From Ambiguous Requirement to Verified Assertions

A Case Study and Reusable Template for SVA Requirement Writing

Ben Cohen and Kais Chibani

systemverilog.us

Abstract

A single small verification task — a two-lane request/ready/acknowledge (req/rdy/ack) handshake — needed five rounds of clarification using the RAG-SVA rules[1] before it could be safely turned into a SystemVerilog Assertion (SVA) checker, including one real gap that slipped past a rigorous halt-on-ambiguity review. This paper shows the original requirement, explains briefly why each part of it was ambiguous, and then presents a reusable requirement template built from that experience — filled in for this exact case as a worked example — followed by the complete generated checker module.

1. The Original Requirement

The task began with the following requirement, given in full, as originally written:

> *"write assertions for these requirements.*
> *req [0:1] input, rdy[1] local variable set by cpu, ack[0:1] output.*
> *[0] has higher priority. req to rdy in 1 to 2 cycles; rdy to ack in 1 to 3 cycles.*
> *posedge clk, rst_n neg edge"*

2. Why It Was Ambiguous — Briefly

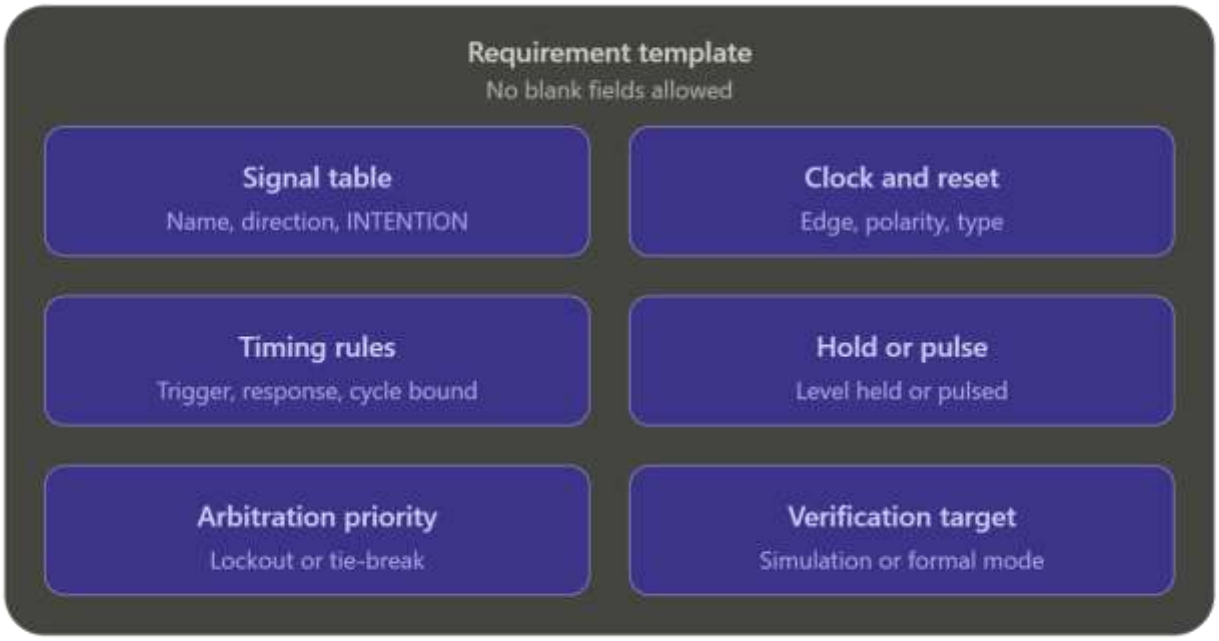
- **Inconsistent signal width. `rdy`** was named `rdy[1]` — a single, oddly-indexed signal — while `req` and `ack` were both clearly 2-bit vectors. It should have been `rdy[0:1]` from the start.
- **Undefined signal INTENTION.** `req` and `ack` get their assume/obligation classification for free from port direction, but `rdy` is a module-local variable with no port direction — its classification (assume, because the CPU drives it) had to be stated explicitly, not left for inference.
- **Priority contradicted the timing bound.** "[0] has higher priority" was left open to two readings — a full lockout of lane 1, or a narrower same-cycle tie-break — and the full-lockout reading directly contradicts the unconditional 1–2 cycle timing bound already given for lane 1. This was a genuine logical contradiction, not just missing detail.
- **Hold-until-ack was reasoning, not a rule.** that req/rdy hold level until ack was true and relevant, but it was only ever spoken as justification for a different decision, never written as its own rule — so it was never checked, never generated, and shipped silently missing until caught by inspection after the fact.
- **A hidden requirement came from settings, not from the prompt.** Formal Verification Mode being enabled mandated fairness and livelock companion assumptions with no analogue in the plain-

