Assume or Assert? Separating the Environment from the Design in SVA

Why assertion statements must identify assumptions separately from assertions: a valid/ready hold example with RAG-SVA code and open-source tool results

Ben Cohen · Dr. Kais Chibani · systemverilog.us · November 2026

Abstract

A requirement such as *valid must hold, with stable data, until ready* is usually written as one `assert property`. Whether it should be an assertion at all depends on who drives the signals. If the design under test (DUT) drives `valid` and `data`, the property is an obligation: the DUT must keep it, and formal verification must prove it. If the environment drives them, the same property is an assumption: it limits what the formal tool may try at the DUT inputs, and the tool never checks it. We take one property through both cases, write it under the RAG-SVA rules (Cohen and Chibani, 2026), and run open-source tools on small designs to show what happens when the label is wrong. An obligation labeled as an assumption is the dangerous error: the check disappears, legal inputs are removed, and the proof reports success on a design with a real bug. An assumption labeled as an assertion, or left out, produces counterexamples that are not DUT bugs. Labeling each property by signal ownership, documenting every assumption, and covering both sides prevents both errors.

1. The requirement and the original statement

The requirement, as an engineer would give it:

Valid must hold, with stable data, until ready.

And the assertion statement that usually comes with it:

```
// Valid must hold, with stable data, until ready
a_hold: assert property (@(posedge clk) disable iff (!rst_n)
  (valid && !ready) |=> (valid && $stable(data)));
```

The temporal logic is reasonable: if `valid` is high and `ready` is low, then in the next cycle `valid` is still high and `data` has not changed. Under the RAG-SVA rules the statement still has four problems:

- `|=>` is banned; the rules write the next-cycle implication as `|-> ##1` (step0, INC-3).
- The label `a_hold` does not follow the naming rule (DOC3): an assertion is `ap_`, an assumption `mp_`, a cover `cp_`, and the named property `p_`.
- There is no cover (UR-5), so nothing shows the property was ever exercised rather than passing vacuously.
- Most important: the statement does not say **who drives** `valid`, `ready` and `data`. That decides whether it is an assertion or an assumption, and nothing in the code records it.

The rest of this paper is about the last point.

2. Assumptions and assertions do different jobs

In formal verification the two keywords are not interchangeable styles. An `assert property` is a claim the tool must **prove** for every input sequence it is allowed to try. An `assume property` **restricts** the input sequences the tool may try; it is taken as true and is never checked against the design. The property text can be identical; the keyword decides whether the tool treats it as a goal or as a given.

The correct keyword follows from signal ownership. RAG-SVA records ownership in a PORT-INTENT table, where each signal has an INTENTION: `assume` for signals driven by the environment, `obligation` for signals driven by the DUT. The INTENTION reference (§4.1) then fixes the label from the signals that appear on each side of the implication:

Row	Antecedent	Consequent	Result	Label
1	assume	assume	assume property	mp_
2	assume	obligation	assert property	ap_
3	obligation	assume	HALT (CASE D)	—
4	obligation	obligation	assert property	ap_
5	obligation (release trigger only)	assume	assume property, F1-A exception	mp_
6	mixed	obligation	assert property	ap_
7	any	mixed	HALT (CASE C): decompose	—

The one-line rule: if the consequent contains any obligation signal, the property is an `ap_`; if it contains only assume signals, it is an `mp_`. Figure 1 shows the boundary this draws

