the observer sees both valid and ready high at the same time.
- The committed payload is the data value seen in that same cycle.

In the presence of a multi-input coupler, the settled ready/valid condition for a Pop at an internal staged/bundle/join boundary in the elaborated realization may be a combinational function of other channels' availability/backpressure and internal pipeline state ( $\epsilon$ -settled). At the chosen  $\Sigma$ -visible boundary endpoints, however, ChannelEnabled/ChannelDisabled for channel  $c_i$  shall remain governed solely by E4 for  $c_i$  together with the process's BoundaryReq on that endpoint; any cross-channel dependency shall be mediated only via the explicit internal staged/bundle channels. A commit—when it occurs—is still recorded as the single per-channel  $\Sigma$ -visible transfer event on  $c_i$ .

Modeling note (buffered FIFO vs. rendezvous at the  $\Sigma$ -visible boundary). The handshake definition above applies uniformly for all channels, but the channel-side behavior of ready/valid at the  $\Sigma$ -visible boundary is constrained by the annotated capacity  $B(c)$  and the cycle-boundary occupancy interpretation in E4 / Appendix I.2.

- If  $B(c) > 0$  (buffered FIFO): the  $\Sigma$ -boundary interface is cycle-boundary, non-fall-through (at least 1-cycle registered at the boundary). Concretely, a Pop commit in cycle  $t$  requires that the FIFO is non-empty at the start of cycle  $t$  ( $\text{occ\_c}(t) > 0$ ), and a Push commit in cycle  $t$  requires that the FIFO has space at the start of cycle  $t$  ( $\text{occ\_c}(t) < B(c)$ ). Therefore, a Push and a Pop may both commit in the same cycle only when the FIFO begins that cycle with both data and space available ( $0 < \text{occ\_c}(t) < B(c)$ ); in that case the occupancy is unchanged by that cycle's commits.
- If  $B(c) = 0$  (rendezvous): there is no storage; a transfer commits only when the complementary endpoint participates in the same cycle, so Push<sub>c</sub> and Pop<sub>c</sub> commit together and the cycle-aligned occupancy is always 0.

For RAM reads/writes as presented above, “commit” denotes the precise cycle the memory storage is accessed; since the access latency is fixed and known to the HLS scheduler, it can schedule the operation's issue cycle so that the commit cycle satisfies the required ordering relative to Sync. The “Scheduling of Array Accesses” constraints above are therefore written in commit-time. For message passing, HLS directly constrains only the local issue time (commit depends on availability/backpressure);

however, when this document states constraints relative to “the cycle the Sync commits,” that Sync-commit cycle is the commit of that Sync call in the calling process’s trace, and may be delayed until earlier issued blocking message requests in the same interval have committed. For non-blocking message-passing operations (PushNB/PopNB), a call that returns “completed” commits in its issue cycle, and a call that returns “not completed” does not commit (i.e., it produces no committed-transfer event).

## 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, depending on buffering/pipelining in the realization, the time between a producer’s committed Push and the consumer’s committed Pop (and the transient in-flight occupancy) may vary. Two unidirectional message-passing channels in opposite directions 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) {
```

```

        ol.write(new_val);
    do {
        wait();
    } while (!go.read());
    new_val = il.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> ol;
Connections::In<int> pop1;
void my_thread() {
    int new_val=0;
    while (1) {
        ol.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
    }
}

```

Under the anchoring rules, `i1.read()` remains associated with the same synchronization point regardless of the intervening Pop. However, in raw pre-HLS simulation, a blocking Pop may stall for one or more cycles, so `i1` can change before the `i1.read()` executes—creating behavior that differs from the anchored interpretation (and potentially from post-HLS scheduling). The fix is simply to place `i1.read()` immediately after its intended `wait()` anchor; this removes the mismatch risk without affecting 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> ol;
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. For direct-input signals/ports (e.g., `#pragma hls_direct_input` and `#pragma hls_direct_input_sync`), this requirement refers to the logical ( $\Sigma$ -visible) anchor cycle of the Read/Write: the event is still considered to occur at its synchronization point even if HLS physically samples a stable signal later; such late-sampling micro-steps are  $\epsilon$ -steps and are not  $\Sigma$ -visible. 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 adhere 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, HLS synthesis tools that do not follow the rules described in this document might 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 (anchoring) of signal IO operations, and synchronization calls can affect the scheduling of message-passing calls. In the conceptual anchoring semantics of this document, message-passing calls do not affect the anchor cycle of signal IO operations (and signal IO does not affect the legality/order of committed message transfers) except through explicit synchronization. Practically, in raw pre-HLS SystemC simulation a blocking message operation may introduce additional cycles of waiting, so signal IO statements that are lexically separated from their intended anchor can observe different values; RULE 1 addresses this by recommending placement of signal IO adjacent to its anchor.
- HLS cannot by default reverse the order of message-passing calls, so it cannot introduce new deadlocks *via call-order reversal*. For pipelined loops (overlapped iterations) and other transformations that change the timing of request/commit under bounded capacities, deadlock preservation depends on the additional pipelined-loop discipline described in this document (automatic flush / WFG-inertness), not on the no-reverse rule alone.
- 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 always occur exactly at synchronization points (e.g., wait statements) that are explicit in the pre-HLS source. (For direct-input signals/ports, “occur at synchronization points” is meant in the logical/anchoring sense: the  $\Sigma$ -visible Read/Write is timestamped at the synchronization point even if the implementation samples the stable signal later; those internal late samples are  $\epsilon$ -steps.)

## 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);	<b>back1 = backs.Pop();</b>
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

This content has been moved into the “Pipelined Loops” section in the main document.

## 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](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](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 operations before a call to a synchronization interface shall be issued before or in the same cycle in which the synchronization interface commits.  
(For blocking message-passing operations (Push/Pop), this implies the corresponding commits occur no later than the synchronization call’s commit cycle.)
- 5) All message-passing operations after a call to a synchronization interface shall be issued after the cycle in which the synchronization interface call commits.
- 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. Operations on the same message-passing interface are issued in source order (they are never reversed).
- 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) We distinguish the rendezvous source model Sys (B(c)=0 for all channels) from the buffered source interpretation Sys\_B, which uses bounded-FIFO channel capacities B(c) matching the post-HLS RTL (Appendices I–K).

Conceptually, systems are composed of many such processes which follow the basic conceptual model, and which communicate using only synchronization interface calls, latency-insensitive signal-level protocols, and committed message-passing transfers. If a design uses non-blocking message-passing calls (PushNB/PopNB), then unsuccessful polls are treated as internal  $\epsilon$ -steps (no  $\Sigma$ -event), and each successful completion is represented in  $\Sigma$  as the corresponding committed Push\_c/Pop\_c transfer ( $\Sigma$  records transfers, not failed polls). If the resulting latency-sensitive behavior is externally visible at the DUT boundary, the snooping wrapper of Appendix L may be used to align the selected admissible pre-HLS execution with the RTL. 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: In the post-HLS RTL, the pipeline may internally stage/dequeue (“accept”) input values for younger overlapped iterations before older iterations’ outputs are produced, up to the back-annotated bound  $B(c\_in)$  (E4). In the formal model, these internal staging/dequeue transfers into pipeline staging are treated as internal handoff within the back-annotated realization ( $\epsilon$ );  $\Sigma$  records only the committed boundary transfers at the  $\Sigma$ -visible endpoints. Thus, the externally relevant effect of pipelining is captured by  $B(c\_in)$  (and resulting backpressure/acceptance behavior), without requiring  $\Sigma$ -visible ‘early committed’ boundary Pop\_{c\\_in} events. Assuming the pipeline automatically flushes, if a required blocking Pop/Push for the current most-upstream pending work (i.e., for some  $op \in \text{Front\_P}(\sigma)$ , the minimal pending blocking frontier in the current Sync interval) cannot complete—either a read (blocking Pop) is not available or a write (blocking Push) is back-pressured (e.g., FIFO full or rendezvous partner not ready)—the process blocks at that Pop/Push as in the unpipelined source (while downstream stages may drain already-started older work).
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: They are modeled at the level of committed transfers (successful completion), with unsuccessful polls treated as  $\epsilon$ -steps and thus not part of  $\Sigma$ . If latency differences in these committed-transfer events are externally visible at the DUT boundary and must be aligned for verification, 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 (Appendix L).
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. For liveness/deadlock guarantees (Appendices I–K), the results additionally assume weak fairness and channel-progress obligations for the scheduler/environment (Appendix N); the cycle-by-cycle R/E rules alone do not imply those progress properties.

## 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
$\Sigma$	The alphabet of observable IO actions
$\tau_P$	<p>A (finite or infinite) per-cycle trace of observable actions executed by process <math>P</math>: for each global clock cycle <math>t</math>, <math>\tau_P[t]</math> is the (possibly empty) set of <math>\Sigma</math>-actions committed by <math>P</math> at <math>t</math>. Actions committed in the same cycle are treated as simultaneous (unordered); only cross-cycle order (and the explicit E1–E5 constraints) is semantically significant.</p> <p>Single-op-per-cycle (channel endpoint) constraint: for each message-passing channel <math>c</math> and each clock cycle <math>t</math>, there is at most one committed <math>\text{Push}_c</math> event in the entire system at <math>t</math>, and at most one committed <math>\text{Pop}_c</math> event in the entire system at <math>t</math> (i.e., the producer endpoint is single-ported for <math>\text{Push}_c</math> and the consumer endpoint is single-ported for <math>\text{Pop}_c</math>). Thus, on a fixed channel in a given cycle, the set of commits contains <math>\leq 1</math> <math>\text{Push}_c</math> and <math>\leq 1</math> <math>\text{Pop}_c</math> (it may</p>

	contain one of each in the same cycle). For rendezvous channels ( $B(c)=0$ ), a committed transfer occurs only as a matching same-cycle Push_c(v) and Pop_c(v) pair. For buffered channels ( $B(c)>0$ ), a same-cycle Push_c and Pop_c do not imply fall-through; the channel semantics are cycle-boundary, non-fall-through as stated in E4.
S	The subset of $\Sigma$ that are synchronization calls (explicit wait, SyncChannel, START_P/FINISH_P indicating process start/finish)
R	The subset of $\Sigma$ that are signal reads
W	The subset of $\Sigma$ that are signal writes
M	The subset of $\Sigma$ that are committed message-passing transfers (i.e., committed Push_c / Pop_c events on message-passing channels, including successful non-blocking transfers). Unsuccessful non-blocking polls contribute no $\Sigma$ -event and are modeled as $\epsilon$ -steps, and therefore are not members of M.
Q	The set of dynamic source-level message-operation instances executed by a process at the $\Sigma$ -visible endpoints (blocking Push/Pop, and non-blocking PushNB/PopNB calls). Elements of Q may or may not commit.

A system trace is the tuple

$\tau = (\tau_P)_P$ , one component per process.

For any action  $a \in \Sigma$ , let  $\text{clk\_pre}(a)$  /  $\text{clk\_post}(a)$  be the clock cycle at which  $a$  is committed.

For any issued message request  $q \in Q$ ,  $\text{issue\_pre}(q)$  /  $\text{issue\_post}(q)$  denotes the auxiliary Issue timestamp for  $q$  in the pre-HLS / post-HLS trace, respectively, as defined in “Definitions: Issue vs. Commit” and required to satisfy the associated Issue/Commit linkage (well-formedness) conditions with respect to the  $\Sigma$ -visible boundary request signal on  $q$ ’s endpoint direction.

If  $q$  commits, it contributes a unique committed transfer  $m(q) \in M$  with commit cycle  $\text{clk\_pre}(m(q))$  /  $\text{clk\_post}(m(q))$ . If  $q$  does not commit (e.g., an unsuccessful non-blocking call), then  $m(q)$  is undefined / absent and no  $\Sigma$ -event is added to M.

(No deassert-before-commit assumption.) For blocking message requests  $q \in Q$ , once  $q$  is issued, its  $\Sigma$ -visible boundary request is not withdrawable: it remains asserted in every cycle from  $\text{issue\_pre}(q)$  (resp.  $\text{issue\_post}(q)$ ) through  $q$ ’s commit cycle (if it commits), and for Push the payload remains stable while the request is outstanding.

### 1. Conceptual Model for a Single Process

For every process  $P$  the pre-HLS trace  $\tau_P$  must satisfy the following *partial-order constraints*:

*R1 (Program order of S).*

$\forall s_1, s_2 \in S : s_1 <_{\text{src}} s_2 \Rightarrow \text{clk\_pre}(s_1) < \text{clk\_pre}(s_2)$

*R2 (Location of R/W).*

$\forall r \in R : \text{clk\_pre}(r) = \text{clk\_pre}(\text{pred}_S(r))$

$\forall w \in W : \text{clk\_pre}(w) = \text{clk\_pre}(\text{succ}_S(w))$

where  $\text{pred}_S(r)$  (resp.  $\text{succ}_S(w)$ ) is the closest synchronization call that precedes (resp. follows) statement  $r$  in the source order. If no lexical predecessor (successor) exists, a virtual synchronization operation at time 0 (resp.  $\infty$ ) is assumed.

*R3 (Isolation around S).*

Let  $\text{pref}_S(s)$  (resp.  $\text{suff}_S(s)$ ) be the set of issued message requests  $q \in Q$  whose call is lexically before (resp. after) synchronization call  $s$ . Then

$\forall q \in \text{pref}_S(s) : \text{issue\_pre}(q) \leq \text{clk\_pre}(s)$  and  $\forall q \in \text{suff}_S(s) : \text{issue\_pre}(q) > \text{clk\_pre}(s)$ .

(For blocking message requests, if the synchronization call commits in the process trace, then any earlier blocking message requests in that source interval must have committed in some cycle  $\leq$  that sync-commit cycle, by blocking-call control-flow semantics.)

*R4 (Safe message issue ordering).*

For every pair  $q1, q2 \in Q$  within the same process,

$q1 <_{\text{src}} q2 \Rightarrow \text{issue\_pre}(q1) \leq \text{issue\_pre}(q2)$ .

(For same-channel operations, this simply states that calls on a single interface are issued in program order; for distinct channels, it forbids issuing them in reverse order.)

*R5 (Channel capacity semantics; Sys vs. Sys<sub>B</sub>).*

Appendix G defines the rendezvous source model Sys, in which every channel has capacity 0.

Concretely: for every channel  $c$  and any clock cycle  $t$ , the number of unmatched committed Push<sub>c</sub> actions is zero and the number of unmatched committed Pop<sub>c</sub> actions is zero; equivalently, no Push<sub>c</sub> shall commit unless a matching Pop<sub>c</sub> also commits at  $t$ , and no Pop<sub>c</sub> shall commit unless a matching Push<sub>c</sub> also commits at  $t$ .

In Appendices I–K we additionally use the buffered source interpretation Sys<sub>B</sub>: the same source processes as Sys, but interpreted under bounded-FIFO channel semantics with capacities  $B(c)$  matching RTL<sub>B</sub> (E4), with rendezvous recovered as the special case  $B(c)=0$ .

The source conceptual semantics for a process is thus a labeled partial order  $(\Sigma, \leq_P)$  generated by R1–R4 plus the channel semantics of Sys (rendezvous) or Sys<sub>B</sub> (bounded FIFO), depending on which model is being referenced.

## 2. Equivalence Relation

Let  $\tau_{\text{pre}}$  and  $\tau_{\text{post}}$  be the  $\Sigma$ -traces (finite or infinite) of the same process before and after HLS, where each  $\Sigma$ -event is labeled exactly as in “Common Formal Definitions for Appendices I–K” (e.g., Push<sub>c</sub>( $v$ ), Pop<sub>c</sub>( $v$ ), Read(sig, val), Write(sig, val), and synchronization events).

Convention (S): all references to  $S$ ,  $\text{pred}_S/\text{succ}_S$ , and  $<_{\text{src}}$  in E1–E5 use the source order of R2. Thus  $\tau_{\text{pre}}$  denotes the  $\Sigma$ -trace of the pre-HLS source.

$\Sigma$ -event matching convention. We use a canonical per-endpoint occurrence matching to make “the same  $\Sigma$ -events” precise. For each channel  $c$ , match committed Push<sub>c</sub> and Pop<sub>c</sub> events by their ordinal occurrence on that channel (e.g., the  $k$ -th committed Push on  $c$  in  $\tau_{\text{pre}}$  matches the  $k$ -th committed Push on  $c$  in  $\tau_{\text{post}}$ ; likewise for Pop<sub>c</sub>). For each such matched pair, the endpoint ( $c$ ) and the payload/value label must agree. For the ordinal index  $k$  to be well-defined under the per-cycle “set/bucket” trace representation used in these appendices, we assume that for any fixed channel endpoint  $c$ , at most one committed Push<sub>c</sub> and at most one committed Pop<sub>c</sub> may occur in any single clock cycle.

For each observable signal sig, match Read(sig,  $\cdot$ ) and Write(sig,  $\cdot$ ) by their ordinal occurrence on that signal within the process trace, again requiring the value labels to agree. In forming  $\tau_{\text{pre}}$  and  $\tau_{\text{post}}$ , we apply the following canonicalization for observable signals: within each anchor interval between consecutive synchronization events in the order  $S$ , we record at most one  $\Sigma$ -visible Read(sig, val) and at most one  $\Sigma$ -visible Write(sig, val) for each (process, sig), with the anchor cycle given by E2. Any additional physical sampling of sig by the implementation, or redundant re-driving of sig within the same anchor interval, is treated as an internal  $\epsilon$ -step and must not change the recorded value label. If a design intentionally relies on intra-interval changes of sig, or on distinguishing multiple reads/writes of sig within the same anchor interval, then sig must be modeled using an explicit synchronized interface (rather than as an ordinary observable signal) so that those distinctions become  $\Sigma$ -visible.