[1] Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules 2026, Ben Cohen & Kais Chibani

English requirement at all — a requirement coming from tooling configuration, not from anything the user wrote.

3. A Reusable Requirement Template

The template below exists to front-load exactly these clarifications, so a handshake-style requirement can go straight to code without a back-and-forth round. Fill in every field; write “N/A” rather than leaving a field blank, since a blank field is itself an ambiguity.



3.1 Signal Table

List every signal. For each one:

```
Name / width: e.g. req[0:1]
Direction: input / output / module-local variable
INTENTION: assume (environment-driven) / obligation (DUT must produce it)
```

For a module-local variable specifically: state who or what sets it, and whether that source is inside the environment (CPU, testbench, another unverified block) or is itself computed from a DUT obligation signal — a signal derived from an obligation signal is itself an obligation, even if it looks like internal plumbing.

3.2 Clock and Reset

```
Clock: e.g. posedge clk
Reset signal, polarity, and type: e.g. rst_n, active-low, asynchronous
```

3.3 Timing Rules

For every cause → effect relationship between two signals, state:

- **Trigger condition:** e.g. “whenever req[i] rises”
- **Required response:** e.g. “rdy[i] must assert”
- **Cycle bound:** a number or explicit range, inclusive or exclusive on both ends. “Soon,” “eventually,” or “quickly” are not bounds.
- **Which side owns this rule:** is this a promise the environment makes (assume) or a promise the DUT must keep (obligation)?

3.4 Hold / Pulse Behavior — Do Not Skip This Section

For every signal in Section 3.3: does it pulse for one cycle and drop, or hold level until something releases it? If it holds, name the exact release condition. This is the field most likely to be true, relevant, and never written down.

```
Signal: hold or pulse?
If hold, released by: the exact condition that lets it deassert
```

3.5 Arbitration / Priority Rules

If more than one lane, channel, or requester can be active at once: does the higher-priority side block the lower-priority side completely for as long as it's active, or only during a narrower window (e.g. until it receives its own grant)? A phrase like “X has priority” is not precise enough alone — say which of these two it means, and check it against every timing bound already stated for the lower-priority side for a contradiction.

3.6 Verification Target

Simulation-only or formal verification:

Formal verification requires fairness assumptions (so a signal can't stay inactive forever to vacuously pass everything) and livelock/anti-cycling assumptions (so a formal engine can't satisfy the spec by endlessly re-triggering instead of completing), in addition to the functional properties above.

4. Worked Example — The Template Filled In

The template applied to the req/rdy/ack case, written as full English description — the level of detail that lets the checker be generated in one pass, with no clarification round needed.

4.1 Signals, Clock, and Reset

> **Signals:** req[0:1] (input, environment-driven), rdy[0:1] (module-local variable, set directly by the CPU — treat as environment/assume, not DUT obligation), ack[0:1] (output, DUT obligation).
>
> **Clock:** posedge clk.
>
> **Reset:** rst_n, active-low, asynchronous.

4.2 Timing

> **Timing:** for each lane i independently, req[i] rising must be followed by rdy[i] asserting within 1–2 cycles (inclusive). For each lane i independently, rdy[i] rising must be followed by ack[i] asserting within 1–3 cycles (inclusive).

4.3 Hold Behavior

> **Hold behavior:** req[i] and rdy[i] are level signals, not pulses — each holds high from assertion until its own ack[i] arrives, then may deassert.