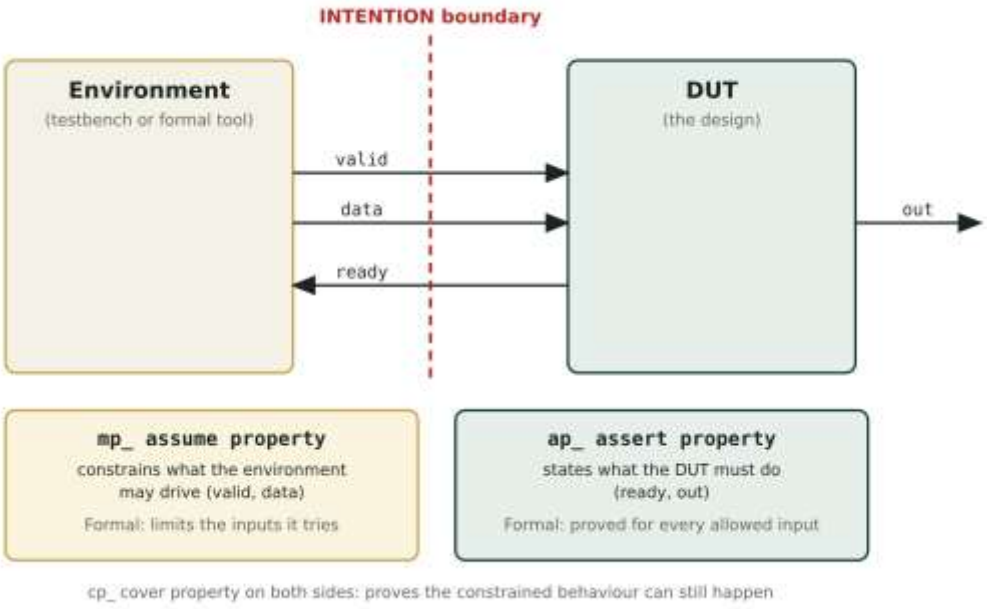


Figure 1. The INTENTION boundary. Assumptions constrain what the environment drives; assertions state what the DUT must do; covers on both sides show that the constrained behavior can still happen.

In simulation the difference is smaller: the testbench, not the property, generates the inputs, and simulators check an `assume property` much like an `assert property` (Section 6 shows this with Verilator). The label still matters there, because it tells the engineer **where the bug is**: a failing `ap_` points at the DUT, a failing `mp_` points at the testbench or at the source block.

3. Case (a): valid, ready and data are obligations

In the first case all three signals are driven inside the DUT, for example an internal handshake between a producer and a consumer that are both part of the block being verified. The design `vr_link` is such a block: the producer drives `valid` and `data`, the consumer drives `ready` from an external back-pressure input `stall_in`.

```
module vr_link (
  input  logic       clk, rst_n,
  input  logic       start,     // request to send din
  input  logic [7:0] din,
  input  logic       stall_in,  // consumer back-pressure from outside
  output logic [7:0] dout,
  output logic       dout_vld
);
  logic       valid, ready;
  logic [7:0] data;
  logic [1:0] wait_cnt;          // cycles valid has waited for ready

  // ---------------- producer ----------------
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      valid <= 1'b0; data <= '0; wait_cnt <= '0;
    end else if (valid && ready) begin
      valid <= 1'b0; wait_cnt <= '0;
    end else if (!valid && start) begin
      valid <= 1'b1; data <= din; wait_cnt <= '0;
    end else if (valid) begin
      wait_cnt <= wait_cnt + 2'd1;
`ifdef BUG_DROP
      if (wait_cnt == 2'd2) valid <= 1'b0;   // BUG: gives up after 3 stall cycles
`endif
`ifdef BUG_DATA
      data <= din;                            // BUG: reloads data while stalled
`endif
    end

  // ---------------- consumer ----------------
  assign ready = !stall_in;
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      dout <= '0; dout_vld <= 1'b0;
    end else begin
      dout_vld <= valid && ready;
      if (valid && ready) dout <= data;
    end
endmodule
```

Antecedent and consequent are both obligations (row 4), so the property is an assertion. The RAG-SVA checker, bound to the internal signals:

```
// vr_link_sva.sv -- RAG-SVA checker for vr_link (case a: valid, ready, data = obligation)
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// PORT-INTENT: valid=obligation, ready=obligation, data=obligation, start/din/stall_in=assume
module vr_link_sva (
  input logic       clk, rst_n,
  input logic       valid, ready,
  input logic [7:0] data
);
  property p_valid_hold_until_ready;
    (valid && !ready) |-> ##1 (valid && $stable(data));
  endproperty

  ap_valid_hold_until_ready: assert property (@(posedge clk) disable iff (!rst_n)
    p_valid_hold_until_ready)
    else $error("FAIL: valid dropped or data changed before ready, time=%0t", $realtime);

  cp_valid_hold_until_ready_trigger: cover property (@(posedge clk) disable iff (!rst_n)
    valid && !ready);

  cp_valid_hold_until_ready_full: cover property (@(posedge clk) disable iff (!rst_n)
    (valid && !ready)[*2] ##1 (valid && ready && $stable(data)));
endmodule

bind vr_link vr_link_sva u_vr_link_sva (.clk, .rst_n, .valid, .ready, .data);
```

Two bugs can be switched on with a define: `BUG_DROP` makes the producer give up after three stall cycles, and `BUG_DATA` reloads `data` from `din` while stalled.