For synchronization events, match by their per-process occurrence index in the source order (R2). Independent  $\Sigma$ -events on different endpoints (including distinct message-passing interfaces) may commute and/or be grouped differently within a clock cycle as permitted by the R-rules and E3. Accordingly,  $\tau_{\text{pre}}$  and  $\tau_{\text{post}}$  are not required to be identical as a single total order, nor to agree on same-cycle “grouping” of distinct-interface Push/Pop operations, beyond the explicit ordering/timestamp constraints in E1–E5 below.

We say  $\tau_{\text{pre}} \approx \tau_{\text{post}}$  iff the two traces match on  $\Sigma$  in this sense, and all of the following hold:

*E1 (Synchronization-order preservation).*

$\forall s1, s2 \in S : \text{clk\_pre}(s1) < \text{clk\_pre}(s2) \Rightarrow \text{clk\_post}(s1) < \text{clk\_post}(s2)$

*E2 (Signal-visibility preservation).*

$\forall r \in R : \text{clk\_post}(r) = \text{clk\_post}(\text{pred}_S(r))$

$\forall w \in W : \text{clk\_post}(w) = \text{clk\_post}(\text{succ}_S(w))$

*E3 (Safe message issue ordering). (Post-HLS preservation of R4).*

For any two message operations  $q1, q2 \in Q$  issued by the same process:

(i) Same-interface order is always preserved: if  $q1$  and  $q2$  access the same message-passing interface/channel and  $q1 <_{\text{src}} q2$ , then  $\text{issue\_post}(q1) \leq \text{issue\_post}(q2)$ .

(ii) Distinct-interface no-reverse: if  $q1$  and  $q2$  access distinct message-passing interfaces and appear in sequence in the source ( $q1 <_{\text{src}} q2$ ), then  $\text{issue\_post}(q1) \leq \text{issue\_post}(q2)$  (i.e., they shall not be issued in the reverse sequence).

(Note: any pipelined-loop internal staging/dequeue discussed in “Pipelined Loops” is  $\epsilon$ -microstate within the back-annotated channel realization and is not a  $\Sigma$ -visible boundary issue event in  $Q$  nor a committed-transfer event in  $M$ ; therefore it does not constitute a counterexample to (ii).)

*E4 (Per-channel bounded-FIFO semantics).*

For every observable channel  $c$  with capacity  $B(c)$ , the projection of  $\tau_{\text{post}}$  to events on  $c$  is FIFO-legal for that bound:

- No drop/duplicate + payload preservation: if the  $\text{Push}_c(v)$  events on  $c$  occur with payload sequence  $v_1, v_2, \dots$  then the  $\text{Pop}_c(v)$  events occur with payload sequence  $v_1, v_2, \dots$  (i.e., pops return the oldest unmatched pushed value).

- No overflow/underflow: for every cycle-aligned prefix  $\pi_t$  of the  $c$ -projection—i.e.,  $\pi_t$  contains exactly the events on  $c$  committed in cycles  $< t$ —letting  $\text{push}(\pi_t)$  and  $\text{pop}(\pi_t)$  be the number of  $\text{Push}_c$  and  $\text{Pop}_c$  events in  $\pi_t$ , we have

$$0 \leq \text{push}(\pi_t) - \text{pop}(\pi_t) \leq B(c).$$

Equivalently, the abstract occupancy after cycle  $t-1$  is  $\text{occ}_c(t) = \text{push}(\pi_t) - \text{pop}(\pi_t)$ .

Because  $\pi_t$  contains exactly the events committed in cycles  $< t$ , the availability predicates for commits in cycle  $t$  are evaluated against  $\text{occ}_c(t)$  at the start of cycle  $t$  (cycle-boundary, non-fall-through for  $B(c) > 0$ ): thus  $\text{Pop}_c$  may commit in cycle  $t$  only if  $\text{occ}_c(t) > 0$ ,  $\text{Push}_c$  may commit in cycle  $t$  only if  $\text{occ}_c(t) < B(c)$ , and simultaneous  $\text{Push}_c$  and  $\text{Pop}_c$  commits in the same cycle are permitted iff  $0 < \text{occ}_c(t) < B(c)$ .

Note:  $\text{occ}_c(t)$  is defined over the entire back-annotated channel realization between the  $\Sigma$ -visible endpoints; any internal handoffs within that realization (e.g.,  $\text{FIFO} \rightarrow \text{staging/skid buffering}$ ,  $\text{staging/skid buffering} \rightarrow \text{pipeline-stage registers}$ ,  $\text{pipeline-stage} \rightarrow \text{pipeline-stage handoff}$ ) are  $\epsilon$ -steps and do not create additional  $\Sigma$ -visible  $\text{Push}_c/\text{Pop}_c$  events beyond the boundary commits.

In particular, in a pipelined-loop implementation, any internal handoff that moves an already-accepted value within the composite realization contributing to  $B(c)$  (e.g., post-HLS RTL pipeline stage implementation) is treated as internal handoff within the realization ( $\epsilon$ ), not as an additional  $\Sigma$ -visible committed  $\text{Pop}_c$  boundary event at the chosen  $\text{Sys}_B$  endpoint.

(Rendezvous is the special case  $B(c)=0$ , which implies  $\text{push}(\pi_t)=\text{pop}(\pi_t)$  for all  $t$ , allowing  $\text{Push}_c$  and  $\text{Pop}_c$  to commit in the same cycle without violating the bound.)

*E5 (Messages cannot cross their immediate synchronization boundary).*

For every synchronization call  $s$  (in the source order used by  $R3$  /  $\text{pref}_S$  /  $\text{suff}_S$ ):

$\forall q \in \text{pref}_S(s) : \text{issue\_post}(q) \leq \text{clk\_post}(s)$

$\forall q \in \text{suff}_S(s) : \text{issue\_post}(q) > \text{clk\_post}(s)$

(For blocking message requests, if the synchronization call commits in the process trace, then any earlier blocking message requests in that source interval must have committed in some cycle  $\leq$  that sync-commit cycle, by blocking-call control-flow semantics. Under the Issue/Commit linkage above, any boundary request assertion that persists while the process is still pending at  $s$  is attributed to the outstanding operation(s) in  $\text{pref}_S(s)$ ; no operation  $q \in \text{suff}_S(s)$  may have  $\text{Issue}(q) \leq \text{clk\_post}(s)$ , and any back-to-back issuance on that endpoint direction may occur only in the first cycle strictly after  $\text{clk\_post}(s)$ .)

See *Appendix I – Per Process Trace Equivalence Proof* for formal proof of the following:

For any single source-level process  $P$  that obeys the conceptual  $R$  rules and  $B$  rules, when interpreted as part of  $\text{Sys}_B$  (i.e., bounded-FIFO channel semantics with finite back-annotated effective capacities  $B(c) \geq 0$  matching the RTL system at the chosen  $\text{Sys}_B$  boundary—see Appendix J, Remark 2—hereafter called  $\text{RTL}_B$ ), every finite observable trace produced by  $P$  has a matching post-HLS RTL trace that satisfies E1–E5. The rendezvous  $\text{Sys}$  case is recovered by setting  $B(c)=0$ .

### 3. Allowed Transformations and Why They Preserve $\approx$

Transformation permitted by the rules	Formal justification
Loop pipelining (overlaps iterations)	Introduces internal pipeline stages (modeled as $\epsilon$ -steps) and overlaps iterations. R1–R5 (and hence E1–E5) are applied to the explicit $S$ set. Loop pipelining may commute independent $\Sigma$ -actions across iterations (e.g., across distinct channels), but it must still preserve: (i) per-channel FIFO legality (E4), (ii) the required issue discipline (E3) for source-ordered pairs, and (iii) the sync-boundary constraints (E1/E5) with respect to the (explicit + implicit) $S$ -actions.
Overlap of internal dequeue/staging vs. later pipeline work ( $\epsilon$ )	Permitted because the internal staging/acceptance activity is $\epsilon$ -internal within the back-annotated channel realization (as defined in “Pipelined Loops”) and does not constitute a $\Sigma$ -visible boundary message-issue reorder. $\Sigma$ -visible boundary message issues must still satisfy E3, and per-channel committed-transfer legality is ensured by E4; automatic flush / Appendix K addresses deadlock preservation for overlap timing.
#pragma hls_direct_input (late sampling of static signals)	This pragma imposes a designer/environment stability contract: the signal’s value must remain stable after reset (or after its last permitted update) while the receiving process may consume it. The logical signal Read actions $r$ remain

Transformation permitted by the rules	Formal justification
	anchored exactly as in E2 ( $\text{clk\_post}(r) = \text{clk\_post}(\text{pred\_S}(r))$ ). HLS may implement this by sampling the signal later (instead of inserting internal storage), but because the signal is stable the sampled value equals the value at $\text{pred\_S}(r)$ ; therefore E2 and $\approx$ are preserved.
#pragma hls_direct_input_sync (controlled updates)	<p>This pragma imposes an explicit timing contract on the environment: the controlled direct-input signals may change only on the cycle of the designated synchronization call <math>s^*</math> (i.e., during the sync handshake). With that contract, the logical anchoring required by E2 is preserved: for the receiving process, the relevant Reads are anchored at the closest preceding synchronization call <math>\text{pred\_S}(r) = s^*</math>, and any environment update Write <math>w\_update</math> is aligned with synchronization (<math>\text{succ\_S}(w\_update) = s^*</math>). HLS may still implement late sampling internally, but the value observed is the same as the value at the E2 anchor point, so <math>\approx</math> is preserved without modifying E2.</p> <p>Instrumentation note (logical anchoring). In the <math>\Sigma</math>-traces used for <math>\approx</math> checking, each signal Read/Write event is timestamped at its E2 anchor point (e.g., <math>\text{clk\_post}(\text{pred\_S}(r))</math> for Reads), not at any later physical RTL sample cycle that HLS may introduce. Any internal late-sampling micro-steps are treated as <math>\epsilon</math>-steps. Therefore, equivalence checking must instrument/record the logical Read/Write events at the anchor (or use a wrapper/transactor that exposes only those anchored events), rather than naively timestamping the physical sample.</p>
Memory-access reordering proven conflict-free	Let $\text{addr}(a)$ be the symbolic address of access $a$ . If the tool proves $\text{addr}(a_1) \neq \text{addr}(a_2)$ 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.
Implicit FSM states / added latency	Extra states introduce <i>silent</i> $\epsilon$ -steps between observable actions; the partial order on $\Sigma$ is unchanged, so E1–E5 are unaffected.
Shared-memory arrays with explicit synchronization	When a memory is accessed by multiple processes, the <i>designer</i> 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 $\approx$ 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 side so that the committed transfer events (successful completions) occur in the same order/timing as observed on the post-HLS channel. This wrapper is itself purely synchronising (S), so it

Transformation permitted by the rules	Formal justification
	cannot violate R1–R5. Once the wrapper is assumed, $\tau_{pre}$ and $\tau_{post}$ coincide on the committed message-transfer history recorded in $\Sigma$ (i.e., the Push_c/Pop_c commit events), so E3–E5 are preserved. Unsuccessful non-blocking polls remain $\epsilon$ -steps and are not required to match. This is not a transformation that inherently preserves $\approx$ , but rather a verification technique to enforce equivalence for inherently latency-sensitive operations.
Latency-sensitive <i>global</i> 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 $\approx$ , but rather a verification technique to enforce equivalence for inherently latency-sensitive operations.
Latency-sensitive <i>local</i> 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 <i>outside</i> the timing-sensitive region, so system-level $\approx$ 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 <i>never</i> 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 $\tau_P$ is the empty trace. The equivalence relation therefore holds vacuously.

For the purposes of the formal analysis, non-blocking message-passing is modeled at the level of committed transfers: a successful PushNB/PopNB contributes the same committed Push\_c/Pop\_c event to  $\Sigma$  as a blocking Push/Pop, while unsuccessful polls are  $\epsilon$ -steps.

If a design contains latency-sensitive global signals or non-blocking transfers whose timing/ordering is externally visible at the DUT boundary and must be reproduced exactly in both simulations, we take as an axiom that a purely synchronizing snooping wrapper can be applied to supply the same latency choices to the pre-HLS simulation as those observed in the post-HLS RTL (Appendix L). Conceptually, this wrapper is a witness-selection device: it does not enlarge the set of admissible pre-HLS behaviors, but restricts the pre-HLS run to one admissible execution whose  $\Sigma$ -events align with the observed RTL execution. Under this wrapper, the committed  $\Sigma$ -traces of the two simulations agree, and the equivalence rules E1–E5 apply to the wrapped runs.

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 the per-process witness correspondence needed by the later system-level proof: under a live environment, every post-HLS RTL\_B process trace (equivalently, every reachable finite observable prefix) has a matching source-level trace in Sys\_B ( $\text{Post} \rightarrow \text{B}$ ), satisfying E1–E5. The converse direction ( $\text{B} \rightarrow \text{Post}$ ) is not claimed for all Sys\_B traces in general; when it is needed, it is restricted to RTL-consistent prefixes (e.g., via Appendix L’s snooping / witness-selection technique).

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.,  $\text{P\_push}(c), \text{P\_pop}(c)$ ). 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 source-level bounded-FIFO interpretation Sys\_B (with capacities  $\text{B}(c)$  matching RTL\_B) is live under weak fairness, then RTL\_B is also live. (When  $\text{B}(c)=0$  for all channels, Sys\_B coincides with the rendezvous model.) 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  $\rightarrow$ , on the set of all observable IO actions ( $\Sigma$ ) across all processes in the system. The relation is causal (information-flow) rather than purely temporal: events related by  $\rightarrow$  may still commit in the same clock cycle (e.g., rendezvous), but the direction of  $\rightarrow$  indicates which event can be depended on by the other. This relation establishes a strict partial order of events, where  $a \rightarrow b$  is read as “a happens before b”.

The *happens-before* relation is the smallest relation satisfying the 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  $a \rightarrow b$ . 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 (committed transfer): If  $a$  is the commit of a send on channel  $c$  in process  $P1$  (a committed  $\text{Push}_c$ , including a successful  $\text{PushNB}$ ), and  $b$  is the commit of the corresponding receive on  $c$  in process  $P2$  (a committed  $\text{Pop}_c$ , including a successful  $\text{PopNB}$ ), for the same logical message, then  $a \rightarrow b$ . For rendezvous channels (i.e.,  $\text{B}(c)=0$ ), the matching  $\text{Push}_c$  and  $\text{Pop}_c$  commits occur in the same clock cycle; nevertheless we still include the directed edge  $a \rightarrow b$  to reflect unidirectional message



transfer (the Pop observes the value supplied by the Push) and to allow system-level causal chaining via transitivity. This edge does not assert an earlier commit cycle—only causal precedence—and no reverse edge ( $b \rightarrow a$ ) is implied unless there is an explicit reverse-direction communication (e.g., another channel or a signal handshake). Unsuccessful non-blocking polls are  $\epsilon$ -steps and do not participate in  $\rightarrow$ .

- Signal Handshake: If  $a$  is a signal write action  $w(\text{sig})$  in  $P1$  and  $b$  is a signal read action  $r(\text{sig})$  in  $P2$  that reads the value written by  $a$  as part of an explicit handshake protocol, then  $a \rightarrow b$ . A complete two-way handshake (e.g.,  $\text{vld}/\text{rdy}$ ) creates a causal chain, such as  $w(\text{vld})@P1 \rightarrow r(\text{vld})@P2 \rightarrow w(\text{rdy})@P2 \rightarrow r(\text{rdy})@P1$ .
- Explicit Synchronization (two-party SyncChannel /  $\text{ac\_sync}$ ): Let  $\text{sout}$  be the commit of a SyncChannel  $\text{sync\_out}$  (or equivalent  $\text{ac\_sync}$  call) in process  $P1$ , and let  $\text{sin}$  be the commit of the matching  $\text{sync\_in}$  in process  $P2$  for the same synchronization instance. Then this synchronization acts as a two-process barrier:
  - Every observable action in  $P1$  that occurs before  $\text{sout}$  (in  $P1$ 's trace/program order) satisfies  $a \rightarrow b$  for every observable action  $b$  in  $P2$  that occurs after  $\text{sin}$ .
  - Symmetrically, every observable action in  $P2$  that occurs before  $\text{sin}$  satisfies  $a \rightarrow b$  for every observable action  $b$  in  $P1$  that occurs after  $\text{sout}$ .
  - (The two sync commits  $\text{sout}$  and  $\text{sin}$  are treated as simultaneous; we do not impose  $\text{sout} \rightarrow \text{sin}$  or  $\text{sin} \rightarrow \text{sout}$ .)

3. Transitivity: The relation is transitive. If  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$ .

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  $s1 \rightarrow s2$ .

If neither  $s1 \rightarrow s2$  nor  $s2 \rightarrow s1$  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).

Fix any test stimulus (environment inputs) and initial state. Let  $\tau_{\text{B,system}}$  be the resulting observable trace of the source-level bounded-FIFO system  $\text{Sys\_B}$  under that stimulus, where  $\text{Sys\_B}$  denotes the same source-level processes as  $\text{Sys}$  but interpreted under E4 as abstract bounded FIFOs of capacity  $B(c)$  (the same  $B(c)$  as in  $\text{RTL\_B}$ , see Remark 2 in Appendix J); when  $B(c)=0$  this coincides with rendezvous system  $\text{Sys}$  under that stimulus, and let  $\tau_{\text{post,system}}$  be the resulting observable trace of  $\text{RTL\_B}$  under

the same stimulus. Assume every process  $P$  satisfies  $\tau_{\text{pre},P} \approx \tau_{\text{post},P}$  by Appendix I, every channel  $c$  has finite capacity  $B(c) \geq 0$ , the progress invariants  $P_{\text{push}}(c)$  and  $P_{\text{pop}}(c)$  hold for every  $c$ , and the system satisfies weak fairness (WF). Then the aggregate traces are equivalent:  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$  under rules E1–E5 (given the R rules and B rules).

*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  $\approx$ .

---

## 7. Practical Implication (“No Surprises” Guarantee)

If a verification environment exercises only the alphabet  $\Sigma$  (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. In effect, we define the rendezvous boundary *before* the skidbuffer.