4.4 Priority

> **Priority:** lane 0 gets a same-cycle tie-break over lane 1, not a full lockout — specifically, while req[0] is asserted and rdy[0] has not yet been granted, rdy[1] must not assert; once rdy[0] is granted, lane 1 is unrestricted. (Lane 1's own 1–2 cycle req→rdy bound still applies unconditionally outside of that contention window.)

4.5 Target

> **Target:** formal verification (not simulation-only) — include the fairness and livelock/anti-cycling companion assumptions needed for a formal engine, not just the functional properties.

5. The Claude Opus 5 high Session

 # Session Log — req/rdy/ack Priority Handshake FV Checker
****Framework:**** RAG-SVA (Ben Cohen & Kais Chibani, systemverilog.us)
****Session type:**** `/new` — SVA generation from a filled requirement template
****Deliverable:**** `req_rdy_ack_checker.sv`

Session Configuration

Resolved at CHECK 0 from the loaded framework corpus:

```
| Setting | Value | Source |
|---|---|---|
| FV MODE (Item 3) | YES | Pre_instructions_for_FV.md |
| DECOMP (Item 1) | ATOMIC | Pre_instructions_for_FV.md |
| GATE MODE | GATE_MODE_2 (ABBREVIATED) | user selection |
| VERBOSITY | `!C` (minimal) | Pre_instructions_for_FV.md |
| COV ITEMS | 9, 10 | Pre_instructions_for_FV.md |
| N | 2 | Pre_instructions_for_FV.md |
| MAX | 16 | Pre_instructions_for_FV.md |
```

Property count N > 10 → Two-Stage Workflow mandatory (Last_Minute_requests.md timeout/performance rule).

Active rule files: DoThisFirst.md (incl. merged CHECK 20, CHECK 21), rag_make.md, step0.md, Intention_Boundary_Property_Type_gold.docx, Last_Minute_requests.docx, Pre_instructions_for_FV.md, SVA_Framework_OnlySourceOfTruth.md.

Input Requirement

Provided as a pre-filled Section 4 worked example from **From Ambiguous Requirement to Verified Assertion** (Cohen & Chibani) — i.e. the requirement arrived already template-complete, which is what allowed generation in a single pass with only two decision gates.

****4.1 Signals, clock, reset****
- `req[0:1]` — input, environment-driven
- `rdy[0:1]` — module-local variable, set directly by the CPU; treat as

environment/assume, not DUT obligation
- `ack[0:1]` — output, DUT obligation
- Clock: `posedge clk`
- Reset: `rst_n`, active-low, asynchronous

4.2 Timing
- Per lane i independently: `req[i]` rising → `rdy[i]` asserts within 1–2 cycles (inclusive)
- Per lane i independently: `rdy[i]` rising → `ack[i]` asserts within 1–3 cycles (inclusive)

4.3 Hold behavior
- `req[i]` and `rdy[i]` are level signals, not pulses — each holds high from assertion until its own `ack[i]` arrives, then may deassert

4.4 Priority
- Lane 0 gets a same-cycle tie-break over lane 1, **not** a full lockout: while `req[0]` is asserted and `rdy[0]` not yet granted, `rdy[1]` must not assert; once `rdy[0]` is granted, lane 1 is unrestricted. Lane 1's own 1–2 cycle req→rdy bound applies unconditionally outside that contention window.

4.5 Target
- Formal verification (not simulation-only) — include fairness and livelock/anti-cycling companion assumptions, not just functional properties.

Phase Execution Log

Gate	Result
CHECK 0 (Settings Priority)	✅ passed
Phase 1 (RQM0 ambiguity)	✅ passed — no ambiguity; requirement was template-complete
Phase 2 (PORT-INTENT + INTENTION)	⚠️ HALT → resolved (see Decision 1)
Phase 3 (CHECK 1–21)	⚠️ HALT → resolved (see Decision 2)
Phase 4 (Code generation)	✅ silent
Phase 5 (Post-generation regex scans)	✅ silent
Phase 6 (Coverage generation)	✅ silent
Phase 7 (Documentation output)	✅ done

PORT-INTENT Table (Phase 2, always shown in GATE_MODE_2)

PORT-INTENT TABLE

PORT/PARAM	DIRECTION	TYPE	NAME	INTENTION	ACTIVITY
PORT	input	logic	clk	N/A	clock
PORT	input	logic	rst_n	N/A	reset
PORT	input	logic [0:1]	req	assume	trigger
LOCAL	N/A	logic [0:1]	rdy	assume	constraint
PORT	output	logic [0:1]	ack	obligation	response

`rdy` classified INTENTION=assume: it is a module-local variable, but §4.1 states the CPU sets it directly — the source is environmental, not derived from any DUT obligation signal. Tagged `// AUX: INPUT-DRIVEN` per Phase 2 Step 0C, which permits use in `assume`.