4. Case (b): valid and data are inputs, ready is an obligation

In the second case the DUT is the consumer, `vr_sink`. The environment drives `valid` and `data`; the DUT drives `ready`. Like many real designs, it captures `data` on the first cycle `valid` is seen and outputs it at the handshake, so it **relies** on the source keeping `data` stable during a stall.

```
module vr_sink (
  input  logic       clk, rst_n,
  input  logic       valid,
  input  logic [7:0] data,
  output logic       ready,
  output logic [7:0] out,
  output logic       out_vld
);
  logic [1:0] cnt;               // stall counter: ready after 2 wait cycles
  logic [7:0] data_q;            // captured when valid is first seen

  assign ready = valid && (cnt == 2'd2);

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      cnt <= '0; data_q <= '0; out <= '0; out_vld <= 1'b0;
    end else begin
      out_vld <= valid && ready;
      if (valid && ready) begin
        out <= data_q;               // relies on data being stable while stalled
        cnt <= '0;
      end else if (valid) begin
        if (cnt == 2'd0) data_q <= data;
```

```
        cnt <= cnt + 2'd1;
      end
    end
endmodule
```

Now the same hold property constrains environment signals: its consequent contains only `valid` and `data`, which are assume signals. It must be an assumption, `mp_`. But `ready`, an obligation, sits in the antecedent. RAG-SVA allows that only under the F1-A exception, where the obligation signal is a **release trigger** that ends the hold and is never constrained; the user must confirm it, the property carries an inline comment, and the DOC9 header carries a NOTE line. Without that confirmation the rules HALT and offer two resolutions.

4.1 Option A: F1-A release trigger

```
module vr_sink_sva (
  input logic       clk, rst_n,
  input logic       valid, ready, out_vld,
  input logic [7:0] data, out
);
  // ---------------- environment: assumption ----------------
  property p_valid_hold_until_ready;
    (valid && !ready) |-> ##1 (valid && $stable(data));   // ready: release trigger only
  endproperty
  /*
  ASSUMPTION:    The source holds valid high, with data stable, while ready is low.
  TYPE:          Input Constraint / Hold Stability
  JUSTIFICATION: Protocol: a transfer is offered until the DUT accepts it with ready.
  RISK LEVEL:    Low
  NOTE:          ready (obligation) in antecedent per F1-A -- NOT constrained.
  */
  mp_valid_hold_until_ready: assume property (@(posedge clk) disable iff (!rst_n)
    p_valid_hold_until_ready);

  cp_valid_hold_until_ready_hit: cover property (@(posedge clk) disable iff (!rst_n)
    (valid && !ready)[*2] ##1 (valid && ready));

  // ---------------- DUT: obligation ----------------
  property p_out_matches;
    (valid && ready) |-> ##1 (out_vld && (out == $past(data)));
  endproperty

  ap_out_matches: assert property (@(posedge clk) disable iff (!rst_n)
    p_out_matches)
    else $error("FAIL: out does not match the accepted data, time=%0t", $realtime);

  cp_out_matches_full: cover property (@(posedge clk) disable iff (!rst_n)
    (valid && ready) ##1 out_vld);
endmodule
```

The checker also holds the DUT's own obligation, `ap_out_matches`: after a handshake, `out` must equal the data that was accepted. That assertion is the reason the assumption exists. Without the assumption, it is not a property of this design.