It is possible to remove some or all the skidbuffers during Catapult HLS. Because the abstract rendezvous boundary is defined before the skidbuffer, this choice is orthogonal to the equivalence rules (R1–R5/E1–E5) as long as skidbuffer-internal behavior is not included in  $\Sigma$ . However, to keep the pre-HLS simulation's *port micro-architecture* aligned with the RTL configuration, the following compile flag must then be used in this case:

```
-DFORCE_AUTO_PORT=Connections::SYN_PORT
```

This will remove all the skidbuffers from the pre-HLS model. In this case, if some of the skidbuffers are still inserted during HLS, the capacity effects of those specific skidbuffers should be added into `Sys_B` so that the formal equivalence relationship still holds.

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 keep the pre-HLS simulation's *port micro-architecture* strictly aligned with the RTL configuration, then use `-DFORCE_AUTO_PORT=Connections::SYN_PORT`, and rerun final verification tests.

---

## Common Formal Definitions for Appendices I-K

Throughout Appendices I–K, the formal comparison is between `RTL_B` and `Sys_B`, i.e., the same source processes interpreted under bounded-FIFO semantics with capacities `B(c)` matching the RTL (E4). The rendezvous model is recovered as the special case `B(c)=0`.

Symbol / Rule	Definition — concise but complete
BoundaryReq( $m, t$ )	<p>For any <math>\Sigma</math>-visible message-passing operation <math>m \in \{\text{Push}_c(v), \text{Pop}_c(v)\}</math> and cycle <math>t</math>, BoundaryReq(<math>m, t</math>) means: at the chosen observable boundary, the endpoint is requesting that transfer at cycle <math>t</math>. Operationally (ready/valid):</p> <ul style="list-style-type: none"> <li>for <math>\text{Push}_c(v)</math>: the producer endpoint asserts <math>\text{vld}_c</math> (with stable payload as required), and</li> <li>for <math>\text{Pop}_c(v)</math>: the consumer endpoint asserts <math>\text{rdy}_c</math>.</li> </ul> <p>BoundaryReq is therefore a predicate about the endpoint's boundary-visible request assertion (Push: producer asserts <math>\text{vld}</math> with stable payload; Pop: consumer asserts <math>\text{rdy}</math>). BoundaryReq refers to the endpoint-owned request signal driven by the process (Push: <math>\text{vld}</math>; Pop: <math>\text{rdy}</math>). Purely combinational realization structure on the channel side (including couplers) may affect only the complementary mirror signal at that boundary (Push: <math>\text{rdy}</math>; Pop: <math>\text{vld}</math>) and thus the commit condition after <math>\epsilon</math>-settling, but it does not change the endpoint request assertion captured by BoundaryReq(<math>m, t</math>).</p>
ChannelEnabled( $m, t$ )	<p>ChannelEnabled(<math>m, t</math>) means: BoundaryReq(<math>m, t</math>) holds, and in the <math>\epsilon</math>-quiescent (settled) ready/valid fixed point of cycle <math>t</math>, the channel-side commit-enabling condition for <math>m</math> holds at the chosen boundary (i.e., <math>m</math> is not blocked by the channel's availability/back-pressure conditions under the Sys_B / E4 interpretation). Concretely, for bounded-FIFO channels <math>B(c) &gt; 0</math>, the channel-side condition is exactly the E4 availability predicate at that boundary (space for Push / data for Pop); rendezvous is the special case <math>B(c) = 0</math>.</p>
ChannelDisabled( $m, t$ )	<p>ChannelDisabled(<math>m, t</math>) means: BoundaryReq(<math>m, t</math>) holds but ChannelEnabled(<math>m, t</math>) does not. Intuitively: the endpoint is requesting the transfer, but the transfer is blocked by channel-side conditions (e.g., empty/full or the rendezvous partner not simultaneously requesting), as evaluated at the <math>\epsilon</math>-quiescent handshake fixed point used throughout the document's enablement tests (e.g., Appendix K).</p> <p>In particular, "couplers" (being purely combinational ready/valid logic) can affect the settled handshake outcome (and thus ChannelEnabled/ChannelDisabled) by driving only the complementary mirror handshake signals at the chosen boundary (Push: <math>\text{rdy}</math>; Pop: <math>\text{vld}</math>) consistently with E4; they do not change the abstract <math>\text{occ}_c(t)</math> / <math>B(c)</math> meaning of E4. Consequently, at the chosen boundary the channel-side enablement test remains exactly E4: for any <math>\Sigma</math>-visible message operation <math>m</math> on an abstract channel <math>c</math>, if BoundaryReq(<math>m, t</math>) holds and <math>c</math>'s E4 availability predicate holds at cycle <math>t</math> (space/data or rendezvous participation, as applicable), then ChannelEnabled(<math>m, t</math>) holds in the <math>\epsilon</math>-quiescent handshake fixed point. Any cross-channel dependency introduced by combinational couplers must be reflected only at explicit internal staged/bundle/join boundaries introduced by the elaboration (i.e., via BoundaryReq/ChannelEnabled on those explicit staged/bundle channels); at the chosen <math>\Sigma</math>-visible channel endpoints, BoundaryReq and ChannelEnabled/ChannelDisabled for channel <math>c</math> may not depend on other channels' state beyond the per-channel E4 predicate for <math>c</math>,</p>

Symbol / Rule	Definition — concise but complete
	and no extra hidden predicate may block an otherwise E4-enabled boundary transfer.
$\Sigma$	<p>Set of observable actions for the equivalence check (committed IO events) at the chosen observable boundary (often the DUT IO boundary for functional equivalence, but optionally expanded—e.g., for Appendix K deadlock reasoning):</p> <ul style="list-style-type: none"> <li>• channel commit events Push<sub>c</sub>(v) and Pop<sub>c</sub>(v) on channels designated as observable, labeled with the transferred message value (or abstract message identity) v</li> <li>• synchronization events wait(), Sync, START_P, FINISH_P</li> <li>• single-cycle signal actions Write(sig, val) and Read(sig, val) on signals designated as observable, labeled with the value written/read.</li> </ul> <p>Internal-only channels/signals may be excluded from <math>\Sigma</math> (treated as <math>\epsilon</math>-microstate) when they are not part of the chosen observable boundary. If such channels/signals are later brought into the observable boundary (e.g., by snooping for debug or analysis), then they become <math>\Sigma</math>-events and must match under <math>\approx</math>.</p> <p>Deadlock-analysis requirement (Appendix K): for Appendix K, <math>\Sigma</math> is instantiated to include every message-passing channel whose <math>\Sigma</math>-visible endpoint transfers at the chosen Sys_B boundary (Push<sub>c</sub>/Pop<sub>c</sub>) can contribute a dependency edge in the Appendix K Wait-For Graph, either because it is on the declared observable boundary or because it is snooped. Internal micro-interfaces that lie within the back-annotated realization counted in B(c) (e.g., post-HLS RTL pipeline stage implementation) are not treated as separate channels for this requirement: they may be exposed/snooped for debug, but Appendix K's WFG edges are defined using the ChannelEnabled/ChannelDisabled status of the corresponding boundary Push<sub>c</sub>/Pop<sub>c</sub> (via E4/occ<sub>c</sub>/B(c)), so such internal points can participate logically in the deadlock argument without being explicitly <math>\Sigma</math>-observable.</p> <p>Non-blocking message-passing (PushNB/PopNB): A non-blocking call that returns “not completed” produces no <math>\Sigma</math>-event and is modeled as an internal <math>\epsilon</math>-step.</p> <p>When a non-blocking call succeeds (returns “completed”), that success is represented in <math>\Sigma</math> as the corresponding committed Push<sub>c</sub>(v) or Pop<sub>c</sub>(v) event on that channel, labeled with the same transferred value/message v (i.e., <math>\Sigma</math> records transfers and their payloads, not failed polls). A non-blocking call is considered issued in the cycle in which it executes (its Issue cycle). A call that returns “completed” commits in that same cycle and therefore yields a committed transfer <math>m(q) \in M</math>; a call that returns “not completed” produces no committed transfer event (<math>m(q)</math> undefined / absent) and may be treated as an <math>\epsilon</math>-step with respect to <math>\Sigma</math>-visible message-transfer events.</p>

Symbol / Rule	Definition — concise but complete
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.
$\epsilon$ -step	An <i>internal</i> , unobservable RTL state transition (pipeline advance, FSM micro-state, etc.).
B(c)	<p>Back-annotated effective capacity of channel c at the chosen <math>\Sigma</math>-observable Sys_B boundary (the same boundary used by E4 and Appendix K). Finite, <math>B(c) \geq 0</math>. It reflects the total bounded “composite buffer” storage/credit in the RTL realization between the producer’s and consumer’s <math>\Sigma</math>-visible endpoints (e.g., FIFO storage, skid/elastic buffering, and any HLS-inserted pipeline staging that can hold in-flight messages).</p> <ul style="list-style-type: none"> <li>• <math>B(c)=0 \Rightarrow</math> rendezvous (capacity-zero) channel.</li> <li>• <math>B(c)&gt;0 \Rightarrow</math> bounded FIFO (of that effective capacity).</li> </ul> <p>Point-to-point endpoint assumption (single-producer/single-consumer): For every message-passing channel c (including both buffered channels with <math>B(c)&gt;0</math> and rendezvous channels with <math>B(c)=0</math>), there is exactly one producer process that may execute Push_c on c, and exactly one consumer process that may execute Pop_c on c. Hence the complementary endpoint relevant to any blocked Push_c/Pop_c on c is unique.</p> <p>Modeling note (shared resources / arbitration): If an implementation conceptually has multiple producers and/or multiple consumers for a logical resource, that sharing must be modeled explicitly (e.g., an arbiter process plus per-client point-to-point channels), rather than as a single multi-endpoint “channel.” This keeps WFG wait dependencies well-defined and unique per edge.</p> <p>Single-transfer-per-cycle endpoint constraint (scalar channel interface): For every message-passing channel c, in any clock cycle t, at most one Push_c may commit and at most one Pop_c may commit. Equivalently, the set of commits on a fixed channel in a single cycle contains <math>\leq 1</math> Push and <math>\leq 1</math> Pop (it may contain one of each in the same cycle).</p> <p>Modeling note (multi-lane / burst transfers): If an implementation can transfer <math>k&gt;1</math> messages per cycle on a logical resource, model it as k parallel point-to-point channels (or as a single widened message transferred by one Push_c/Pop_c per cycle) so that the <math>\Sigma</math>-level event model continues to satisfy the scalar constraint above.</p>
occ_c(t)	<p>Abstract FIFO occupancy at the Sys_B channel boundary: number of items committed by Push_c but not yet committed by Pop_c at that boundary. Equivalently (E4), <math>occ\_c(t) = push(\pi\_t) - pop(\pi\_t)</math>, where <math>\pi\_t</math> is the <math>\Sigma</math>-visible boundary-commit prefix strictly before cycle t (i.e., excluding any Push_c/Pop_c commits that occur in cycle t), so occ_c(t) is the abstract occupancy at the beginning of cycle t. For Pure-FIFO channels with <math>B(c)&gt;0</math> (Appendix H), the FIFO availability predicates (not-full/not-empty) are evaluated against this begin-of-cycle occupancy; hence a Pop_c cannot commit</p>

Symbol / Rule	Definition — concise but complete
	<p>in a cycle that begins empty solely because a Push_c also commits in that same cycle (no fall-through), and likewise a Push_c cannot commit in a cycle that begins full solely because a Pop_c also commits in that same cycle.</p> <p>Clarification (no “second Push/Pop”): movements of an item within the RTL channel realization (e.g., FIFO→staging/skid buffering, staging/skid buffering→pipeline-stage registers, pipeline-stage→pipeline-stage handoff) are internal <math>\epsilon</math>-steps; they do not constitute additional <math>\Sigma</math>-visible committed Push_c/Pop_c events beyond the boundary events counted by <math>\text{push}(\pi_t)/\text{pop}(\pi_t)</math>.</p> <p>Note: <math>\text{occ}_c(t)</math> is an abstract analysis quantity derived from <math>\Sigma</math>-visible boundary commits; in both Sys_B and RTL_B, processes do not directly read <math>\text{occ}_c(t)</math> and instead proceed solely based on the per-channel ready/valid outcome of their own Push/Pop attempts.</p>
P_push(c)	<p><i>Finite-progress invariant (producer)</i>: interpret “continuously enabled” as a persistent (non-withdrawable; no deassert-before-commit) ready/valid request.</p> <ul style="list-style-type: none"> <li>• Buffered case (<math>B(c) &gt; 0</math>): If the producer’s Push_c request remains asserted continuously from some cycle <math>t_0</math> onward while the channel is full (<math>\text{occ}_c = B(c)</math>)—with stable payload while asserted—and the complementary endpoint continuously requests the matching Pop_c (persistent request asserted; Pop_c boundary request remains true), then some cycle <math>t' \geq t_0</math> commits a complementary Pop_c (i.e., the full condition is eventually discharged).</li> <li>• Rendezvous case (<math>B(c) = 0</math>): If the producer’s Push_c request remains asserted continuously from some cycle <math>t_0</math> onward (with stable payload while asserted) and the complementary endpoint continuously requests the matching Pop_c (persistent request asserted; Pop_c boundary request remains true), then some cycle <math>t' \geq t_0</math> commits the rendezvous transfer—i.e., a matching Push_c(v) and Pop_c(v) commit in the same cycle.</li> </ul>
P_pop(c)	<p><i>Finite-progress invariant (consumer)</i>: interpret “continuously enabled” as a persistent (non-withdrawable; no deassert-before-commit) ready/valid request.</p> <ul style="list-style-type: none"> <li>• Buffered case (<math>B(c) &gt; 0</math>): If the consumer’s Pop_c request remains asserted continuously from some cycle <math>t_0</math> onward while the channel is empty (<math>\text{occ}_c = 0</math>)—and the complementary endpoint continuously requests the matching Push_c (persistent request asserted with stable payload; Push_c boundary request remains true)—then some cycle <math>t' \geq t_0</math> commits a complementary Push_c (i.e., the empty condition is eventually discharged).</li> <li>• Rendezvous case (<math>B(c) = 0</math>): If the consumer’s Pop_c request remains asserted continuously from some cycle <math>t_0</math> onward and the complementary endpoint continuously requests the matching Push_c (persistent request asserted with stable payload; Push_c boundary request remains true), then some cycle <math>t' \geq t_0</math> commits the rendezvous transfer—i.e., a matching Push_c(v) and Pop_c(v) commit in the same cycle.</li> </ul>

Symbol / Rule	Definition — concise but complete
Equivalence rules (E1–E5)	<p><i>E1 Synchronization-order preservation:</i> For each process P, the sequence of synchronization events executed by P (wait(), SyncChannel, START_P, FINISH_P) appears in the same source order in the pre-HLS and post-HLS traces.</p> <p><i>E2 Signal visibility:</i> For each process P, each signal Read in P occurs at the closest preceding synchronization event in P, and each signal Write in P occurs at the closest succeeding synchronization event in P, in both the pre-HLS and post-HLS traces (per R2/E2) (except explicitly declared direct-input exceptions). Interpretation (logical timestamping). The equations in E2 define the logical timestamps of Read/Write <math>\Sigma</math>-events (the anchor points at which equivalence is checked). An implementation may physically sample a Read later than pred_S(r) (or physically apply a Write earlier than succ_S(w)) provided the value is stable across the anchor interval and the trace emitted for equivalence records the <math>\Sigma</math>-event at the anchor point (see the Instrumentation note in Appendix G).</p> <p>Direct-input pragmas (e.g., #pragma hls_direct_input and #pragma hls_direct_input_sync) are not exceptions to E2: they impose stability/timing contracts that allow late physical sampling (or controlled updates) while the logical Read/Write <math>\Sigma</math>-events used for <math>\approx</math> checking remain timestamped at the E2 anchor points (see Appendix G, Direct Inputs and Instrumentation note).</p> <p><i>E3 (Safe message issue ordering). (Post-HLS preservation of R4).</i>  For any two message operations <math>q_1, q_2 \in Q</math> issued by the same process:  (i) Same-interface order is always preserved: if <math>q_1</math> and <math>q_2</math> access the same message-passing interface/channel and <math>q_1 &lt;_{\text{src}} q_2</math>, then <math>\text{issue\_post}(q_1) \leq \text{issue\_post}(q_2)</math>.  (ii) Distinct-interface no-reverse: if <math>q_1</math> and <math>q_2</math> access distinct message-passing interfaces and appear in sequence in the source (<math>q_1 &lt;_{\text{src}} q_2</math>), then <math>\text{issue\_post}(q_1) \leq \text{issue\_post}(q_2)</math> (i.e., they shall not be issued in the reverse sequence).  (Note: any pipelined-loop internal staging/dequeue discussed in “Pipelined Loops” is <math>\epsilon</math>-microstate within the back-annotated channel realization and is not a <math>\Sigma</math>-visible boundary issue event in Q nor a committed-transfer event in M; therefore it does not constitute a counterexample to (ii).)</p> <p><i>E4 FIFO legality:</i> for each channel c, the post-HLS committed (Push_c(v), Pop_c(v)) history is a legal bounded-capacity execution consistent with Sys_B for that channel (capacity B(c)), and reduces to rendezvous consistency when B(c)=0. In particular, on each channel, the committed Pop_c(v) events return exactly the FIFO-ordered sequence of values v previously committed by Push_c(v), with no drops, duplications, or reordering.</p> <p><i>E5 Side-of-Sync guarantee:</i> a message action never moves across its bounding synchronization event (START_P, FINISH_P, explicit wait/Sync).</p>

#### 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 emitted by the post-HLS design is unique, including the channel identity and payload/value of each committed $\text{Push}_c(v)/\text{Pop}_c(v)$ and the value of each $\text{Write}(\text{sig}, \text{val})/\text{Read}(\text{sig}, \text{val})$ . Internal $\epsilon$ -steps may differ between runs, but the externally visible $\Sigma$ -trace (with labels) cannot. Clarification: B1 is assumed of $\text{RTL\_B}$ (post-HLS) only. The source-level models $\text{Sys}/\text{Sys\_B}$ may admit multiple $\epsilon$ -/latency-different executions with the same $\Sigma$ -projection. When a single concrete witness execution of $\text{Sys\_B}$ is required for simulation/debug, Appendix L's snooping wrapper may be used to select one admissible witness whose latency-sensitive events align with the unique $\text{RTL } \Sigma$ -trace.	Ensures the per-process and system-level equivalence proofs (App. I & J) can match <i>one</i> post-HLS trace to <i>one</i> 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 $B(c)$ : <ul style="list-style-type: none"> <li>• <math>P_{\text{push}}(c)</math>: If a process <math>P</math> is stalled at a blocking <math>\text{Push}_c</math> with its boundary request true and a persistent request asserted (payload stable, if applicable) while the channel is full (<math>\text{occ}_c = B(c)</math>), and the complementary endpoint process continuously requests the matching <math>\text{Pop}_c</math> (persistent request asserted; <math>\text{Pop}_c</math> boundary request remains true), then the transfer on <math>c</math> must eventually complete (in particular: for <math>B(c) &gt; 0</math>, some <math>\text{Pop}_c</math> eventually commits, freeing space; for <math>B(c) = 0</math>, the rendezvous completion eventually occurs).</li> <li>• <math>P_{\text{pop}}(c)</math>: If a process <math>P</math> is stalled at a blocking <math>\text{Pop}_c</math> with its boundary request true and a persistent request asserted while the channel is empty (<math>\text{occ}_c = 0</math>), and the complementary endpoint process continuously requests the matching <math>\text{Push}_c</math> (persistent</li> </ul>	Explicitly assumed in Appendix J to rule out permanent back-pressure loops in which both endpoints continuously request a transfer but it never completes, which are logically independent of the cycle-by-cycle R-rules.

