Deriving a Formal req/ack Assertion Under the RAG-SVA Methodology

A Worked Example: Hold-Until-Ack and a Bounded Response Window

Ben Cohen — Cohen Publishing (systemverilog.us) · Dr. Kais Chibani

Introduction

Formal verification lives or dies on whether an assertion actually says what the requirement meant. A property that compiles, simulates cleanly, and even passes in a formal tool can still be quietly wrong — checking a different obligation than intended, blaming the design for the environment's behavior, or passing vacuously because it can never fail. The gap between a plain-English requirement and a trustworthy SystemVerilog Assertion is exactly where a disciplined methodology earns its keep.

This paper is a worked example of that discipline. It takes a compact, realistic requirement — *a req/ack handshake in which req must be held until and including the acknowledging cycle, and ack must arrive two to three cycles after req rises* — and derives the assertions for it step by step under the RAG-SVA methodology of Cohen and Chibani's *Knowledge-Driven AI for SystemVerilog Assertions* [1]. The point is not the final code alone but the reasoning that produces it: how signals are classified before any property is written, why one requirement becomes an assumption and the other an assertion, why a banned operator is never reached for in the first place, and where a genuine ambiguity is halted for human confirmation rather than silently resolved. The sections that follow reproduce that derivation in full, ending with the final code and a note on how the rules shaped it.

Reasoning: req/ack Hold + 2-to-3-Cycle Ack Window

LICENSE: Personal/academic use only. Commercial use requires a Commercial License.

Requirement (as given)

- **`req` is input, `ack` is output, `rst_n` reset active-low, `clk` posedge.**
- **R1: if(req) then req holds until and with ack.**
- **R2: ack occurs 2 to 3 cycles after the rise of req.**

[CLAUDE]

Step 1 — Signal classification (intention boundary)

Signal	Direction	Intention	Role
clk	input	N/A	clock
rst_n	input	N/A	reset, active-low
req	input	assume	trigger; hold-until-and-with-ack
ack	output	obligation	response, window [2:3]

Key consequence: R1 constrains an **input** (`req`), so it is an environment constraint — an `mp_` **assumption**, not an assertion. Writing it as an `ap_` would blame the DUT for behavior only the environment controls. R2 obligates an **output** (`ack`), so it is an `ap_` **assertion**.

Step 2 — R1: the hold as an assumption (F1-A form)

"req holds **until and with** ack" = inclusive hold: `req` stays high through the cycle `ack` occurs, and may drop only after it.

Canonical continuation form:

```
(req && !ack) |-> ##1 req    // ack: release trigger only
```

Reasoning for this shape:

- Trigger is **level-based** (`req`, not `$rose(req)`) because the requirement says "if(req)" — any cycle where req is high and unacknowledged obligates the next cycle's req. This also makes the hold self-extending for however long ack takes, with no hardcoded bound that could fall out of sync with R2's window (no hold/window offset hazard).
- `ack` (an obligation signal) appears in the assumption's **antecedent** — normally a boundary violation. It is permitted here strictly as a **release trigger**: it gates when the hold obligation ends and is itself never constrained by the assumption. This exception was explicitly confirmed earlier in this session ("F1 resolution — A") for this exact construct and is carried forward.
- Inclusivity check: if ack fires at cycle k, then at k-1 we had (req && !ack) → req must be high at k. So req is high **on** the ack cycle — "until and with" satisfied. After k, the antecedent is false (ack high), so req is free to drop at k+1.

Step 3 — R2: the ack window as an assertion

"ack occurs 2 to 3 cycles after the **rise** of req":
`$rose(req) |-> strong(##[2:3] ack)`

Reasoning:

- Trigger is **edge-based** (`$rose(req)`) because the requirement anchors the window to "the rise of req" — unlike R1, which is level-anchored. Using a level trigger here would re-fire the window on every held cycle and produce overlapping, meaningless obligations.
- Window `##[2:3]` — explicit bounded delay, exactly as stated. Cycle 1 is outside the window (see Open Item below).
- `ack` checked as a **level** at the window cycles. The requirement says "ack occurs," with no pulse-width language, so the level-based default applies (no $rose forced on the consequent, no pulse-width check invented).
- `strong()` on the bounded consequent: consistent with the session's established convention for bounded response obligations. In formal verification, weak bounded sequences cannot pass by trace exhaustion anyway on infinite traces, but in simulation a weak form passes silently if the test ends mid-window; `strong()` makes an unfinished window a failure rather than a silent pass. Classification: SAFETY-leaning bounded response (no liveness keyword such as "eventually"/"livelock" appears in the requirement, so no fairness/livelock companion assumptions are mandated by keyword scan).