4.2 Option B: decompose

Option B keeps every obligation out of the assumption. The environment holds `valid` and `data` for a fixed window, and a separate assertion requires the DUT to answer within it:

```
// ===== Option B: decompose into mp_ (inputs only) + ap_ (DUT only) =====
property p_valid_hold_window;
  $rose(valid) |-> ##1 (valid && $stable(data))[*MAX_STALL];
endproperty
/*
ASSUMPTION:    After valid rises, the source holds valid high with data stable
               for MAX_STALL more cycles.
TYPE:          Input Constraint / Hold Stability
JUSTIFICATION: Spec bound on the DUT's stall; the hold window is fixed, not
               released by ready, so no obligation signal is referenced.
RISK LEVEL:    Medium -- the source keeps valid for the full window even after
               ready, and only transfers that start with a rise of valid are
               checked (back-to-back transfers with valid held high are not).
*/
mp_valid_hold_window: assume property (@(posedge clk) disable iff (!rst_n)
  p_valid_hold_window);

cp_valid_hold_window_hit: cover property (@(posedge clk) disable iff (!rst_n)
  $rose(valid) ##1 (valid && $stable(data))[*MAX_STALL]);

property p_ready_within_hold;
  $rose(valid) |-> ##[0:MAX_STALL] ready;
endproperty

ap_ready_within_hold: assert property (@(posedge clk) disable iff (!rst_n)
  p_ready_within_hold)
  else $error("FAIL: ready not seen within %0d cycles of valid, time=%0t",
              MAX_STALL, $realtime);

cp_ready_within_hold_trigger: cover property (@(posedge clk) disable iff (!rst_n)
  $rose(valid));

cp_ready_within_hold_full: cover property (@(posedge clk) disable iff (!rst_n)
```

`MAX_STALL` has no default: the requirement gives no bound, so whoever instantiates the checker must set it from the specification. The price is recorded in the DOC9 header (RISK LEVEL: Medium): the environment keeps `valid` high for the whole window even after `ready`, only transfers that start with a rise of `valid` are checked, and the DUT gains an obligation the requirement never stated. Option A is the faithful translation of the requirement; Option B is the choice when an assumption may not mention a DUT signal at all.

5. What goes wrong when the label is wrong

Figure 2 crosses the two questions: is the property about the environment or about the DUT, and is it labeled `assume` or `assert`? Two cells are correct. The other two are the errors this paper is about. We ran each cell with an open-source formal engine (Section 7 gives the method).

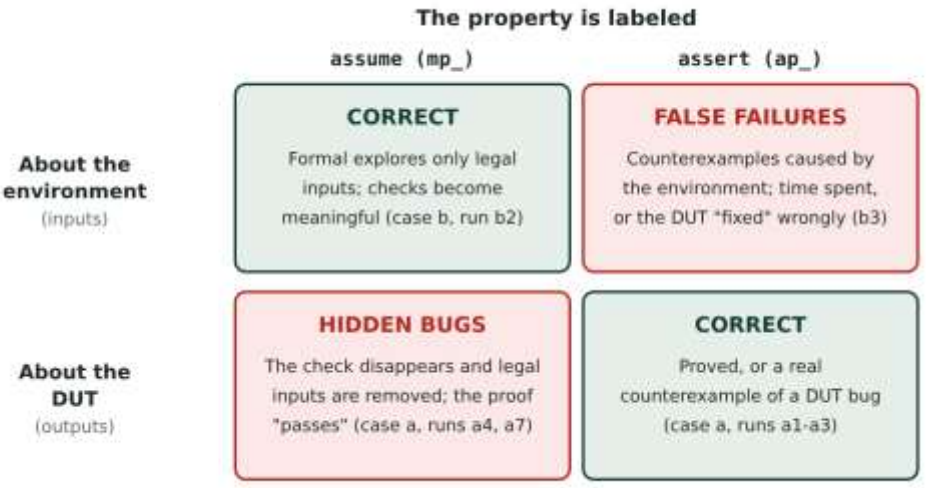


Figure 2. The four combinations of what a property describes and how it is labeled. Run names refer to Table 2.

5.1 An obligation labeled as an assumption hides bugs

Take case (a), where the producer is inside the DUT, and write the hold property as `assume` instead of `assert`. With `BUG_DROP` or `BUG_DATA` switched on, the formal run reports **success** (runs a4 and a5). It has nothing to fail: the only check became a constraint, and the run contains zero assertions. Worse, the assumption is about DUT outputs, so the tool satisfies it by removing **inputs**. The producer drops `valid` after three stall cycles, so the only way to keep the assumed property true is never to stall three cycles in a row. A reachability probe shows exactly that (Figure 3): with the correct `ap_` label a three-cycle stall is reachable (run a6, counterexample); with the `mp_` label it is unreachable (run a7, proved). Every input sequence that would expose the bug is gone.