Consequence for property placement:
- `req→rdy` is an **assume** (both signals assume-classified)
- `rdy→ack` is the **assert** (`ack` is the sole obligation signal)
- The priority rule constrains `rdy[1]` — assume-side

Three-Axis Classification (Phase 2, FV mode)

Property group	Safety/Liveness	Strong/Weak	Bounded/Unbounded
`ap_rdy0_to_ack0`, `ap_rdy1_to_ack1`	Safety	`strong()`	Bounded [1:3]
`mp_*` functional / hold / priority	Safety	weak (strong banned in assume)	Bounded
`mp_fair_*`, `mp_livelock_*`	Liveness	`s_eventually`	Unbounded

Per FV-SW-2, the bounded `##[1:3]` obligations with "must be followed by" phrasing classify as SAFETY, not LIVENESS-B. No FV-FA/FV-LL requirement therefore arises from the assertions themselves — the fairness and livelock assumptions in this module come solely from §4.5's explicit direction.

Decision Gates

Decision 1 — CHECK 2 / F1-A: obligation signal in `mp_` antecedent

Raised at: Phase 2

The §4.3 hold rule requires `ack[i]` — an obligation signal — in the antecedent of four assume properties:

```systemverilog
mp_req0_holds_until_ack:  (req[0] && !ack[0]) |-> ##1 req[0]
mp_req1_holds_until_ack:  (req[1] && !ack[1]) |-> ##1 req[1]
mp_rdy0_holds_until_ack:  (rdy[0] && !ack[0]) |-> ##1 rdy[0]
mp_rdy1_holds_until_ack:  (rdy[1] && !ack[1]) |-> ##1 rdy[1]
```

F1-A permits an obligation signal in an `mp_*` antecedent **only** as a pure release trigger, never as something being constrained. Here `ack[i]` gates when the hold ends and is not itself constrained — satisfying F1-A's three requirements — but the rule mandates explicit written authorization rather than inference.

User response: Y — F1-A exception confirmed for all four hold assumptions. Recorded in the generated file header and in each property's DOC9 NOTE field.

Decision 2 — CHECK FV-FA-3: fairness/livelock in a pure-Safety property set

Raised at: Phase 3

FV-FA-3 forbids `mp_fair_*` and `mp_livelock_*` in property sets that are purely Safety-class. Every assertion in this set is Safety (bounded

`##[1:3]`, `strong()` applied). §4.5 explicitly requires fairness and livelock companions. Both readings were defensible:

- **Option A** — honor §4.5 as written; generate all 6 fairness/livelock assumptions, documenting the FV-FA-3 deviation as user-directed. Rationale: without activation fairness a formal engine can hold `req[i]` low forever and every property passes vacuously — a real concern even for Safety-class properties.
- **Option B** — honor FV-FA-3; omit fairness/livelock, generate the 9 functional properties plus covers. The vacuity concern would instead be addressed by the Item 9/10 reachability covers, which fail visibly if the antecedent is never exercised.

User response: A — fairness and livelock generated; FV-FA-3 deviation recorded in the file header as user-directed, not inferred.