ID	Basic-process property (informal statement)	Why it is needed / how it is used
	request asserted with stable payload; $\text{Push}_i c$ , boundary request remains true), then the transfer on $c$ must eventually complete (in particular: for $B(c)>0$ , some $\text{Push}_i c$ , eventually commits, providing data; for $B(c)=0$ , the rendezvous completion eventually occurs).	
B4 Finite Stutter Bound	Every process $P$ has a finite constant depth $D_P$ such that whenever $P$ is not externally stalled (i.e., $P$ is advancing internal micro-state toward its next observable $\Sigma$ -action, or its next $\Sigma$ -action is locally enabled), $P$ can execute at most $D_P$ consecutive $\epsilon$ -steps before either (i) executing a $\Sigma$ -action, or (ii) reaching a stable waiting state in which $P$ has no further $\epsilon$ -step available until some external/peer/environment condition changes the enabling predicates.	This guarantees the $\epsilon$ -stutter closure needed to construct witness mappings in Appendix I and to bound internal micro-latency (pipeline/FSM bookkeeping) by $\leq D_P$ . Unbounded waiting due to back-pressure or missing peer stimulus is governed by WF/B2 and the progress obligations B3, not by B4.
B5 System Quiescence Closure	If, at some cycle $t_0$ , 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 $\epsilon$ -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

This appendix establishes a per-process witness property in the direction needed by Appendix J's execution-level correspondence: under a fixed initial state and fixed external stimulus/environment behavior, every reachable finite observable prefix of the post-HLS model  $\text{RTL\_B}$  has a matching finite observable prefix in the source-level bounded-FIFO interpretation  $\text{Sys\_B}$ , satisfying E1–E5.

Importantly, the converse direction ("every  $\text{Sys\_B}$  trace has a matching  $\text{RTL\_B}$  trace") is NOT claimed in general when latency-sensitive / non-blocking effects are made  $\Sigma$ -visible (e.g., by observing non-blocking poll outcomes or other timing-sensitive events via an expanded observation boundary). In such cases,  $\text{Sys\_B}$  may admit additional  $\Sigma$ -traces (for example, extra unsuccessful poll observations) that do not match the  $\text{RTL\_B}$   $\Sigma$ -trace under the same stimulus. When a reverse correspondence is required, Appendix L's snooping / witness-selection technique is used to restrict attention to RTL-consistent  $\text{Sys\_B}$  executions (see Corollary J.1).

Rendezvous channels are the special case  $B(c)=0$ .

## I.2 Formal Preconditions

- Observable alphabet  $\Sigma$  is as defined in *Common Formal Definitions for Appendices I–K*: it consists of the event schemas {Push\_c, Pop\_c, Sync, wait, START\_P, FINISH\_P, Write, Read} instantiated only for channels/signals designated observable; additionally, all message-passing channels that can participate in Appendix K deadlock reasoning are designated observable (possibly via snooping) and hence included in  $\Sigma$ .
- Non-blocking message-passing interpretation. A successful non-blocking PushNB/PopNB contributes the same observable event as the corresponding committed Push\_c/Pop\_c on that channel. An unsuccessful non-blocking call (returns “false” / “not completed”) contributes no  $\Sigma$ -event and is modeled as an  $\epsilon$ -step. Nevertheless, non-blocking polling may still assert a  $\Sigma$ -visible endpoint request; issuance is attributed using the Issue/Commit linkage conditions in the Issue definition above (stable attribution while outstanding): a continuously asserted endpoint request with no intervening commit is attributed to one issued request  $q \in Q$  (with Issue( $q$ ) at the first asserted cycle, or at  $t+1$  immediately following the previous commit if the request never deasserts), and only a poll that succeeds contributes the corresponding committed transfer  $m(q) \in M$ . If multiple poll checks occur while the endpoint request remains continuously asserted, they are  $\epsilon$ -steps within the same  $q$  (no additional  $q \in Q$ ).
- Silent step Any internal RTL microstate transition is written  $\epsilon$ . In particular, unsuccessful non-blocking polls, arbitration bookkeeping, and pipeline advances are  $\epsilon$ -steps.
- Finite-pipeline-depth premise (FPD) There exists a finite constant  $\text{pipe\_depth}(P) = D_P < \infty$  such that once  $P$  begins internal micro-progress toward its next observable  $\Sigma$ -action (or that next  $\Sigma$ -action is locally enabled),  $P$  executes at most  $D_P$  cycles of  $\epsilon$ -steps before either committing a  $\Sigma$ -action or reaching a stable waiting state with no further  $\epsilon$ -step available until external/peer conditions change. In particular, a process cannot perform an unbounded number of  $\epsilon$ -only failed non-blocking polls while making no progress toward any  $\Sigma$ -action; designs that can spin indefinitely without reaching either a  $\Sigma$ -action or a stable wait state violate FPD/B4 and are outside the model.
- Weak-fairness premise (WF) If a micro-operation remains continuously enabled, the scheduler eventually issues it. This is an assumption on the execution/scheduling environment (See Appendix N), not an automatic guarantee of “clock-synchronous RTL.” In practice it requires that any arbiters/back-pressure logic that can affect whether an enabled operation is selected must be fair in the sense of B2/B3.
- Finite-progress invariants (B3) For every channel  $c$ , assume producer and consumer obligations  $P_{\text{push}}(c)$  and  $P_{\text{pop}}(c)$  hold, guaranteeing that data (or space) eventually becomes available. This is an assumption on the execution/scheduling environment (e.g., fairness of arbiters/back-pressure; see Appendix N), not a consequence of the cycle-by-cycle R-rules.
- Pure-FIFO channel semantics. For any message-passing channel  $c$  with  $B(c) > 0$ : if a process has issued a persistent request to transfer on  $c$  (i.e., the request remains asserted and the payload is stable, if applicable), then the channel-side commit-enabling predicate is exactly the FIFO availability predicate—namely:
  - for Push\_c:  $\text{occ}_c < B(c)$  (“not full”), and
  - for Pop\_c:  $\text{occ}_c > 0$  (“not empty”).
  - Modeling note (fall-through / simultaneous empty/full cases). This abstraction is cycle-boundary, non-fall-through for  $B(c) > 0$ : however, it does permit a Push and a Pop to both commit in the same cycle when the FIFO begins the cycle neither empty nor full ( $0 < \text{occ}_c(t) < B(c)$ ); in that case both commits are legal and the occupancy is unchanged. It does not permit a Pop to commit in a cycle that begins empty solely because a Push also commits in that same cycle, nor does it permit a Push to commit in a cycle that begins

full solely because a Pop also commits in that same cycle. If an implementation uses a fall-through FIFO (allowing same-cycle pass-through when empty), model that behavior explicitly (e.g., as an explicit bypass/rendezvous path composed with the bounded FIFO), so that the  $\Sigma$ -level committed-transfer trace matches the intended cycle semantics.

- Operational interpretation (ready/valid):  $\text{occ}_c(t)$  is an abstract quantity derived from  $\Sigma$ -visible boundary commits; processes do not read  $\text{occ}_c(t)$  directly. In both Sys\_B and RTL\_B, whether a pending blocking Push/Pop can proceed is realized operationally by the settled per-cycle ready/valid handshake outcome (evaluated in the  $\epsilon$ -quiescent fixed point). A concrete realization (including inserted staging and purely combinational couplers) is correct iff, for a persistent request with BoundaryReq true, the ready/valid outcome agrees with the corresponding availability predicate above at the chosen boundary.

No additional arbitration/grant/back-pressure gating is part of the channel semantics; any cross-channel coordination in the realization (including couplers) must be expressed only through the settled ready/valid handshake behavior that determines ChannelEnabled/ChannelDisabled, without adding enablement conditions beyond E4. Process-local control (BoundaryReq) determines when an endpoint requests; realization logic determines whether that request is enabled (ChannelEnabled) in that cycle.

Therefore, when the boundary request is true and the relevant FIFO availability predicate holds continuously, the transfer is continuously enabled; by weak fairness (B2) the corresponding commit occurs after a finite (though not necessarily bounded) delay.

- 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).
- Fixed-stimulus interpretation (reactive environments). If the “stimulus” is produced by a reactive testbench/environment, interpret “fixed stimulus” as fixing the environment’s behavior as a  $\Sigma$ -causal function of the observed  $\Sigma$ -history—equivalently, include the environment as part of the closed system being executed, with its own fixed initial state and fixed nondeterministic choices. Under this interpretation, whenever two runs have matched  $\Sigma$ -prefixes, they are subject to the same subsequent environment behavior.
- Non-blocking message passing interpretation. In the post-HLS model, a non-blocking call (PopNB / PushNB) is a poll: if it fails (no data / no space / rendezvous partner not ready), it produces no observable  $\Sigma$  action and corresponds to an  $\epsilon$ -step at the per-process trace level. If it succeeds, it produces the corresponding committed Pop\_c / Push\_c  $\Sigma$ -event. In the source-level bounded-FIFO model Sys\_B, we treat non-blocking polls analogously: an unsuccessful poll contributes only  $\epsilon$ , and the first successful completion is the  $\Sigma$ -visible committed transfer.
- A non-blocking poll may still assert a  $\Sigma$ -visible endpoint request; we attribute issuance using the Issue/Commit linkage conditions in the Issue definition above (stable attribution while outstanding), so that a continuously asserted endpoint request with no intervening commit is attributed to one issued request  $q \in Q$  (with Issue( $q$ ) at the first asserted cycle, or at  $t+1$  immediately following the previous commit if the request never deasserts), and only a poll that succeeds yields the corresponding committed transfer  $m(q) \in M$ . Repeated poll checks under a continuously asserted request are  $\epsilon$ -steps within that same  $q$ .
  - NB-CF (Non-blocking control-flow condition). Because failed polls are modeled as  $\epsilon$  (unobservable), the Appendix I Post $\rightarrow$ B construction assumes that success/failure outcomes of  $\epsilon$ -modeled non-blocking polls do not influence the next  $\Sigma$ -visible control flow / next  $\Sigma$ -visible blocking frontier. Equivalently: between two consecutive  $\Sigma$ -events of a process, inserting/removing any finite sequence of failed PopNB/PushNB polls may

- change only internal  $\epsilon$ -behavior/timing, not which  $\Sigma$ -visible action (Push/Pop/synchronization anchor) is next in program order.
- If a design does intentionally branch on non-blocking outcomes in a way that changes the next  $\Sigma$ -visible behavior, then the development must use witness-selected NB outcomes (Post $\rightarrow$ B). In this case we strengthen the Post $\rightarrow$ B correspondence to range only over RTL-consistent Sys\_B witnesses in which the permitted non-blocking outcomes are constrained/selected to match the observed RTL\_B run (e.g., via the Appendix L explicit witness-selection/snooping mechanism that fixes the poll outcomes seen by Sys\_B to those consistent with  $\tau_{\text{post}}$ ).

---

### I.3 Auxiliary Lemmas

Lemma I.0 (Message-issue discipline in RTL).

The RTL satisfies E3: (i) per-interface issue order is preserved for all message operations; and (ii) for distinct interfaces, the no-reverse rule holds.

Lemma I.1 (Bounded  $\epsilon$ -chain to  $\Sigma$ -action or stable wait). Starting from any state of P, within at most  $D_P$  clock cycles P either (i) commits its next observable  $\Sigma$ -action, or (ii) reaches a stable waiting state with no further  $\epsilon$ -step available until some external/peer condition changes the enabling predicates.

*Proof.* Direct from FPD/B4. ■

Lemma I.2 (Eventual space / data). Assume P is blocked on  $\bullet$  Push\_c with  $\text{occ\_c} = B(c)$ , and the complementary endpoint continuously requests the matching Pop\_c (persistent request asserted; Pop\_c boundary request remains true), or  $\bullet$  Pop\_c with  $\text{occ\_c} = 0$ , and the complementary endpoint continuously requests the matching Push\_c (persistent request asserted with stable payload; Push\_c boundary request remains true). Then a complementary Pop\_c (respectively Push\_c) commits within finite time.

*Proof.* (By B3 / Appendix N.) In either case, P is stalled at a blocking channel call while its persistent request for that call remains asserted (payload stable, if applicable). By the additional premise, the complementary endpoint also continuously requests the matching operation with its boundary request remaining true. Therefore the premises of P\_push/P\_pop (Appendix N) hold, and the corresponding transfer must eventually complete: for Push\_c at full, some complementary Pop\_c eventually commits (freeing space); for Pop\_c at empty, some complementary Push\_c eventually commits (providing data).

■

---

### I.4 Inductive Construction for Finite Traces (Post $\rightarrow$ B)

Let

$$\tau_{\text{post}} = \tau_{\text{post}}[0..n-1] \circ e \quad (\text{where } |\tau_{\text{post}}| = n + 1)$$

be the next postHLS prefix of RTL\_B for process P (under the fixed initial state and fixed stimulus).

Assume by induction that we already have a matching Sys\_B prefix  $\tau_B[0..n-1]$  satisfying E1–E5 with  $\tau_{\text{post}}[0..n-1]$ . We extend  $\tau_B$  by a finite  $\epsilon$ -chain (written  $\epsilon^*$ ) followed by an observable action  $e'$  so that

$$\tau_B \circ \epsilon^* \circ e' \text{ matches } \tau_{\text{post}}.$$

Non-blocking note. For a non-blocking PopNB/PushNB, unsuccessful polls are treated as  $\epsilon$ -steps and handled by the existing “ $\epsilon$  internal step” rows; the first successful completion is the corresponding committed Pop\_c/Push\_c  $\Sigma$ -event and is handled by the Pop/Push rows above. This is sound for the Post $\rightarrow$ B construction provided NB-CF holds (I.2):  $\epsilon$ -modeled poll outcomes do not alter the next  $\Sigma$ -visible control flow / next  $\Sigma$ -visible blocking frontier.

If NB-CF does not hold (i.e., a process branches on the success/failure of a non-blocking poll in a way that changes its next  $\Sigma$ -visible behavior), then the  $\text{Post} \rightarrow B$  construction must instead be carried out with an explicit RTL-consistent witness-selection restriction on  $\text{Sys}_B$  executions (so that the  $\text{Sys}_B$  witness chosen for  $\text{Post} \rightarrow B$  is constrained to take poll outcomes consistent with  $\tau_{\text{post}}$ , e.g., via snooping as in Appendix L).

Kind of event $e$	Why $\epsilon^*$ is finite
Sync, wait, START_P, FINISH_P, Write, Read	Internal pipeline/microstate progress for P to reach the blocked state at this operation is bounded by $\leq D_P \epsilon$ -steps (Lemma I.1). Completion of Sync (or wait) may then require peer/environment enabling and may therefore be delayed by an a priori unbounded number of cycles while P is externally stalled. However, under the fixed-stimulus comparison (and $\Sigma$ -causal reactive-environment interpretation), once $\tau_B$ and $\tau_{\text{post}}$ have matched the $\Sigma$ -prefix so far the same subsequent peer/environment enabling that leads to $e$ in $\text{RTL}_B$ must also occur for $\text{Sys}_B$ . Once completion is continuously enabled, weak fairness (B2) guarantees that the scheduler selects it after a further finite (though not necessarily bounded) delay. Thus $e'$ occurs after a finite (though not necessarily bounded) delay.
Push_c	<ul style="list-style-type: none"> <li>• <math>\text{issue}_B(e')</math> occurs when P reaches and issues the Push_c operation (i.e., when Push_c is one of P's next enabled <math>\Sigma</math>-actions with its boundary request true). Lemma I.0 guarantees the required source-order issue discipline for message operations (E3).</li> <li>• If <math>B(c) &gt; 0</math> and the transfer's commit-enabling predicate holds continuously (pure-FIFO: in particular <math>\text{occ}_c &lt; B(c)</math> for Push_c), then the transfer is continuously enabled; by weak fairness (B2) the scheduler selects it after a finite (though not necessarily bounded) delay, so the commit occurs after finitely many <math>\epsilon</math>-steps.</li> <li>• If <math>B(c) &gt; 0</math> and <math>\text{occ}_c = B(c)</math>: Lemma I.2 frees space.</li> <li>• If <math>B(c) = 0</math>: rendezvous fires when both sides are enabled; Lemma I.2 guarantees eventual completion given both sides continuously request.</li> </ul>
Pop_c	Symmetric to Push_c (pure-FIFO: for Pop_c with $B(c) > 0$ , the availability predicate is $\text{occ}_c > 0$ ).