Step 4 — Consistency check between R1 and R2

- R2 forces ack by cycle 3 after a req rise. R1 holds req high until that ack. No conflict: the hold self-extends to meet whichever cycle (2 or 3) ack arrives, inclusive.
- No hold/window offset hazard: because R1 uses the continuation form (`##1 req`) rather than a fixed-length hold (`req[*2:3] intersect ...`), there is no second set of bounds that must be kept aligned (+1) with the assertion window.

Step 5 — Coverage

Cover	**Purpose**
`cp_req_to_ack_trigger` — `$rose(req)`	antecedent reachability (assertion can fire)
`cp_req_to_ack_full` — `$rose(req) ##[2:3] ack`	full path exercised end-to-end (vacuity defense)

Cover	Purpose
`cp_req_holds_until_ack_hit` — `req && !ack`	assumption's active state reachable (an unreachable assumption either over-constrains or constrains nothing)

Open Item (flagged, not silently resolved)

The requirement "ack occurs 2 to 3 cycles after the rise of req" does **not** state whether ack at cycle 1 is *forbidden* or merely *not required*. `##[2:3] ack` only requires ack somewhere in cycles 2–3; an early ack at cycle 1 would not violate it. If early ack must be forbidden, add:

```
// OPTIONAL — enable only if "2 to 3" is exclusive (no early ack):
// ap_no_early_ack: assert property (@(posedge clk) disable iff (!rst_n)
//   $rose(req) |-> ##1 !ack);
```

This is left commented out rather than guessed into the active set.

Final code

```
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// R1 — environment hold: req holds until-and-including the ack cycle.
// F1-A: ack in antecedent as release trigger only (user-confirmed).
property p_req_holds_until_ack;
  (req && !ack) |-> ##1 req;
endproperty

mp_req_holds_until_ack: assume property (@(posedge clk) disable iff (!rst_n)
  p_req_holds_until_ack);
// R2 — DUT obligation: ack within 2 to 3 cycles of the rise of req.

property p_req_to_ack_window;
  $rose(req) |-> strong(##[2:3] ack);
endproperty
ap_req_to_ack_window: assert property (@(posedge clk) disable iff (!rst_n)
  p_req_to_ack_window)
  else $error("FV: ack did not occur within [2:3] cycles of req rise, time=%0t",$realtime);

// Coverage
cp_req_to_ack_trigger: cover property (@(posedge clk) disable iff (!rst_n) $rose(req));

cp_req_to_ack_full: cover property (@(posedge clk) disable iff (!rst_n)
  $rose(req) ##[2:3] ack);

cp_req_holds_until_ack_hit: cover property (@(posedge clk) disable iff (!rst_n)
  req && !ack);

// OPTIONAL — enable only if "2 to 3" is exclusive (no early ack permitted):
// ap_no_early_ack: assert property (@(posedge clk) disable iff (!rst_n)
//   $rose(req) |-> ##1 !ack);
```

//==

[User] Why did you not use `$rose(req) |-> req until_with ack;` along with `$rose(req) |-> strong(##[2:3] ack);`
Why did you handle the `until_with` this way: `(req && !ack) |-> ##1 req;`

[Claude] Good question, and the honest answer has two parts: one about your rule corpus, and one about a semantic subtlety that matters independent of the corpus.

Part 1 — `until_with` is on your banned-operator list.

From the rules loaded this session (rag_make.md overrides, echoed in step0.md's banned set), `until` and `until_with` are both prohibited, alongside `within`, `first_match`, and unbounded `##[0:$]`. So `req until_with ack` isn't something I can emit under your own corpus regardless of whether it's semantically correct — CHECK 17-adjacent operator bans would halt on it. The `(req && !ack) |-> ##1 req` continuation form is the corpus's prescribed replacement for exactly this "hold until X" pattern. That alone is the governing reason in your framework.