Generated Property Inventory (N = 22 properties, 15 covers)

Assumptions — functional (3)

Label	Requirement source
`mp_req0_to_rdy0`	§4.2 lane 0 req→rdy, 1–2 cycles
`mp_req1_to_rdy1`	§4.2 lane 1 req→rdy, 1–2 cycles
`mp_rdy_priority`	§4.4 same-cycle tie-break (not lockout)

Assumptions — hold-until-ack, F1-A confirmed (4)

Label	Requirement source
`mp_req0_holds_until_ack`	§4.3
`mp_req1_holds_until_ack`	§4.3
`mp_rdy0_holds_until_ack`	§4.3
`mp_rdy1_holds_until_ack`	§4.3

Assertions — obligations (2)

Label	Class	Requirement source
`ap_rdy0_to_ack0`	SAFETY	§4.2 lane 0 rdy→ack, 1–3 cycles
`ap_rdy1_to_ack1`	SAFETY	§4.2 lane 1 rdy→ack, 1–3 cycles

Assumptions — fairness (4) and livelock (2), per §4.5

Label	TYPE	Rationale
`mp_fair_req0`	Fairness/Activation	L3-B — req[0] low forever ⇒ vacuous pass
`mp_fair_req1`	Fairness/Activation	L3-B — same, lane 1
`mp_fair_rdy0`	Fairness/Blocking-Signal	L3-A — gates both lanes via `mp_rdy_priority`
`mp_fair_rdy1`	Fairness/Blocking-Signal	L3-A — stalls lane 1 rdy→ack
`mp_livelock_req0`	Fairness/Blocking-Signal (FV-LL-3)	FV-LL-6 hold-until-completion
`mp_livelock_req1`	Fairness/Blocking-Signal (FV-LL-3)	FV-LL-6 hold-until-completion

Coverage (15)

- Item 9 (antecedent reachability): `cp_rdy0_to_ack0_trigger`, `cp_rdy1_to_ack1_trigger`
- Item 10 (full sequence): `cp_rdy0_to_ack0_full`, `cp_rdy1_to_ack1_full`
- UR-5 (one `cp_*_hit` per `mp_*`): `cp_req0_to_rdy0_hit`,

`cp_req1_to_rdy1_hit`, `cp_rdy_priority_hit`,
`cp_req0_holds_until_ack_hit`, `cp_req1_holds_until_ack_hit`,
`cp_rdy0_holds_until_ack_hit`, `cp_rdy1_holds_until_ack_hit`
- Fairness hits: `cp_fair_req0_hit`, `cp_fair_req1_hit`, `cp_fair_rdy0_hit`,
`cp_fair_rdy1_hit`
- Livelock + FV-LL-6 diagnostics: `cp_livelock_req0_hit`, `cp_new_req0_hit`,
`cp_livelock_req1_hit`, `cp_new_req1_hit`

Rules Applied — Notable Cases

****CHECK 21 (let vs. assign)**** — the FV-LL-6 auxiliary signals were declared with `let`, not `logic` + `assign`:

```systemverilog
let new_req0 = $rose(req[0]) || (req[0] && $past(ack[0], 1));
let new_req1 = $rose(req[1]) || (req[1] && $past(ack[1], 1));
```

These are assertion-only auxiliaries, so `let` avoids an unnecessary synthesizable net and preserves correct assertion-sampling semantics.

****FV-LL-6 hold-until-completion protocol**** — because §4.3 makes `req[i]` a held level rather than a pulse, `s_eventually always !req[i]` would contradict the required hold. The `new_req[i]` auxiliary captures only a genuinely new transaction (first rising edge, or re-request immediately after its own ack), and livelock is applied to that instead.

****step0.md §C3-STRONG**** — `strong()` is banned in `assume property`. The `req→rdy` assumptions use weak `##[1:2]`; `strong()` appears only on the assert side.

****Cross-lane index asymmetry**** — `mp_rdy_priority` intentionally references both `req[0]` and `rdy[1]`. The index-symmetry guidance in Last_Minute_requests.md targets accidental index mismatch inside a per-lane template, not a deliberate inter-lane arbitration property. Flagged explicitly in the property's DOC9 block rather than silently allowed.

Open Items / Watch List

- ****`mp_fair_rdy0` is RISK LEVEL: Medium**** — it gates both lanes through `mp_rdy_priority`. If lane 1 coverage returns unexpectedly low at proof time, this assumption is the first place to look.
- ****FV-LL-6 diagnostics**** — if `cp_new_req0_hit` or `cp_new_req1_hit` proves UNREACHABLE, the `new_req[i]` definition is over-restrictive; loosen it to `$rose(req[i])` alone before re-adding the `$past(ack[i],1)` term.
- ****Bind placeholder**** — `YOUR_DUT_MODULE_NAME` must be replaced with the actual DUT module name, and the port connections adjusted if the DUT's internal signals are not literally named `req`/`rdy`/`ack`.