Each case ensures  $\epsilon^*$  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 ( $\text{Post} \rightarrow B$ )

Fix a process P and a fixed external stimulus/initial state. (Clarification for reactive environments: if the “stimulus” is produced by a reactive testbench/environment, interpret “fixed stimulus” as fixing the environment's behavior as a  $\Sigma$ -causal function of the observed  $\Sigma$ -history—equivalently, include the environment as part of the closed system being executed, with its own fixed initial state and fixed nondeterministic choices. Under this interpretation, whenever two runs have matched  $\Sigma$ -prefixes, they are subject to the same subsequent environment behavior; this is the same interpretation used later in Corollary J.1.)

Let  $\tau_{\text{post},P}$  be the (countably infinite) observable trace of P in  $\text{RTL}_B$  under this stimulus (restricted, as in Appendix J / Corollary J.1, to traces that are prefixes of at least one fair/progress-satisfying infinite

run). We construct an infinite Sys\_B trace  $\tau_{B,P}$  by an explicit inductive limit of the finite-prefix construction in I.4.

Let  $\tau_{B,P}[0..0)$  be the empty prefix. For each  $n \geq 0$ , assume we have constructed a finite Sys\_B prefix  $\tau_{B,P}[0..n)$  such that its  $\Sigma$ -projection matches the first  $n$  observable events of  $\tau_{\text{post},P}$  and the correspondence satisfies E1–E5. Let  $e_n$  be the  $(n+1)$ -st  $\Sigma$ -event of  $\tau_{\text{post},P}$ . Apply the I.4 construction step to extend  $\tau_{B,P}[0..n)$  by a finite  $\varepsilon^*$  segment followed by a matching observable event  $e'_n$ , obtaining  $\tau_{B,P}[0..n+1)$  while preserving E1–E5.

Because each extension step adds a finite segment, the increasing chain of prefixes defines a (countably infinite) Sys\_B trace  $\tau_{B,P}$  whose every finite prefix matches the corresponding prefix of  $\tau_{\text{post},P}$ . Therefore,  $\tau_{B,P} \approx \tau_{\text{post},P}$  (for countably infinite traces as well). ■

## Appendix J – System-Level Trace Equivalence Proof

### Theorem J.1 (Compositional Equivalence)

Fix any test stimulus (environment inputs) and initial state. Let  $\tau_{B,\text{system}}$  be an observable trace of the source-level bounded-FIFO system Sys\_B under that stimulus, and let  $\tau_{\text{post},\text{system}}$  be an observable trace of RTL\_B under the same stimulus. (If B1 holds, RTL\_B is  $\Sigma$ -deterministic for the given stimulus/initial state.) Assume every process  $P$  satisfies  $\tau_{B,P} \approx \tau_{\text{post},P}$  by Appendix I, where  $\tau_{B,P}$  is the projection of the Sys\_B system trace  $\tau_{B,\text{system}}$  onto  $P$ , every channel  $c$  has finite capacity  $B(c) \geq 0$ , the progress invariants  $P_{\text{push}}(c)$  and  $P_{\text{pop}}(c)$  hold for every  $c$ , and the system satisfies weak fairness (WF). Assume also B1 (Deterministic Trace Property) when uniqueness of the trace is relied upon. Then the aggregate traces are equivalent:  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$  under rules E1–E5 (given the R rules and B rules).

Notation: Let  $<_{\text{src}}$  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  $\text{occ\_post}(c,t)$  satisfies  $0 \leq \text{occ\_post}(c,t) \leq B(c)$ .

*Proof of Lemma J.1:*

We argue by cases on  $B(c)$ .

If  $B(c) > 0$  (buffered FIFO): RTL never de-queues from an empty FIFO and never en-queues into a full FIFO. A Push<sub>c</sub> can only commit when  $\text{occ\_c}(t) < B(c)$  and a Pop<sub>c</sub> can only commit when  $\text{occ\_c}(t) > 0$ , so the occupancy stays within  $0..B(c)$  by induction.

If  $B(c) = 0$  (rendezvous): a transfer can commit only when the complementary endpoint is enabled in the same cycle (Push<sub>c</sub> and Pop<sub>c</sub> commit together), so the cycle-aligned occupancy is always 0 and thus satisfies  $0 \leq \text{occ\_c}(t) \leq B(c)=0$  at every cycle.

#### Lemma J.2 (Per-Channel Push/Pop Precedence)

Fix a channel  $c$ . Let Push<sub>c</sub>[ $k$ ] denote the  $k$ -th committed Push on  $c$  in the post-HLS execution, and Pop<sub>c</sub>[ $k$ ] denote the  $k$ -th committed Pop on  $c$  ( $k \geq 1$ ). By A9 (Single-transfer-per-cycle endpoint constraint), at most one Push<sub>c</sub> and at most one Pop<sub>c</sub> can commit per cycle on a fixed channel, so these “ $k$ -th” events are unambiguous. Then:  $\text{clk\_post}(\text{Push\_c}[k]) \leq \text{clk\_post}(\text{Pop\_c}[k])$ .

*Proof of Lemma J.2:*

- For  $B(c)=0$  (rendezvous), Push<sub>c</sub>[ $k$ ] and Pop<sub>c</sub>[ $k$ ] commit simultaneously, so equality holds.

- For  $B(c) > 0$ , a Pop can only commit when the FIFO is non-empty. After  $i$  committed Pushes and  $j$  committed Pops, occupancy is  $i-j$ . For  $\text{Pop}_c[k]$  to commit, immediately before it commits we must have  $i-j > 0$ , hence  $i \geq j+1 = k$ . Therefore the  $k$ -th Push has already committed by that time, i.e.,  $\text{clk\_post}(\text{Push}_c[k]) \leq \text{clk\_post}(\text{Pop}_c[k])$ .  $\square$

We now prove each equivalence property:

E1 (Synchronization-order preservation):

Fix any process  $P$ . By Appendix I,  $\tau_{B,P} \approx \tau_{\text{post},P}$ , so the synchronization events of  $P$  occur in the same source order in both traces. Since this holds for every  $P$ , E1 holds for the system traces  $\tau_{B,\text{system}}$  and  $\tau_{\text{post},\text{system}}$ .

E2 (Signal-visibility preservation):

Fix any process  $P$ . By Appendix I (and rule R2 as used there), every signal Read in  $P$  is anchored to  $P$ 's closest preceding synchronization event, and every signal Write in  $P$  is anchored to  $P$ 's closest succeeding synchronization event, in both  $\tau_{B,P}$  and  $\tau_{\text{post},P}$ . Since this holds for every  $P$ , E2 holds for  $\tau_{B,\text{system}}$  and  $\tau_{\text{post},\text{system}}$ .

E3 (Safe message issue ordering):

Fix any process  $P$ . We must show that  $P$ 's post-HLS execution satisfies E3 as stated in Appendix G: (i) for same-interface pairs  $q1, q2 \in Q$  with  $q1 <_{\text{src}} q2$ , we have  $\text{issue\_post}(q1) \leq \text{issue\_post}(q2)$ ; and (ii) for distinct interfaces that appear in sequence in the source ( $q1 <_{\text{src}} q2$ ), the operations are not issued in reverse order. This is exactly the per-process property established in Appendix I (Lemma I.0). Since it holds for every  $P$ , E3 holds for the system traces  $\tau_{B,\text{system}}$  and  $\tau_{\text{post},\text{system}}$ .

E4 (Per-channel FIFO semantics):

For each channel  $c$ , we must show:

1. The sequence of  $\text{Push}_c$  and  $\text{Pop}_c$  operations in  $\tau_{\text{post}}$  forms a legal FIFO schedule
2. No messages are dropped or duplicated
3. FIFO order is preserved in the  $\Sigma$ -trace sense: if a committed transfer  $m1$  precedes  $m2$  in the per-channel ( $\text{Push}_c/\text{Pop}_c$ ) history required by  $\text{Sys}_B/E4$ , then the corresponding committed transfers appear in the same order in  $\tau_{\text{post}}$  on that channel.

From Appendix I, each process preserves the order of its operations on each channel. Lemma J.1 establishes that  $0 \leq \text{occ\_post}(c,t) \leq B(c)$  at all times, and Lemma J.2 establishes that, for each channel  $c$ , the  $k$ -th Pop cannot commit before the  $k$ -th Push. Together these imply:

- Pops never underflow and Pushes never overflow the bounded FIFO (legality).
- Each committed Pop consumes exactly one previously committed Push, in FIFO order (no drops/duplication; FIFO order preserved).
- Capacity constraints are respected at every clock cycle.

Therefore, the post-HLS ( $\text{Push}_c$ ,  $\text{Pop}_c$ ) history on each channel is a legal bounded-FIFO execution consistent with the source-level FIFO semantics required by E4.

E5 (Messages cannot cross syncs):

For every synchronization call  $s$  (in the source order used by R3 /  $\text{pref}_S$  /  $\text{suff}_S$ ):

$$\forall q \in \text{pref}_S(s) : \text{issue\_post}(q) \leq \text{clk\_post}(s)$$

$$\forall q \in \text{suff}_S(s) : \text{issue\_post}(q) > \text{clk\_post}(s)$$

(Note: For blocking message transfers, the "no-withdraw"/persistence semantics imply that if a blocking operation is issued in the pre-side interval before  $s$ , then it must also commit no later than  $s$  (and similarly on the post side), because the process cannot complete  $s$  until all earlier-issued blocking requests in that interval have committed.)

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 the matching construction can always advance (i.e., executions do not diverge via permanent stalling on enabled actions).

Therefore,  $\tau_{B,system} \approx \tau_{post,system}$ .  $\square$

Remark 1. When all  $B(c)=0$ ,  $Sys\_B$  coincides with the rendezvous interpretation of  $Sys$ , so the theorem specializes to the rendezvous case.

Remark 2 (Back-annotation and abstract channel boundaries).

The Matchlib library provides a capacity back-annotation feature that extracts finite capacities  $B(c)$  from the RTL realization of each message-passing channel and makes them available to the source-level model. In this document,  $Sys\_B$  is the source-level system consisting of the user-written processes of  $Sys$ , interpreted under E4 as abstract bounded-FIFO channels of capacity  $B(c)$ . (When  $B(c)=0$  this coincides with rendezvous.) The formal results of Appendices J–K are stated for this abstract  $Sys\_B$  and depend only on the capacities  $B(c)$  and the E4 channel semantics at the chosen abstract channel boundaries.

For a simple pipelined loop with a single message-passing input and output, the RTL pipeline’s internal staging capacity is accounted for entirely by the back-annotated effective capacity  $B(c)$  at the chosen  $Sys\_B$  channel boundary. Concretely,  $Sys\_B$  still has exactly one  $\Sigma$ -visible  $Push\_c(v)$  at the producer endpoint and one  $\Sigma$ -visible  $Pop\_c(v)$  at the consumer endpoint. With overlapped iterations, the post-HLS RTL may accept/read-ahead up to  $B(c)$  values into internal pipeline staging before older iterations produce their corresponding outputs; these accept/read-ahead transfers are modeled as internal microstate ( $\epsilon$ ) within the concrete realization contributing to  $B(c)$ . The RTL may realize this staging on either side of a particular module-local “Pop interface” handshake point; that handshake point need not coincide with the chosen  $\Sigma$ -visible Pop endpoint. By construction, the chosen  $Sys\_B$  boundary encloses all staging counted in  $B(c)$  between the  $\Sigma$ -visible endpoints, so movements within that enclosed realization (including stage-to-stage handoff) do not create additional  $\Sigma$ -visible committed Push/Pop events beyond the single boundary Push/Pop events.

In more complex cases such as HW pipelines with multiple message-passing inputs, the back-annotation mechanism may elaborate the  $Sys\_B$  (source-level) simulation by introducing additional internal realization structure. In this document, we adopt the following simplified modeling discipline:

1. Terminology (elaboration and channel accounting). In an elaborated realization, the quantities  $B(\cdot)$ ,  $occ\_(\cdot)(t)$ , and the E4 availability predicate are interpreted over the explicit abstract channels present in the elaborated model (including any introduced staged/bundle channels). When we continue to write a source-level channel name  $c$  at the  $\Sigma$  boundary, we mean the chosen  $\Sigma$ -visible abstract boundary channel associated with  $c$  in that elaboration; downstream staging that is reachable only through a join (and therefore may be withheld when “join not met”) is represented by distinct internal channels with their own  $B(\cdot)$  and  $occ\_(\cdot)(t)$ , rather than as independent “slack” for  $c$  in isolation.
2. All buffering / capacity effects are represented explicitly as finite-capacity message channels (annotated with bounded capacity) that may appear upstream and/or downstream of any internal glue logic; and
3. Any internal “coupler” used to enforce mutual-stall between multiple inputs is purely combinational (stateless) handshake logic.

Concretely, a combinational coupler is a stateless component that (a) reads the upstream channels’ valid and data signals and computes/drives the downstream channels’ valid and data signals, and (b) reads

downstream ready signals and upstream valid signals and then computes/drives upstream ready signals. The coupler contains no internal state and has no separate credit/capacity state; any shared pipeline capacity, per-input staging, skid/elastic buffering, or credit-like effects are modeled only by the explicit finite-capacity channels inserted by the back-annotation realization.

Example (two inputs, staged Pops,  $II=1$ , latency=4):

Consider a 4-stage RTL pipeline that (i) performs Pop\_c1 in stage 1, (ii) performs Pop\_c2 in stage 2, and (iii) performs Push\_cout in stage 4, with  $II=1$ . To model the pipeline's internal staging and the stage-2 *input coupling* in the source-level Sys\_B simulation while keeping the same  $\Sigma$  boundary, the back-annotation elaboration may introduce the following internal realization structure:

- An explicit bounded channel F1 of capacity 1 on the c1 path to represent stage-1 in-flight storage of values that have been accepted from c1 but have not yet reached the stage-2 join point.
- A purely combinational coupler/join at the stage-2 boundary that enforces mutual stall: it allows a transfer to proceed only when (a) a token is available from F1, (b) a token is available from c2, and (c) downstream staging has space; it then presents the paired value(s) to the downstream staging.
- An explicit bounded channel F34 of capacity 2 after the coupler to represent the in-flight capacity of stages 3–4 for the joined/paired work.

This example illustrates why multi-input pipelines cannot always be captured by a single per-channel capacity number in isolation: c1 and c2 do not have independent “slack” because acceptance across the stage-2 join boundary (the paired transfer that consumes a token from c2 together with the staged token from F1) is coupled to availability of the corresponding stage-1 value from c1.

All buffering/capacity resides in the explicit bounded channels (F1, F34, and any additional bounded stages the realization uses); the coupler itself remains stateless combinational handshake logic.

This elaboration is an implementation/refinement of the abstract bounded-FIFO channels at the chosen observable boundary; it does not change the formal interface ( $\Sigma$ ) or E4's per-channel FIFO meaning at that boundary. Concretely, any back-annotation elaboration (including extra FIFO stages and/or combinational couplers) must satisfy:

- No new DUT-boundary observables: It introduces no new  $\Sigma$ -observable actions at the chosen DUT boundary, and it does not change the labeling of existing  $\Sigma$ -events (Push\_c(v), Pop\_c(v), etc.).
- Boundary/E4 preservation: At the chosen observable boundary, each  $\Sigma$ -visible committed Push\_c/Pop\_c event still denotes a committed transfer on the same abstract channel c, with per-channel FIFO order and the same E4 capacity/enableness interpretation under the annotated B(c) and occ\_c(t); any additional internal channels introduced by the elaboration (e.g., staged/bundle channels) are separate abstract channels in the elaborated model with their own B(·), occ\_(t), and E4 availability predicates.
- No double-counting of internal staging: internal transfers within the concrete realization that contributes to the effective capacity (extra FIFO stages, skid/elastic buffering, FIFO→pipeline-stage handoffs, movements across internal staged channels, and value propagation through combinational couplers) are  $\epsilon$ -steps and do not create additional  $\Sigma$ -visible committed Push\_c/Pop\_c events beyond the single boundary Push/Pop events.
- Multi-input pipelines / combinational couplers (mutual stall + explicit capacity):

To preserve per-channel E4 at the  $\Sigma$  boundary, any downstream staging that is reachable only through a join (and therefore may be withheld when “join not met”) shall be modeled as distinct internal finite-capacity channels with their own B(·) and occ\_(t), rather than being treated as additional independent “slack” for an upstream  $\Sigma$ -visible channel in isolation; the back-annotation elaboration may use

combinational couplers to enforce the intended mutual-stall behavior when some required inputs are unavailable. Any “shared capacity” or “distributed staging” effects needed for such a pipeline shall be represented only by explicit finite-capacity channels in the realization (e.g., bounded staging FIFOs, bounded slot/credit channels, bounded bundle channels), not by coupler state. The coupler itself remains stateless combinational handshake logic and does not redefine the per-channel E4 capacity/availability predicates at the chosen observable boundary. Explicit join boundary requirement (normative): The mutual-stall point shall be an explicit staged/bundle/join boundary inside the elaborated realization (i.e., the endpoint of an explicit finite-capacity message channel). “Join not met” shall therefore be expressed as ordinary ChannelDisabled at that explicit internal boundary (because that boundary’s own E4 availability predicate fails in the  $\epsilon$ -quiescent handshake fixed point). The realization must not force a  $\Sigma$ -visible operation on an abstract channel  $c$  at the chosen boundary to be ChannelDisabled solely due to the state of some other channel; any cross-channel dependency shall be expressed only via the explicit staged/bundle channels introduced by the elaboration. Operationally, the coupler is “invisible” at the  $\Sigma$  boundary: it only propagates ready/valid between the explicit staged/bundle channels inside the elaborated realization, and it does not change  $occ_c(t)$  (which is defined solely by  $\Sigma$ -visible boundary commits).