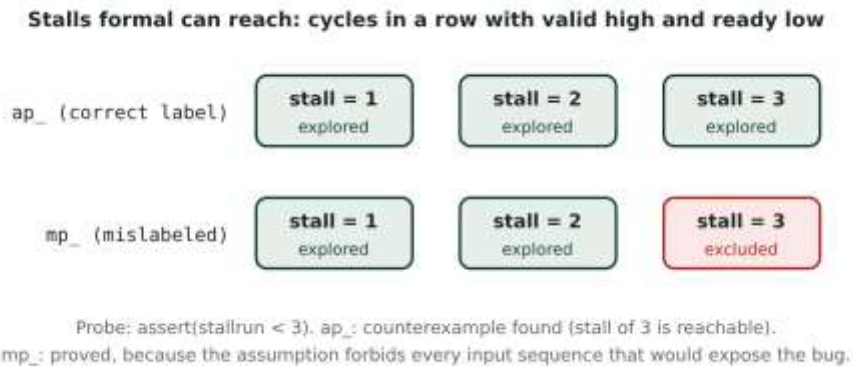


Figure 3. Stalls the formal tool can reach on the BUG_DROP producer. Labeling the DUT's own property as an assumption removes the three-cycle stall from the input space.

This is the error a review must catch, because the tools will not: a proof that passes because of an over-constraining assumption looks exactly like a proof that passes because the design is correct.

5.2 An assumption labeled as an assertion gives false failures

Take case (b) and leave the environment unconstrained, or write its hold property as `assert`. The DUT's own assertion `ap_out_matches` fails (run b1): the tool changes `data` in the cycle `ready` rises, after a stall, and the correct design outputs the value it captured (Figure 4b). If the hold property is written as `assert` (run b3), it fails too, with a trace made entirely of input values. Neither is a DUT bug. In practice such counterexamples cost debug time, and they push engineers toward the worst fix: changing a correct design, or adding an ad hoc assumption with no documentation.

5.3 With the correct labels

Case (a) with `ap_`: the correct producer is proved by induction (run a1), and both bugs give real counterexamples (runs a2, a3; Figure 4a). Case (b) with the F1-A `mp_`: `ap_out_matches` is proved by induction (run b2, with three helper invariants that are themselves proved). The property text never changed; only the keyword and the reason for it did.