But you deserve the *why behind the ban*, not just "the rules say so" — so:
Part 2 — the two forms don't actually mean the same thing.
`req until_with ack` (LRM property) means: `req` holds continuously from now up to **and including** the cycle where `ack` is true, and — critically — it *requires that ack eventually occur*. `until_with` is the **weak-to-strong** distinction point: plain `until` is weak (ack need not ever arrive), `until_with`/`s_until_with` carry an inclusivity and (for the `s_` form) an eventuality obligation.

That creates a subtle problem when you pair it with your R2 assertion:

- `$rose(req) |-> req until_with ack` — placed as an **assertion** — makes the *hold-and-eventual-ack* a DUT obligation. But holding `req` is the **environment's** job (`req` is an input). You'd be asserting that the DUT guarantees an input behaves a certain way, which inverts the intention boundary.
- Even placed as an assumption, `until_with` folds two distinct claims into one operator: "req stays high" (environment's duty) **and** "ack eventually happens" (the DUT's duty, already covered by your R2 `strong(##[2:3] ack)`). Bundling the DUT's eventual-ack obligation into the environment assumption double-counts it, and worse, an *assumed* eventuality can mask a real DUT failure — the tool would take "ack eventually arrives" as a given rather than checking it.

The continuation form `(req && !ack) |-> ##1 req` deliberately encodes **only** the environment's half: "*as long as req is high and unacked, req stays high next cycle*." It says nothing about whether ack ever arrives — that obligation lives entirely in R2, where it belongs

and where it's *checked* rather than *assumed*. The two properties stay cleanly separated: R1 = environment holds req, R2 = DUT must ack in [2:3]. `until_with` would blur that line.

Part 3 — why not `until_with` for R1 even as a pure hold?
Because `req until_with ack` still carries the eventuality/inclusivity semantics that make it heavier than needed. Your R2 already guarantees ack by cycle 3, so R1 doesn't need an operator that *also* promises ack arrives — it just needs to keep req high in the meantime. The continuation form is the minimal, separable encoding: no eventuality baked in, no bound to keep aligned with R2's window, and no banned operator.

Net: functionally, under the R1+R2 pair, `(req && !ack) |-> ##1 req` and a correctly-placed `req until_with ack` would produce nearly the same accepted traces — but the continuation form is the one your corpus permits, keeps the environment/obligation split clean, and avoids assuming the very eventuality R2 is supposed to prove.

[User] Q: Did you first use the LLM to arrive at code, then check the results against the RAG-SVA rules?

[Claude] No — and this is the important distinction. The rules acted as **constraints on the derivation**, not as a filter applied after code already existed. Here's the honest ordering of what happened:

1. **Classification first, before any property text.** The very first move was the intention-boundary table (*input→assume, output→obligation*). That decision — made from the rules, not from drafting code — is what determined R1 must be an `mp_` and R2 an `ap_`. If I'd generated code first and checked after, I might well have written both as assertions and only caught it on review.

2. **The operator bans pruned the search space up front.** `req until_with ack` is the natural first thing an unconstrained LLM reaches for. Because `until_with` is on your banned list, it was excluded *before* being written, which is what steered me to the continuation form. That's prevention, not correction.

3. **The remaining choices** (edge vs. level trigger, `strong()` on the bounded consequent, UR-5 paired covers, halting on the early-ack ambiguity) each trace to a named rule, not to free-form judgment.

That said, full honesty about the mechanics: I'm a single model, so there isn't literally a separate "generator process" and "checker process" running as distinct programs — the rules and the generation happen in the same pass. The meaningful claim isn't that two separate engines ran; **it's that the rules shaped the output *as it was formed* rather than being consulted afterward.** The difference shows up in cases like `until_with`: a check-after-the-fact approach would have produced the banned operator, then flagged it. A constrain-during approach never

produces it. This session's behavior was the latter — which is exactly what the RAG-SVA methodology is designed to enforce, and why **the corpus is loaded *before* the requirement rather than applied to finished code.**

The one caveat worth stating plainly: verifying that this discipline actually held (rather than my just asserting it did) is exactly what the reasoning .md file is for — it exposes each decision to a named rule so you can audit the chain yourself, rather than taking "the rules were followed" on faith.

Reference

[1] Ben Cohen and Kais Chibani, *Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules*, Cohen Publishing, 2026.
Book ships with downloadable RAG-SVA Files
Available at https://systemverilog.us/