- WFG compatibility (Appendix K): The realization must not introduce a new blocking point that is invisible at the abstract boundary. In particular, if the consuming process cannot advance the current most-upstream pending work (as defined under “automatic flush”), then at least one blocking frontier operation (Push/Pop) at the chosen boundary is asserted with a true boundary request and is channel-disabled by the K.1 enablement test; the process is therefore “blocked on message passing” exactly as defined in K.1. Because couplers are purely combinational, they have no internal state transitions; any changes in readiness/validity induced by the coupler are purely a function of (i) downstream readiness and (ii) the same channel occupancies /  $\Sigma$ -visible committed transfers already accounted for by E4/K.1, with “downstream readiness” interpreted in the  $\epsilon$ -quiescent ( $\epsilon$ -normalized) state used for WFG evaluation.

Internal buffer movements across inserted staged channels are  $\epsilon$ -steps and are assumed WFG-inert under the same “most-upstream-work” discipline used elsewhere in Appendix K.

- Combinational well-formedness: The network of combinational couplers and ready/valid connections shall be combinational acyclic (no pure combinational ready/valid loops). Any required feedback that would otherwise create a combinational loop must pass through an explicit finite-capacity channel stage (and is therefore represented in the back-annotated realization and its effective capacities).

Accordingly, the formal development need not depend on the particular back-annotation construction, provided it respects E4 at the chosen observable boundary and satisfies the constraints above.

Remark 3 (Practical rendezvous liveness screening under determinism).

When B1 holds and the design’s control flow does not branch on empty/full observations of non-blocking interfaces (i.e., it depends only on the ordered history of observable Push/Pop/synchronization actions, not on “probe” outcomes (NB-CF, Appendix I.2)), the rendezvous specialization  $B(c)=0$  is often an effective practical screen for design-level deadlocks: it exercises the most constrained communication discipline and can expose true cyclic data-dependency deadlocks early. Because rendezvous is strictly more constrained than buffered operation, it can also yield false positives: a deadlock observed under  $B(c)=0$  may disappear once finite buffering is present. However, bounded buffering can still introduce capacity-induced deadlocks (FIFO-depth artifacts) when FIFO depths are underestimated; these deadlocks are real for that modeled capacity configuration but may disappear once capacities are increased to the intended design point. Increasing buffer capacity (or applying dynamic buffer-growth strategies) is a standard way to distinguish depth-limited deadlocks from true

cyclic dependency deadlocks. Accordingly, the formal results of Appendices J–K deliberately target Sys\_B with the actual back-annotated capacities  $B(c)$ , rather than relying on rendezvous alone.

Remark 4 (When Sys may be used instead of Sys\_B for safety-only checks).

If verification is concerned only with safety properties over the chosen DUT-boundary projection  $\Sigma$  (denote it  $\Sigma_{\text{DUT}}$ ) (and not with deadlock/liveness), Sys\_B remains the default reference model because  $\Sigma_{\text{DUT}}$  includes committed channel-transfer events, and bounded buffering ( $B(c) > 0$ ) generally changes the set/timing of those committed Push\_c/Pop\_c events relative to the rendezvous ( $B(c)=0$ ) case.

Practical note (early bring-up): Before accurate capacities  $B(c)$  are available (or before back-annotation can be used), running the rendezvous specialization Sys ( $B(c)=0$ ) against the intended testbench is still useful to validate functional intent and catch gross protocol/ordering bugs early; however, it remains a screening check, not a proof about the buffered RTL.

Important (soundness): When any channel that can influence  $\Sigma_{\text{DUT}}$ -observable behavior has  $B(c) > 0$ , the rendezvous model Sys is often a restriction of Sys\_B / RTL\_B at  $\Sigma_{\text{DUT}}$  (it can admit fewer  $\Sigma_{\text{DUT}}$  behaviors). Therefore, using Sys in place of Sys\_B is generally suitable only as a debug/screening check: a failure can be useful diagnostically, but a pass does not by itself prove  $\Sigma_{\text{DUT}}$ -safety of the buffered implementation.

Sys may be used as a proof-equivalent substitute for Sys\_B only when Sys and Sys\_B are  $\Sigma_{\text{DUT}}$ -equivalent for the chosen  $\Sigma_{\text{DUT}}$  (i.e., their  $\Sigma_{\text{DUT}}$ -projected trace sets coincide under the intended stimulus). A simple sufficient condition is that every  $\Sigma_{\text{DUT}}$ -visible channel has effective capacity zero at that boundary and that buffering elsewhere is  $\Sigma_{\text{DUT}}$ -opaque (it cannot enable/disable/reorder  $\Sigma_{\text{DUT}}$ -events or change when  $\Sigma_{\text{DUT}}$ -events occur). In all other cases—i.e., if any  $\Sigma_{\text{DUT}}$ -visible channel has  $B(c) > 0$ , or if buffered internal channels can affect when  $\Sigma_{\text{DUT}}$ -events occur—use Sys\_B with capacities matching RTL\_B (Remark 2).

Formal soundness condition. Let  $\text{Traces}_{\Sigma_{\text{DUT}}}(M)$  denote the set of  $\Sigma_{\text{DUT}}$ -projected traces of model  $M$  under the intended fixed stimulus. Sys may replace Sys\_B for proving a universal  $\Sigma_{\text{DUT}}$ -safety property  $\phi$  only if  $\text{Traces}_{\Sigma_{\text{DUT}}}(\text{Sys}) = \text{Traces}_{\Sigma_{\text{DUT}}}(\text{Sys}_B)$ ; under that condition,  $\text{Sys} \models \phi$  iff  $\text{Sys}_B \models \phi$  (and otherwise Sys is only a diagnostic/screening check as stated above).

Corollary J.1 (Execution-Level / Trace-Set Form of Theorem J.1)

Purpose This corollary makes explicit the quantifiers implicit in Theorem J.1: Appendix J is intended to relate sets of executions (under a fixed stimulus), not merely to relate a single pre-chosen pre-HLS trace to a single pre-chosen post-HLS trace.

Statement

Fix an initial state and a fixed external stimulus/environment behavior (e.g., the same testbench-driven inputs) for both Sys\_B and RTL\_B, where Sys\_B is the source-level system consisting of the user-written processes of Sys, interpreted under the channel semantics of E4 as abstract bounded FIFOs of capacity  $B(c)$  matching RTL\_B (see Remark 2 for the abstraction boundary; any back-annotation elaboration is an internal implementation detail). (When  $B(c)=0$  this coincides with rendezvous.)

Clarification (reactive environments). If the “stimulus” is produced by a reactive testbench/environment, interpret “fixed stimulus” as fixing the environment’s behavior as a  $\Sigma$ -causal function of the observed  $\Sigma$ -history (equivalently: include the environment as part of the closed system being executed, with its own fixed initial state and fixed nondeterministic choices). Under this interpretation, whenever two runs have matched  $\Sigma$ -prefixes, they are subject to the same subsequent environment behavior.

Let  $\text{Exec}_{\text{post}}$  denote the set of finite execution prefixes of RTL\_B that are reachable under this stimulus and that are prefixes of at least one (infinite) execution under the same stimulus that satisfies the Appendix J fairness/progress premises; and let  $\text{Exec}_B$  denote the corresponding set of finite execution prefixes of Sys\_B (defined analogously).

Under the assumptions of Theorem J.1 (per-process equivalence from Appendix I, including NB-CF for  $\epsilon$ -modeled non-blocking polls (I.2), or else using the RTL-consistent witness-selected non-blocking variants stated there, finite capacities  $B(c)$ , channel progress invariants  $P_{\text{push}} / P_{\text{pop}}$  (B3), and weak fairness (B2)), plus B1/B4/B5 when  $\epsilon$ -normalization is needed, the following prefix-matching property holds:

- (Post  $\rightarrow$  B existence) For every  $\tau_{\text{post}} \in \text{Exec\_post}$ , there exists  $\tau_B \in \text{Exec}_B$  such that  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$  under E1–E5.
- (B  $\rightarrow$  Post existence, RTL-consistent prefixes only) For every  $\tau_B \in \text{Exec}_B$  whose  $\Sigma$ -projection matches a prefix of the  $\text{RTL}_B$   $\Sigma$ -trace under the same fixed stimulus (in particular, any  $\tau_B$  produced by applying Appendix L's snooping wrapper when B1 holds for  $\text{RTL}_B$ ), there exists  $\tau_{\text{post}} \in \text{Exec\_post}$  such that  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$ .

Moreover, when B1 holds (deterministic  $\Sigma$ -projected trace under fixed stimulus), the matching  $\Sigma$ -label sequence is unique (up to insertion/removal of finite  $\epsilon$ -steps permitted by B4/B5). The corresponding execution prefix witness  $\tau_B$  need not be unique as a stateful prefix, since  $\text{Sys}_B$  may admit multiple  $\epsilon$ -different realizations with the same  $\Sigma$ -projection.  $\text{Sys}_B$  may also admit executions whose  $\Sigma$ -projection does not match  $\text{RTL}_B$ 's  $\Sigma$ -trace when latency-sensitive / non-blocking effects are  $\Sigma$ -visible; such executions are outside the scope of (B  $\rightarrow$  Post) unless constrained by the snooping/witness-selection technique (Appendix L).

Proof (sketch) Theorem J.1 establishes that the system-level E1–E5 invariants are preserved when composing processes that individually satisfy Appendix I and when channels satisfy the stated progress/fairness assumptions. Using those invariants, construct a witness prefix by induction on the length of the chosen prefix:

1. Given a reachable  $\tau_{\text{post}}$ , repeatedly extend a candidate  $\tau_B$  so that each next observable event in  $\tau_{\text{post}}$  is matched by the corresponding observable event in  $\tau_B$ , possibly preceded by a finite  $\epsilon^*$  chain to account for micro-steps/stuttering.
2. Finiteness of each required  $\epsilon^*$  extension follows from bounded stutter (B4) plus the progress/fairness premises that rule out permanent divergence via endlessly enabled-but-never-taken steps.
3. Each inductive extension preserves E1–E5 by the same reasoning used in Appendix I's finite-prefix construction, now lifted to the system via Theorem J.1's channel and composition invariants.

This yields the required existence of a matching prefix for an arbitrary reachable prefix of  $\text{RTL}_B$  (Post  $\rightarrow$  B). Conversely, (B  $\rightarrow$  Post) holds for those  $\text{Sys}_B$  prefixes that are RTL-consistent as stated above (e.g., prefixes along a snooped witness run when needed). Thus the corollary provides an execution-level correspondence without requiring that every  $\text{Sys}_B$  prefix correspond to some  $\text{RTL}_B$  prefix.  $\square$

#### Corollary J.2 (State Correspondence at the End of a Matched Observable Prefix)

Purpose This corollary serves as the bridge between trace equivalence and state correspondence needed in Appendix K. The mapping target is the source-level bounded-FIFO semantics of the design (capacity  $B(c)$ ), not an unrelated rendezvous ( $B(c)=0$ ) execution state.

Statement

Assume the hypotheses of Corollary J.1 (equivalently: the system-level assumptions of Theorem J.1 for  $\text{Sys}_B$  vs  $\text{RTL}_B$  under the fixed stimulus), and additionally assume B1 holds for  $\text{RTL}_B$  (unique  $\Sigma$ -projected trace under the fixed stimulus), plus B4 and B5 for  $\epsilon$ -normalization. The  $\Sigma$ -label sequence is therefore unique (up to insertion/removal of finite  $\epsilon$ -steps), although the  $\text{Sys}_B$  witness prefix/state need not be unique as a full micro-state.

Let  $\tau_{\text{post}}$  be any  $\tau_{\text{post}} \in \text{Exec\_post}$  (as defined in Corollary J.1).

Define its  $\epsilon$ -normalization  $\bar{\tau}_{\text{post}}$  as  $\tau_{\text{post}}$  extended by a (possibly empty) finite suffix  $\epsilon^*$  to a terminal state  $\bar{\sigma}_{\text{post}}$  that is  $\epsilon$ -quiescent, meaning: no further  $\epsilon$ -step is enabled from  $\bar{\sigma}_{\text{post}}$  (so any further

progress, once enabled, must proceed via an observable  $\Sigma$ -action rather than additional internal  $\epsilon$ -steps). Take  $\epsilon^*$  to be a maximal finite  $\epsilon$ -extension of  $\tau_{\text{post}}$ ; such a maximal finite extension exists under B4 (finite stutter bound), with B5 (quiescence closure) ensuring we can treat the resulting  $\epsilon$ -quiescent representative as the canonical endpoint for the observable prefix.

(In clauses (1)–(2) directly below, interpret  $\sigma_{\text{post}}$  as  $\bar{\sigma}_{\text{post}}$ , i.e., the  $\epsilon$ -normalized terminal state. When referring below to “source-level variables,” the “next observable frontier,” and the “logical FIFO state” in  $\sigma_{\text{post}}$ , interpret those notions via the standard source-level projection/abstraction from RTL\_B microstate to the corresponding source-level state as used in Appendix I; micro-architectural registers and bookkeeping are ignored by this projection.)

Let Sys\_B denote the source-level system consisting of the user-written processes of Sys, interpreted under the channel semantics of E4 as abstract bounded FIFOs of capacity B(c) matching RTL\_B (see Remark 2 for the abstraction boundary; any back-annotation elaboration is an internal implementation detail). (When B(c)=0 this coincides with rendezvous.) Then there exists a finite execution prefix  $\tau_B$  of Sys\_B ending in some reachable state  $\sigma_B$  such that:

(1) The observable prefixes match:  $\tau_B, \text{system} \approx \tau_{\text{post}}, \text{system}$  (equivalence under E1–E5).

(2) The terminal states correspond at the source level:  $\sigma_B \equiv \sigma_{\text{post}}$ , where “ $\equiv$ ” means:

(Note: “ $\equiv$ ” is intentionally a frontier/enable/value correspondence (clauses (a)(b)(c)), not full micro-state equality; internal pipeline/buffering microstate may differ between  $\sigma_B$  and  $\sigma_{\text{post}}$  provided such differences are  $\epsilon$ -steps that are WFG-inert as assumed in Appendix K.)

(a) For every process P, the next observable source-level frontier with respect to P’s source partial order (i.e., the per-process ordering relation used by R1–R4 / referenced by E3 and not necessarily the single next syntactic statement of a sequential thread) is the same in  $\sigma_B$  and  $\sigma_{\text{post}}$ . Here the “frontier” means the set of minimal (w.r.t. P’s source partial order) observable items that may occur next.

Concretely, this means either:

- a (possibly multi-element) set of pending message actions (Push\_c and/or Pop\_c) on distinct interfaces that are currently minimal candidates and may be issued and (if channel-enabled) committed in any order or together in one cycle, or
- an explicit synchronization call s, together with the same set of signal Read/Write actions that are logically anchored at s per E2.
- This frontier equality does not require the pre-HLS and post-HLS models to commit identical subsets of that message set in the same cycle; it only identifies the set of currently-minimal candidates.

(b) For every process P, and for every item x in P’s next observable frontier:

- (i) The evaluation of x’s boundary request (and any source-level control predicate used to decide that x is the next observable action) is the same in  $\sigma_B$  and  $\sigma_{\text{post}}$ ; equivalently,  $\sigma_B$  and  $\sigma_{\text{post}}$  agree on all source-level state variables read when evaluating that guard/predicate. (Channel-side enablement of Push/Pop is handled separately by clause (c).)
- (ii) If x produces a value—i.e., x is a Push\_c, or x is a signal Write action (or set of such actions) anchored at the next synchronization call—then the value produced by x (payload/value expression) is the same in  $\sigma_B$  and  $\sigma_{\text{post}}$  (hence  $\sigma_B$  and  $\sigma_{\text{post}}$  agree on all source-level variables read by that expression). If x observes a value via signal Read actions anchored at the next synchronization call, those observed signal values are the same in  $\sigma_B$  and  $\sigma_{\text{post}}$  because the stimulus is the same and E2 anchors the reads to that same synchronization call.
- (iii) If x is a Pop\_c, then the value returned by that Pop\_c is determined by the channel’s logical FIFO contents after the matched observable prefix; by E4 and the matched Push/Pop history, that logical head element is the same for  $\sigma_B$  and  $\sigma_{\text{post}}$ .

(c) For every channel  $c$ , the abstract FIFO state in  $\sigma_B$  (empty/full predicate, and the logical head element when non-empty) agrees with the logical FIFO state induced by the matched Push<sub>c</sub>/Pop<sub>c</sub> history of (1). Physical micro-architectural realization—pipeline registers, arbitration bookkeeping, and the implementation of buffering—is abstracted away by “ $\equiv$ ”, but the logical occupancy/ordering facts induced by the matched trace are not.

In particular, internal transfers within the concrete realization (e.g., movement between pipeline/buffering stages) that do not commit a boundary transfer are treated as  $\epsilon$  and do not change this abstract FIFO state. The logical head/occupancy at the chosen  $\Sigma$  boundary advances only on  $\Sigma$ -visible boundary commits (Push<sub>c</sub>/Pop<sub>c</sub>) that appear in the matched Push/Pop history; bounded-FIFO legality is checked solely at this boundary using the annotated B(c).

Proof

Step 0 (By definition, work with  $\epsilon$ -normalized terminal states) By the  $\epsilon$ -normalization defined in the Statement, w.l.o.g. assume  $\tau_{\text{post}}$  ends in an  $\epsilon$ -quiescent state (no  $\epsilon$ -step is enabled); we write this terminal state as  $\sigma_{\text{post}}$ . (If not, replace  $\tau_{\text{post}}$  by its maximal finite  $\epsilon$ -extension  $\bar{\tau}_{\text{post}} = \tau_{\text{post}} \circ \epsilon^*$ , which preserves the observable prefix and is finite by B4, with the endpoint treated as the canonical  $\epsilon$ -quiescent representative per B5.)