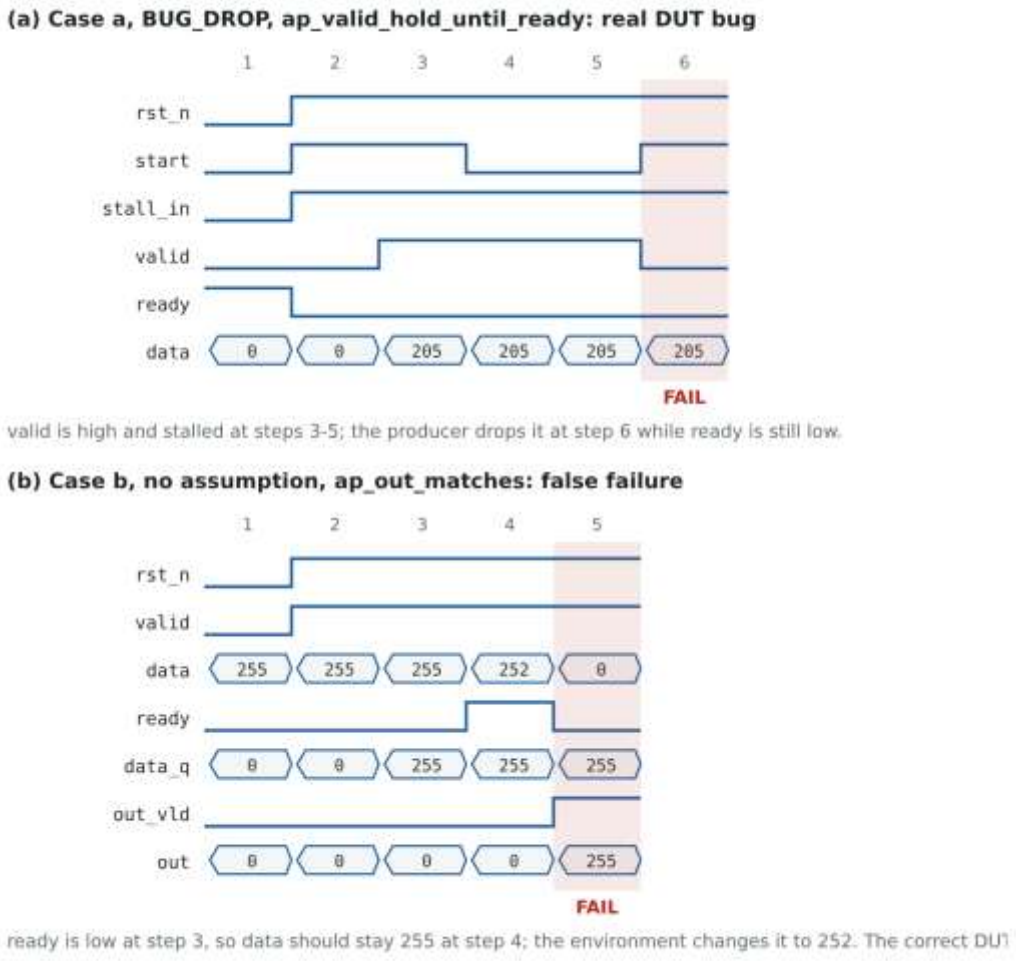


Figure 4. Counterexamples from the formal runs. (a) A real DUT bug found by the correct ap_. (b) A false failure caused by an unconstrained environment.

6. In simulation

Both designs were simulated with their RAG-SVA checkers and random testbenches. The vr_sink testbench follows the protocol by default; with +ROGUE it changes data during a stall, breaking the assumption. To see what a mislabel does in simulation, case (a) was also run with the hold property relabeled as an assumption.

Run	Labels	Result	What it shows
vr_link, correct, seeds 1–3	ap_	0 failures; 440–453 transfers; covers hit (trigger 638–674, full 153–160)	The correct producer keeps the hold
vr_link, BUG_DROP, seed 1	ap_	104 failures, first at time 155	Real DUT bug, blamed on the DUT
vr_link, BUG_DATA, seed 1	ap_	659 failures, first at time 95	Real DUT bug, blamed on the DUT
vr_link, BUG_DROP, seed 1	mp_ (mislabeled)	104 failures, reported as a failed assumption	Simulation still sees the bug, but the label blames the environment
vr_sink, protocol kept, seeds 1–3	mp_ + ap_	0 failures; 491–504 transfers; scoreboard 0 errors; covers hit	Assumption holds, DUT correct
vr_sink, +ROGUE, seed 1	mp_ + ap_	mp_ fails 250 times (first at 205); ap_out_matches 220; scoreboard 220	The failing mp_ points at the testbench

Two points follow. First, Verilator checks an `assume property` like an assertion, so in simulation the mislabeled case (a) still fails, 104 times, exactly as the `ap_` does. Simulation does not hide the bug; it misattributes it, telling the engineer the environment broke a rule when the DUT did. Formal is where the same mislabel becomes silent (Section 5.1). Second, in case (b) the label does the triage: when the rogue testbench breaks the protocol, the `mp_` fails first and the 220 `ap_out_matches` failures that follow are recognizable as consequences, not DUT bugs.

7. Method and tool results

All files compile with slang (pyslang 12.1.0) with 0 diagnostics: the two designs, both RAG-SVA checkers, both testbenches, and the Option A/B checker. Simulation used Verilator 5.055 (development version, built from source; the packaged 5.020 does not support `##1` or `[*2]` in properties), 2000 cycles per run, random stimulus.