Step 1 (Get a matching source-level observable prefix) Fix the external test stimulus that produced the given post-HLS prefix  $\tau_{\text{post}}$ . By the system-level prefix-matching property (Corollary J.1) applied to Sys<sub>B</sub> and RTL<sub>B</sub>, there exists a finite execution prefix  $\tau_B$  of Sys<sub>B</sub> under that same stimulus such that  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$  under E1–E5. Let  $\sigma_B$  be the terminal state of this witness prefix  $\tau_B$ .

Step 2 (Align per-process control points) Because  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$  and E1/E2/E3/E4/E5 preserve per-process same-interface order, no-reverse constraints, side-of-sync constraints, and signal anchoring at synchronization points, each process  $P$  has executed the same per-endpoint observable history in both prefixes (up to commutation/regrouping of independent distinct-interface message actions). Therefore, at the end of the prefixes,  $P$  has the same next observable frontier in both  $\sigma_B$  and  $\sigma_{\text{post}}$  (clause (2a)), including the same logically anchored signal Read/Write actions at the next synchronization call when applicable.

Step 3 (Align the relevant source-level state) Matched prefixes preserve the source-level state needed for next-action enablement and payloads exactly as in Appendix I, giving clause (2b). Clause (2c) holds because, under E4, the abstract FIFO empty/full status and logical head element are uniquely determined by the matched Push/Pop history of (1), and thus coincide in  $\sigma_B$  and  $\sigma_{\text{post}}$ . Combining Steps 1–3 gives a terminal source-level state  $\sigma_B$  with  $\sigma_B \equiv \sigma_{\text{post}}$ .  $\square$

### J.3 Causal Dependency (“Happens-Before”) and What System-Level Equivalence Guarantees (Informative)

The system-level result of Appendix J establishes equivalence of the *aggregate* observable behavior, i.e.  $\tau_{B,\text{system}} \approx \tau_{\text{post},\text{system}}$  over the alphabet  $\Sigma$  of channel operations, synchronization events, and signal Read/Write actions.

This equivalence is intentionally *observational*: it constrains what an external environment can distinguish at  $\Sigma$ , not internal RTL micro-timing.

As a consequence, Appendix J should be read as preserving functionally significant ordering, not incidental latency. Functionally significant ordering is exactly the ordering induced by the happens-before relation ( $\rightarrow$ ) defined in the “Causal Dependency Between Processes” section: a designer must express any required cross-process ordering using message-passing edges, explicit synchronization (e.g., barriers), or explicit signal handshakes; designs “shall not rely” on the relative order of causally independent events.

Accordingly, the system-level theorem implies the following *causality preservation* principle:

- If two synchronization anchors  $s1$  and  $s2$  are causally related ( $s1 \rightarrow s2$ ), then the post-HLS execution preserves their order. This is because each elementary happens-before link is preserved: intra-process order is preserved by the per-process result; message-passing links are preserved by Lemma J.2 ( $\text{clk\_post}(\text{Push\_c}) \leq \text{clk\_post}(\text{Pop\_c})$ ); and explicit synchronization/handshake links propagate only at clock edges under the same signal and synchronization semantics. By transitivity, the entire causal chain preserves order.
- If  $s1$  and  $s2$  are causally independent (neither  $s1 \rightarrow s2$  nor  $s2 \rightarrow s1$ ), then their relative order is *not* constrained by the methodology, and differences are treated as incidental latency variation; relying on such ordering is outside the formal guarantees.

This perspective is what makes the “single testbench / no surprises” implication precise: any testbench that observes only  $\Sigma$  and encodes its expectations through causal dependencies (channels, barriers, or explicit handshakes) cannot distinguish Sys\_B (the pre-HLS model interpreted with bounded-FIFO capacities  $B(c)$  matching RTL\_B) from the post-HLS RTL. (The rendezvous specialization Sys with  $B(c)=0$  is covered only under the explicit conditions of Remark 4.)

---

## Appendix K – Liveness Preservation for Buffered Implementations

### K.1 Scope, Notation, and Definition of Internal Message-Passing Deadlock

Terminology alignment. Throughout Appendix K, we use the Common Formal Definitions predicates: ‘boundary request is asserted’ means  $\text{BoundaryReq}(m,t)$ ; ‘channel-enabled’ means  $\text{ChannelEnabled}(m,t)$ ; and ‘channel-disabled’ means  $\text{ChannelDisabled}(m,t)$  (all evaluated at the  $\epsilon$ -quiescent ready/valid fixed point for cycle  $t$ ).

We work with a fixed design:

- System Sys\_B: The collection of pre-HLS processes obeying the R-rules and B-rules, interpreted under the source-level channel semantics of E4 as abstract bounded FIFOs. Each channel  $c$  has capacity  $B(c) \geq 0$  (the same  $B(c)$  as in the RTL implementation, see Remark 2 in Appendix J).
- Post-HLS implementation RTL\_B: The hardware implementation of the same processes in which each channel  $c$  is realized with finite capacity  $B(c) \geq 0$ . In this appendix, we compare Sys\_B (source-level bounded-FIFO semantics) to RTL\_B (micro-architectural realization). The proof does not require mapping buffered RTL states to a distinct rendezvous ( $B(c)=0$ ) state.
- Witness-selection note (debug/simulation): Appendix K’s liveness argument uses Corollary J.2 only to assert existence of a corresponding Sys\_B state  $\sigma_B$  for a given  $\epsilon$ -normalized RTL state  $\sigma_{\text{post}}$  (via the source-level correspondence  $\equiv$ ). If, for practical debug, one wishes to construct a deterministic Sys\_B execution that tracks the unique RTL  $\Sigma$ -trace ( $B1$ ) so that  $\sigma_B/\text{WFG}_B$  are reproducible, then the Appendix L snooping wrapper may be applied to select a witness Sys\_B execution aligned to RTL latency-sensitive events.

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 abstract bounded-FIFO state at the Sys\_B channel boundary—its current occupancy  $\text{occ}_c(t)$  and (for buffered channels) its ordered contents. Here  $\text{occ}_c(t)$  counts all items that have been committed by  $\text{Push}_c$  but have not yet been committed by  $\text{Pop}_c$  at that abstract boundary. In RTL\_B, this count includes items residing anywhere in the concrete realization of channel  $c$  that contributes to the back-annotated capacity  $B(c)$  (e.g., FIFO storage, skid/elastic buffers, and any HLS-inserted pipeline staging that can hold in-flight messages). Thus Sys\_B treats that entire contributing realization as a single composite buffer at one Sys\_B



boundary; internal stage-to-stage movements inside that realization (including FIFO→pipeline-stage handoffs) are  $\epsilon$ -steps and do not create additional committed Push\_c/Pop\_c  $\Sigma$ -events beyond the boundary events counted by  $occ\_c$ .

3. Any additional architectural state not representing buffered messages on any channel (e.g., computation pipeline registers, arbitration state, bookkeeping, etc.).

We adopt the usual Wait-For Graph (WFG) abstraction, restricted to internal message-passing channels. Internal means: the producer and consumer of channel  $c$  are both processes of  $Sys\_B / RTL\_B$  (i.e., both endpoints are in the modeled system).

Channels whose complementary endpoint is the external environment/testbench (DUT boundary) are excluded from the WFG and from the deadlock definition in Appendix K.

- Vertices: Processes.
- Edges: There is a directed edge  $P \rightarrow Q$  via channel  $c$  in state  $\sigma$  if there exists some operation  $op \in \text{Front\_P}(\sigma)$  on channel  $c$  such that  $op$  is channel-disabled (i.e.,  $\text{ChannelDisabled}(op, t)$  holds), and  $Q$  is the unique process that is the complementary endpoint for  $op$  (producer/consumer partner for  $c$ ). This edges definition applies uniformly for both buffered channels ( $B(c) > 0$ , enable is space/data availability via  $occ\_c$ ) and rendezvous channels ( $B(c) = 0$ , enable requires the complementary endpoint's boundary request to be true in the same cycle), hence WFGs may contain a mix of edges via both kinds of channels. Here  $\text{Front\_P}(\sigma)$  is the set of minimal (w.r.t.  $P$ 's program-order / partial source order) pending blocking channel operations  $op \in \{\text{Push\_c}, \text{Pop\_c}\}$  (issued by  $P$  but not yet committed);  $\text{Front\_P}(\sigma)$  may contain multiple operations on distinct interfaces (reflecting the R/E freedom to issue/commit multiple Push/Pop in one cycle).  $P$  is "blocked on message passing" iff  $\text{Front\_P}(\sigma)$  is non-empty and every  $op' \in \text{Front\_P}(\sigma)$  is channel-disabled (its channel-side enabling condition is false). Thus WFG edges arise only from states where no enabled frontier operation exists; and when  $\text{Front\_P}(\sigma)$  has multiple members,  $P$  may have multiple outgoing WFG edges (one per blocked frontier  $op$ ). (No deassert-before-commit assumption.) Blocking Push\_c/Pop\_c requests are non-withdrawable: once the request corresponding to a frontier operation is asserted, it is not deasserted before that operation commits (and Push\_c payload remains stable while asserted). This rules out "try-and-withdraw" behavior for blocking transfers within the WFG abstraction. By Assumption A6 (point-to-point channels), the complementary endpoint  $Q$  for any blocked Push\_c/Pop\_c on  $c$  is unique.

Shared-resource note: If the RTL contains sharing that would otherwise make the complementary endpoint non-unique (e.g., multi-producer or multi-consumer access to a logical resource), that sharing is modeled as an explicit arbiter process connected to each client by point-to-point channels, so WFG edges remain well-defined.

All WFG reasoning is over  $\epsilon$ -quiescent ( $\epsilon$ -normalized) global states: internal pipeline/arbitration micro-steps are treated as  $\epsilon$  and are not represented as separate WFG blocking points. Here, " $\epsilon$ -quiescent" includes the fixed point of purely combinational ready/valid propagation (including through combinational couplers), so K.1 enablement is evaluated only on those settled ready/valid values.

(WFG-inert  $\epsilon$  premise.) We assume these  $\epsilon$ -steps are observationally silent with respect to the WFG abstraction: for any  $\sigma \xrightarrow{\epsilon^*} \sigma'$ , they do not change (i) the potentially-blocking frontier  $\text{Front\_P}(\sigma)$  for any process  $P$  (i.e., which guarded blocking Push/Pop operations are minimal and pending), nor (ii) the truth of any boundary requests relevant to those frontier operations, nor (iii) the channel-side enabling status (as defined in K.1) of any  $op \in \text{Front\_P}(\sigma)$ . Equivalently,  $\epsilon$ -steps do not change the channel state relevant to K.1 enabling (e.g.,  $occ\_c(t)$  for  $B(c) > 0$  buffered channels), and they do not create/remove rendezvous enablement for a frontier transfer except via  $\Sigma$ -visible committed Push\_c/Pop\_c events.

In particular (pipelined loops / overlapped iterations): when HLS introduces overlap that may initiate later-iteration message reads “early,” we assume the automatic flush discipline (e.g., flush-style control such as `hls_stall_mode flush`) guarantees WFG-inertness. Concretely, this discipline is drain-only: if a required blocking message-passing operation for the current most-upstream pending work (i.e., a minimal pending blocking frontier operation—minimal w.r.t. source program order among currently pending asserted blocking requests in the current Sync interval) is not channel-enabled—either a read (blocking Pop) is not available or a write (blocking Push) is back-pressured (e.g., FIFO full or rendezvous partner not ready)—the process blocks at that Pop/Push; it does not initiate any new message-passing Pops or Pushes for younger overlapped iterations while blocked, and it does not withdraw any already-asserted blocking Push/Pop request (no “try-and-withdraw” behavior). The implementation may continue to advance and complete already-started older work and may commit any resulting output Pushes so long as this progress occurs downstream of the blocked Pop/Push and does not advance any work past that blocked operation (and subject to normal channel enablement). Thus, “drain” here means draining the in-flight portion of the pipeline *beyond the blocked point*; if the blocked point is in a late stage, draining may stop immediately. Under this discipline, flush does not introduce a later-iteration blocking Push/Pop into  $\text{Front\_P}(\sigma)$ , so transient internal pipeline activity remains  $\epsilon$  and does not contribute wait-for edges. Overlapped execution in pipelined loops is modeled only as a potential retiming of ordinary  $\Sigma$ -visible boundary commits ( $\text{Push\_c}/\text{Pop\_c}$ ) relative to the unpipelined source, and is accounted for by the back-annotated bounded-FIFO semantics at the chosen  $\text{Sys\_B}$  boundary (E4). In particular, the channel facts relevant to K.1 enabling/WFG edges—e.g.,  $\text{occ\_c}(t)$  for  $B(c)>0$ , and rendezvous enablement for  $B(c)=0$ —change only on  $\Sigma$ -visible committed  $\text{Push\_c}/\text{Pop\_c}$  events at that boundary; internal pipeline movement/staging within the composite realization counted in  $B(c)$  is  $\epsilon$  and does not affect K.1 enablement or introduce/remove WFG edges.

If a design intentionally uses internal micro-timing to change such control decisions (e.g., a sequence of “internal” steps changes the frontier  $\text{Front\_P}(\sigma)$  of minimal guarded blocking ops, or changes guard truth for any frontier op), then that behavior must be treated as  $\Sigma$ -observable (e.g., via explicit modeling/snooping) and is outside the scope of Appendix K’s WFG-based deadlock preservation argument. A common instance is branching on the success/failure of a non-blocking poll in a way that changes the next  $\Sigma$ -visible frontier; in that case, either NB-CF must hold (Appendix I.2), or the poll outcomes must be witness-selected as stated there.

Non-blocking polling attempts and other purely internal actions do not contribute edges to the WFG; they are modeled as internal  $\epsilon$ -steps. When a non-blocking call succeeds, that success is already represented at the  $\Sigma$  level as the corresponding committed  $\text{Push\_c}/\text{Pop\_c}$  event, but it does not itself add a wait-for edge because it is not a blocking operation.

SyncChannels and barrier well-formedness.

SyncChannel (barrier) operations are treated as pure synchronization events and are omitted from the Wait-For Graph, which tracks only blocking Push/Pop dependencies. We therefore interpret “deadlock” in this appendix as internal message-passing deadlock (i.e., a deadlocked set in which all processes are blocked on some internal Push/Pop dependency captured by the WFG). Any global stall at a barrier caused by mismatched or conditional participation (e.g., a process can permanently bypass the barrier or reach it a different number of times) is a specification-level error already present in the pre-HLS model; such stalls are not created by the HLS transformation and are excluded by an explicit barrier well-formedness assumption (A8 below), not by the internal message-passing deadlock premise. Under this assumption, omitting SyncChannels from the WFG is sound: a post-HLS deadlock implies the existence of a Push/Pop wait-cycle as characterized below.

We say that a process  $P$  is blocked on message passing in state  $\sigma$  if:

- Let  $\text{Front}_P(\sigma)$  denote the set of minimal (w.r.t.  $P$ 's program-order / partial source order) pending blocking channel operations  $op \in \{\text{Push}_c, \text{Pop}_c\}$  (i.e., operations issued by  $P$  but not yet committed). We say that  $P$  is blocked on message passing in state  $\sigma$  if, for every  $op \in \text{Front}_P(\sigma)$ ,  $\text{BoundaryReq}(op, \sigma)$  holds and  $\text{ChannelDisabled}(op, \sigma)$  holds (equivalently,  $\text{BoundaryReq}(op, \sigma) \wedge \neg \text{ChannelEnabled}(op, \sigma)$ ).
- Remark (pipelined loops / overlapped iterations). Under the automatic flush (drain-only) discipline assumed above, while an older required blocking Pop/Push remains pending the implementation does not initiate any new blocking request assertions for younger overlapped iterations. Therefore, even in a pipelined loop, the “minimal (w.r.t. program order)” guarded blocking operations captured by  $\text{Front}_P(\sigma)$  coincide with the most-upstream pending blocking op(s) (i.e., the minimal pending blocking frontier) as defined under “Automatic flush (drain-only blocking discipline),” and pipelining does not introduce additional younger-iteration frontier operations beyond those already present in the unpipelined process.
- $\text{Front}_P(\sigma)$  is non-empty, and for every  $op \in \text{Front}_P(\sigma)$ , that operation cannot make progress because it is channel-disabled (i.e.,  $\text{ChannelDisabled}(op, \sigma)$  holds).
- Equivalently:  $P$  is not blocked on message passing if at least one operation in its frontier is channel-enabled (i.e.,  $\text{ChannelEnabled}(op, \sigma)$  holds).

An internal message-passing 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 message passing.
- 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 modeled system.

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

## K.2 Assumptions and Imported Results

K.2.1 Standing Assumptions We assume throughout Appendix K:

A1. R-rules and B-rules. The source-level model  $\text{Sys}_B$  and the post-HLS implementation  $\text{RTL}_B$  satisfy the process- and channel-level semantic rules defined earlier (R-rules and B-rules).

A2. Finite Pipeline Depth (FPD). There is a global bound on the number of purely internal steps ( $\epsilon$ -steps) that can occur between consecutive observable actions.

A3. Weak Fairness (WF) (B2). If an action remains continuously enabled (its guard remains true), the scheduler eventually selects it.

A4. Channel Progress (B3; Appendix N). For each channel  $c$  with capacity  $B(c)$ , assume the progress invariants of Appendix N hold. This is an obligation on the scheduler/environment (e.g., fairness of arbiters/back-pressure) and is not implied by the local R-rules alone:

- $P_{\text{push}}(c)$ : If a process  $P$  is stalled at a blocking  $\text{Push}_c$  with a persistent (non-withdrawable; no deassert-before-commit) request asserted, and the complementary endpoint process  $Q$  continuously has the matching  $\text{Pop}_c$  pending as a frontier operation (i.e.,  $\text{Pop}_c \in \text{Front}_Q(\sigma)$  continuously, with its boundary request (i.e.,  $\text{BoundaryReq}(\text{Pop}_c, \sigma)$ ) remaining true), then the transfer on  $c$  must eventually

complete (in particular: for  $B(c) > 0$ , some complementary Pop\_c eventually commits, freeing space; for  $B(c) = 0$ , the rendezvous completion eventually occurs).