Formal runs used Yosys 0.33 (`sat -tempinduct`, up to 20 steps). Open-source Yosys does not read SVA sequences, so each formal run used a small harness with the immediate-assertion equivalent of the property: under `$past(valid && !ready)`, check or assume `valid && data == $past(data)`. The harness is a separate file, included into a copy of the design for the formal run only. These are therefore

results for an equivalent formulation on an open-source engine, not for the SVA text on a commercial formal tool; they demonstrate the behavior the paper describes, not a sign-off.

Run	Configuration	Result	Meaning
a1	vr_link correct, hold as ap_	Proved	Correct design, correct label
a2	vr_link BUG_DROP, hold as ap_	Counterexample at step 6	Real bug found (Figure 4a)
a3	vr_link BUG_DATA, hold as ap_	Counterexample at step 4	Real bug found
a4	vr_link BUG_DROP, hold as mp_	"SUCCESS" with 0 assertions	Bug hidden: nothing left to check
a5	vr_link BUG_DATA, hold as mp_	"SUCCESS" with 0 assertions	Bug hidden
a6	BUG_DROP, ap_, probe stall < 3	Counterexample	A 3-cycle stall is reachable
a7	BUG_DROP, mp_, probe stall < 3	Proved	The assumption removed every 3-cycle stall
b1	vr_sink, no assumption, ap_out_matches	Counterexample at step 5	False failure (Figure 4b)
b2	vr_sink, hold as mp_ (F1-A), ap_out_matches	Proved (with 3 proved helper invariants)	Correct labels: a meaningful proof
b3	vr_sink, hold as ap_	Counterexample	False failure, made of input values only

Table 2. Formal runs. "Proved" means proved by k-induction; "counterexample" means the base case found a trace from reset.

8. How RAG-SVA keeps the two apart

The examples show why the separation cannot be left to style. RAG-SVA makes it a set of checks that run before any code is written:

1. **PORT-INTENT table first.** Every signal is classified `assume` or `obligation` before properties are written. An unknown owner is an ambiguity, and RQM0 halts.
2. **The label follows from the table.** The INTENTION rows fix `mp_` or `ap_` from the signals on each side of the implication; `a_hold` cannot be produced.
3. **Illegal mixes halt.** An obligation in an assumption's consequent is CASE D; a mixed consequent is CASE C. Both stop generation and ask, which is how case (b) surfaced the F1-A decision instead of guessing.
4. **Every assumption is documented.** The DOC9 header (ASSUMPTION, TYPE, JUSTIFICATION, RISK LEVEL, and the F1-A NOTE) lets a reviewer see each constraint and its risk, the review that Section 5.1 says the tools cannot do.
5. **Covers on both sides (UR-5).** A cover on each assumption shows the constrained behavior is still reachable. A cover for the longest legal stall would have exposed run a7: it is unreachable under the mislabeled assumption.

6. **Real tools, truthful reports.** The TOOL EXECUTION rule compiles and simulates, and reports separately what was compiled, simulated, evaluated by an SVA engine, and covered.

9. Conclusion

The same line of SVA can be a proof goal or a given. Which one it is depends on who drives the signals, and that information is not in the property text. Written as an assertion about the environment, it produces counterexamples that are not bugs. Written as an assumption about the design, it removes the check and the inputs that would reveal a bug, and the proof passes. In this example the second error turned two real bugs into a successful formal run. Identifying assumptions separately from assertions, by signal ownership, with documentation and covers, is therefore not a matter of style. It is what makes a formal result mean what it says.

References

[1] IEEE Std 1800-2023, *IEEE Standard for SystemVerilog*, clause 16 (assertions), 16.14 (concurrent assertion statements: assert, assume, cover).

[2] B. Cohen, K. Chibani, *Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules*, Cohen Publishing, 2026.

[3] B. Cohen, K. Chibani, "Degeneracy, Environment vs. Obligation, and Design Intent: A req/ack Case Study," systemverilog.us/req_ack_degeneracy_paper.pdf.

[4] B. Cohen, K. Chibani, "Deriving a Formal req/ack Assertion Under the RAG-SVA Methodology," systemverilog.us/req_ack_hold_window_reasoning.pdf.

[5] B. Cohen, K. Chibani, *RAG-SVA: The Collected Papers*, Cohen Publishing, 2026.

[6] Yosys Open SYnthesis Suite, yosyshq.net; Verilator, verilator.org; slang, sv-lang.com.