- P\_pop(c): If a process P is stalled at a blocking Pop\_c with a persistent (non-withdrawable; no deassert-before-commit) request asserted, and the complementary endpoint process Q continuously has the matching Push\_c pending as a frontier operation (i.e.,  $\text{Push}_c \in \text{Front}_Q(\sigma)$  continuously, with its boundary request (i.e.,  $\text{BoundaryReq}(\text{Push}_c, t)$ ) remaining true), then the transfer on c must eventually complete (in particular: for  $B(c) > 0$ , some complementary Push\_c eventually commits, providing data; for  $B(c) = 0$ , the rendezvous completion eventually occurs).

A5. Source-Level Liveness Assumption. The source-level bounded-FIFO system Sys\_B (with capacities  $B(c)$  as defined in K.1) is deadlock-free under assumptions A1–A4.

A6. Point-to-point channels (single-producer/single-consumer). For each message-passing channel c (including buffered channels with  $B(c) > 0$  and rendezvous channels with  $B(c) = 0$ ), exactly one process performs all Push\_c operations on c (the producer) and exactly one process performs all Pop\_c operations on c (the consumer). Hence the complementary endpoint for any blocked Push\_c/Pop\_c is unique, and WFG edges are well-defined even for mixed WFGs that include both buffered and rendezvous channels.

If a design has a shared resource that would violate this uniqueness, it must be modeled via an explicit arbiter process plus point-to-point channels (as noted in K.1).

A7 (Determinism / witness selection). Assume B1 holds for RTL\_B (post-HLS), so the  $\Sigma$ -labeled execution under the given stimulus is unique up to finite  $\epsilon$ -stutter.

Do not require B1 for Sys\_B: there may be multiple Sys\_B executions that are  $\equiv$  to the same  $\tau_{\text{post}}$  at the  $\Sigma$ /frontier level. Appendix K uses only the existence of such a Sys\_B witness (Corollary J.2).

When a deterministic, reproducible Sys\_B witness is desired for simulation/debug, apply the Appendix L snooping wrapper to select a particular witness aligned with the latency-sensitive (arbiter-visible)  $\epsilon$ -choices. We also rely on B4 and B5 (Appendix N) to justify  $\epsilon$ -normalization /  $\epsilon$ -quiescent endpoints, as required by Corollary J.2 (state correspondence).

A8. Barrier well-formedness (SyncChannels). For each SyncChannel instance, the set of designated participant processes reaches the barrier the same number of times under the fixed stimulus/initial state; equivalently, no participant can permanently bypass a barrier call while another participant is waiting at that barrier. (Barrier stalls are therefore outside the behaviors considered in Appendix K's internal message-passing deadlock analysis.)

A9. Single-transfer-per-cycle endpoint constraint (scalar channel interface). For every message-passing channel c, in any clock cycle t, at most one Push\_c commit and at most one Pop\_c commit may occur on c. If a design uses multi-lane/burst transfers on a logical resource, it must be modeled explicitly (e.g., k parallel point-to-point channels or a widened message) so that the  $\Sigma$ -level per-channel event model satisfies this constraint.

### K.3 Lemma K.1 — Characterization of Internal Message-Passing Deadlock (Buffered and Rendezvous Cases)

Statement: Let  $\sigma_{\text{post}}$  be a reachable  $\epsilon$ -quiescent global state of the post-HLS system RTL\_B (equivalently, the  $\epsilon$ -normalization of some reachable state, which exists as a finite extension by B4/B5). Suppose there is a non-empty set of processes D that is internally message-passing deadlocked (as defined in K.1). Then:

Every process P in D is blocked on a blocking channel operation (Push or Pop), not on an internal step.

Buffered-channel case ( $B(c) > 0$ ):

- If P is blocked on Push\_c and  $B(c) > 0$ , then  $\text{occ}_c(t) = B(c)$  (channel is full).
- If P is blocked on Pop\_c and  $B(c) > 0$ , then  $\text{occ}_c(t) = 0$  (channel is empty).

Rendezvous case ( $B(c) = 0$ ):

- If  $P$  is blocked on  $\text{Push}_c$  and  $B(c) = 0$ , then the complementary endpoint is not simultaneously requesting the matching  $\text{Pop}_c$  in  $\sigma_{\text{post}}$  (i.e., its  $\text{Pop}_c$  boundary request is not true in that cycle, so rendezvous enabling for this transfer is false).

- If  $P$  is blocked on  $\text{Pop}_c$  and  $B(c) = 0$ , then the complementary endpoint is not simultaneously requesting the matching  $\text{Push}_c$  in  $\sigma_{\text{post}}$ .

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: By the definition of “blocked on message passing” in K.1, if  $P$  is blocked due to a frontier  $\text{Push}_c$  (i.e.,  $\text{Push}_c \in \text{Front}_P(\sigma_{\text{post}})$ ), then  $\text{Push}_c$ ’s boundary request is true and the channel-side enabling condition for  $\text{Push}_c$  is false.

- If  $B(c) > 0$ , the channel-side enabling condition for  $\text{Push}_c$  is space availability, i.e.,  $\text{occ}_c(t) < B(c)$ .

Hence, if  $P$  is blocked on  $\text{Push}_c$  in  $\sigma_{\text{post}}$ , necessarily  $\text{occ}_c(t) = B(c)$  (channel is full).

- If  $B(c) = 0$  (rendezvous), the channel-side enabling condition is “the complementary  $\text{Pop}_c$  endpoint’s boundary request is true in the same cycle” (K.1). Since  $P$  is blocked, this rendezvous enabling condition is false; equivalently, the complementary endpoint is not simultaneously requesting  $\text{Pop}_c$  in  $\sigma_{\text{post}}$ .

Blocked Pop: Symmetric. By the definition of “blocked on message passing,” if  $P$  is blocked due to a frontier  $\text{Pop}_c$  (i.e.,  $\text{Pop}_c \in \text{Front}_P(\sigma_{\text{post}})$ ), then  $\text{Pop}_c$ ’s boundary request is true and the channel-side enabling condition for  $\text{Pop}_c$  is false.

- If  $B(c) > 0$ , the channel-side enabling condition for  $\text{Pop}_c$  is data availability, i.e.,  $\text{occ}_c(t) > 0$ . Hence  $\text{occ}_c(t) = 0$  (channel is empty).

- If  $B(c) = 0$  (rendezvous), the enabling condition is “the complementary  $\text{Push}_c$  endpoint is enabled in the same cycle.” Since  $P$  is blocked, the complementary endpoint is not simultaneously enabled for  $\text{Push}_c$  in  $\sigma_{\text{post}}$ .

Closed Cycle: By definition of internal message-passing 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 RTL buffered system to those in the source-level bounded-FIFO semantics  $\text{Sys}_B$ .

Statement: For any pair of corresponding states  $(\sigma B, \sigma_{\text{post}})$  where  $\sigma B \equiv \sigma_{\text{post}}$  (Corollary J.2), every wait-for edge in  $\text{WFG}_{\text{post}}$  is also present in the source-level  $\text{WFG}_B$ . (Equivalently,  $\text{WFG}_{\text{post}} \subseteq \text{WFG}_B$ .) This holds for mixed WFGs containing edges via both buffered channels ( $B(c) > 0$ ) and rendezvous channels ( $B(c) = 0$ ).

Proof: Since  $\sigma B \equiv \sigma_{\text{post}}$ , every process  $P$  is at the same abstract program point and has the same relevant guard truth values for its current potentially-blocking frontier (i.e., the same set of minimal guarded blocking Push/Pop operations, and the same local enabling conditions for each). Moreover, for buffered channels with  $B(c) > 0$ , the channel empty/full predicate agrees between  $\sigma B$  and  $\sigma_{\text{post}}$  (Corollary J.2, clause (2c)). We show that each RTL edge transfers to the source-level WFG by case-splitting on  $B(c)$ .

Case 1: Buffered channel ( $B(c) > 0$ ).

Consumer (Blocked Pop): Suppose  $P$  waits for  $Q$  via  $\text{Pop}_c$  in the RTL system on a channel  $c$  with  $B(c) > 0$ . By Lemma K.1,  $\text{occ}_c(t) = 0$  in  $\sigma_{\text{post}}$  (empty). By  $\sigma B \equiv \sigma_{\text{post}}$ ,  $c$  is also empty in  $\sigma B$ . Under the bounded-FIFO channel semantics, completion of  $P$ ’s  $\text{Pop}_c$  requires a complementary  $\text{Push}_c$  by  $Q$ . Hence  $P \rightarrow Q$  is also an edge in  $\text{WFG}_B$ .

Producer (Blocked Push): Symmetric. If  $P$  waits for  $Q$  via  $\text{Push}_c$  on a channel  $c$  with  $B(c) > 0$ , Lemma K.1

gives  $\text{occ\_c}(t)=B(c)$  (full) in  $\sigma_{\text{post}}$ , hence also in  $\sigma_B$ . Completion of P's  $\text{Push\_c}$  requires a complementary  $\text{Pop\_c}$  by Q to make space. Hence  $P \rightarrow Q$  is also an edge in  $\text{WFG}_B$ .

Case 2: Rendezvous channel ( $B(c)=0$ ).

If P is blocked on  $\text{Push\_c}$  in  $\sigma_{\text{post}}$ , then the complementary endpoint is not simultaneously requesting  $\text{Pop\_c}$  (its  $\text{Pop\_c}$  boundary request is not true in that cycle); so the corresponding edge exists in  $\text{WFG}_B$ .

Therefore every RTL wait-for dependency is also a source-level wait-for dependency, so  $\text{WFG}_{\text{post}} \subseteq \text{WFG}_B$ .  $\square$

#### K.5 Theorem K.1 — Liveness Preservation

Statement: If the source-level bounded-FIFO system  $\text{Sys}_B$  is deadlock-free and the Appendix I/J correspondence prerequisites hold (in particular NB-CF, or else RTL-consistent witness-selected NB outcomes for latency-sensitive NB control flow), then  $\text{RTL}_B$  is also deadlock-free.

This theorem preserves absence of message-passing deadlock (WFG over blocking Push/Pop).

SyncChannel/barrier deadlocks are treated as specification errors and are out of scope here.

Proof (by Contradiction):

1. Assume  $\text{RTL}_B$  reaches a deadlocked state  $\sigma_{\text{post}}$ .
2. Extract Cycle: By Lemma K.1, there exists a non-empty set of processes D forming a closed cycle in  $\text{WFG}_{\text{post}}$ .
3. Map to Source-Level Buffered Semantics: By Corollary J.2 (State Correspondence at the End of a Matched Observable Prefix), there exists a corresponding source-level state  $\sigma_B$  of  $\text{Sys}_B$  such that  $\sigma_B \equiv \sigma_{\text{post}}$ .
4. Transfer Cycle: By Lemma K.2, every edge  $P \rightarrow Q$  in  $\text{WFG}_{\text{post}}$  is also present in  $\text{WFG}_B$ . Since D is a closed SCC in  $\text{WFG}_{\text{post}}$ , D has no outgoing edges: for each  $P \in D$  and for every outgoing wait-for edge  $P \rightarrow Q$  in  $\text{WFG}_{\text{post}}$ , we have  $Q \in D$ . By A6 (well-defined complementary endpoint per channel), the channel and complementary endpoint are the same at the source level, so each such outgoing edge  $P \rightarrow Q$  in  $\text{WFG}_{\text{post}}$  corresponds to an outgoing edge  $P \rightarrow Q$  in  $\text{WFG}_B$  with the same target  $Q \in D$ . Hence D has no outgoing edges in  $\text{WFG}_B$  and forms a closed wait-for cycle there as well.
5. Contradiction: This implies  $\text{Sys}_B$  is deadlocked in state  $\sigma_B$ , contradicting the Source-Level Liveness Assumption (A5).
6. Conclusion:  $\text{RTL}_B$  cannot deadlock.  $\square$

## 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  $\text{PopNB}()$  on its inputs, and blocking  $\text{Push}()$  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 under-specified with respect to micro-timing/latency: subject to the R-rules, B-rules, weak fairness, and the happens-before relation on  $\Sigma$  (committed IO events; unsuccessful non-blocking polls are  $\epsilon$ -steps), it can admit multiple  $\epsilon$ -/latency-different executions that respect the same happens-before constraints. This is compatible with B1, which asserts that the post-HLS  $\Sigma$ -projected trace is unique under a fixed stimulus; the pre-HLS side may still have multiple admissible realizations that (when projected to  $\Sigma$ ) match that unique RTL  $\Sigma$ -trace.

Snooping does not change what executions are allowed by the pre-HLS specification. Instead, it selects—purely for verification purposes—one admissible pre-HLS execution whose latency-sensitive events align with those observed in the RTL. In other words, the implementation is used to pick a witness execution of the specification, not to redefine the specification.

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  $\approx$  with  $\Sigma$  instantiated to include only the DUT-boundary observables (Appendix G; Common Formal Definitions above). 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 at the DUT boundary. In such cases, snooping is not required. (See also Note 1 below).

(More precisely: the equivalence relation  $\approx$  is parameterized by the chosen observable alphabet  $\Sigma$ . If internal functionality (such as the above RAM arbiter) uses channels/signals that are not designated observable in the chosen instantiation of  $\Sigma$  (see Common Formal Definitions), then mismatches on those internal actions are outside  $\approx$  by construction: they are not in  $\Sigma$ . If those internal channels/signals are designated observable (e.g., by snooping for debug, or by expanding  $\Sigma$  for Appendix K deadlock analysis), then they are  $\Sigma$ -events and must match under  $\approx$ . In addition, even when B1 holds (unique post-HLS  $\Sigma$ -trace under fixed stimulus), the pre-HLS model may still admit multiple  $\epsilon$ -/latency-different executions consistent with the same happens-before constraints; Corollary J.1 captures the intended quantification ("existence of a matching witness execution prefix") rather than requiring a single pre-chosen trace.)

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 (Unobservable non-blocking micro-timing).

Safety / functional equivalence. The equivalence relation  $\approx$  is parameterized by the chosen observable alphabet  $\Sigma$  (Appendix G). If, for safety-only checking, we instantiate  $\Sigma$  to include only DUT-boundary observables ( $\Sigma_{\text{DUT}}$ ) and we assume that any latency-sensitive internal behavior is not visible at that boundary under  $\approx$ —i.e., it may change only internal micro-timing, but it does not change the values, ordering, or presence of  $\Sigma_{\text{DUT}}$  events—then snooping of those internal latency-sensitive interfaces is not required for proving  $\Sigma_{\text{DUT}}$ -safety. Any mismatches on actions outside  $\Sigma_{\text{DUT}}$  are outside  $\approx$  by construction.

Liveness / deadlock analysis. For message-passing deadlock analysis (Appendix K WFG), any latency-sensitive interface whose  $\epsilon$ -level choices can affect whether a blocking Push/Pop is enabled, or can create/remove a wait-for dependency without an intervening committed  $\Sigma$ -event, must be accounted for. Practically, the default is to snoop and replay every such latency-sensitive (typically non-blocking) arbitration/probe interface in the DUT. Snooping can be avoided only if the design is certified “WFG-inert” with respect to that interface: it cannot change WFG edges except via committed Push\_c/Pop\_c (or explicit synchronization) events.

## 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 source model and the post-HLS RTL, expressed as partial-order constraints on observable I/O actions and encapsulated in equivalence rules E1–E5. For channels whose RTL implementations introduce finite buffering, the correspondence is stated against a source-level bounded-FIFO interpretation Sys\_B whose capacities  $B(c)$  match RTL\_B; the rendezvous case  $B(c)=0$  is recovered as a special case.

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 (ready/valid form).  
For every channel  $c$  with capacity  $B(c)$ , interpret “continuously enabled” in terms of continuous request, not as an immediate commit.  
(No deassert-before-commit assumption.) For blocking Push\_c/Pop\_c transfers, these requests are non-withdrawable: once asserted for a transfer attempt, they are not deasserted prior to the corresponding commit (and Push\_c payload remains stable while vld\_c is asserted). Let vld\_c denote the producer’s persistent request to transfer (e.g., valid=1 with a stable payload for a Push\_c), and let rdy\_c denote the consumer’s persistent request to transfer (e.g., ready=1 for a Pop\_c).  
• P\_push(c): If vld\_c remains asserted continuously from some cycle  $t_0$  onward while the channel is full ( $occ_c = B(c)$ ), and the complementary endpoint process continuously requests the matching Pop\_c (i.e., rdy\_c remains asserted continuously, with the Pop\_c boundary request (i.e., BoundaryReq(Pop\_c,t) remaining true), then some Pop\_c commit must eventually occur

(i.e., the full condition is eventually discharged). Equivalently: when both endpoints continuously request the transfer, the system cannot remain stuck forever in the full state without a Pop\_c commit.

- P\_pop(c): If rdy\_c remains asserted continuously from some cycle  $t_0$  onward while the channel is empty (occ\_c = 0), and the complementary endpoint process continuously requests the matching Push\_c (i.e., vld\_c remains asserted continuously with stable payload, with the Push\_c boundary request (i.e., BoundaryReq(Push\_c,t)) remaining true), then some Push\_c commit must eventually occur (i.e., the empty condition is eventually discharged). Equivalently: when both endpoints continuously request the transfer, the system cannot remain stuck forever in the empty state without a Push\_c commit. (For B(c)=0 rendezvous channels: if both endpoints continuously request the transfer from some cycle  $t_0$  onward (both requests persistent; guards remain true), then the rendezvous completion 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  $\epsilon$ -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  $\epsilon$ -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.

---

## Appendix P - AI Discussion of Alternative Formal Frameworks

The following links provide an AI discussion of alternative formal frameworks compared to the one presented in this document:

<https://gemini.google.com/share/e36f154864a9>

<https://chatgpt.com/share/69457119-ec30-8006-8684-9a2a8b19f96f>