Deliverable

`req_rdy_ack_checker.sv` — complete checker module plus bind statement.
Every property traces to a numbered section of the input requirement.

6. The Generated Checker Module

The complete SystemVerilog module below is the direct product of the worked example in Section 4 — every property traces back to one line above. File: `req_rdy_ack_checker.sv`.

```
 // ============================================================================
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// ============================================================================
// req_rdy_ack_checker
//
// Priority req/rdy/ack handshake checker (RAG-SVA framework, Ben Cohen &
// Kais Chibani, systemverilog.us).
//
// PORT-INTENT TABLE
// | PORT/PARAM | DIRECTION | TYPE        | NAME  | INTENTION  | ACTIVITY   |
// |------------|-----------|-------------|-------|------------|------------|
// | PORT       | input     | logic       | clk   | N/A        | clock      |
// | PORT       | input     | logic       | rst_n | N/A        | reset      |
// | PORT       | input     | logic [0:1] | req   | assume     | trigger    |
// | LOCAL      | N/A       | logic [0:1] | rdy   | assume     | constraint |
// | PORT       | output    | logic [0:1] | ack   | obligation | response   |
//
// rdy is a module-local variable set directly by the CPU (Section 4.1), so it
// is environment-driven: INTENTION=assume, tagged AUX: INPUT-DRIVEN per Phase 2
// Step 0C. ack is the sole obligation signal, so req->rdy is an assume property
// and rdy->ack is the assert property.
//
// THREE-AXIS CLASSIFICATION
//   ap_rdy{0,1}_to_ack{0,1} : SAFETY / strong() / bounded [1:3]
//   mp_* functional, hold, priority : SAFETY / weak / bounded
//   mp_*_fair, mp_livelock_* : LIVENESS / s_eventually / unbounded
//
// CHECK 0: Item 3 (Formal Verification Mode) SETTING = YES.
//
// DEVIATION NOTE (CHECK FV-FA-3): every assertion in this module is
// Safety-class (bounded ##[1:3]), so FV-FA-3 would normally forbid fairness
// and livelock assumptions here. Section 4.5 of the requirement explicitly
// directs that they be included; user confirmed option A. Recorded as a
// user-directed deviation, not an inferred one.
//
// F1-A: ack[i] (obligation) appears in the antecedent of the four
// hold-until-ack assumptions as a pure release trigger only — never
// constrained. User confirmed F1-A for all four (Y).
//
// NOTE: rdy is visible by name inside the bound module's scope (bind splices
// this checker into that scope) — it need not be a port on the DUT itself.
// ============================================================================
module req_rdy_ack_checker (
  input logic       clk,
  input logic       rst_n,
  input logic [0:1] req,
  input logic [0:1] rdy,
  input logic [0:1] ack
);
```

```
// ------------------------------------------------------------------
// ASSUMPTIONS (mp_*) — req and rdy are both INTENTION=assume
// strong() is BANNED in assume property (step0.md §C3-STRONG); the weak
// ##[1:2] form is correct — an assumption constrains the environment, it
// does not force the sequence to complete.
// ------------------------------------------------------------------

/*
ASSUMPTION:    req[0] rising is followed by rdy[0] within 1-2 cycles.
TYPE:          Input Constraint
JUSTIFICATION: Section 4.2 — per-lane req->rdy bound, CPU-driven rdy.
RISK LEVEL:    Low
*/
property p_req0_to_rdy0;
  @(posedge clk) disable iff (!rst_n)
    $rose(req[0]) |-> ##[1:2] rdy[0];
endproperty
mp_req0_to_rdy0: assume property (p_req0_to_rdy0);

/*
ASSUMPTION:    req[1] rising is followed by rdy[1] within 1-2 cycles.
TYPE:          Input Constraint
JUSTIFICATION: Section 4.2 — same bound as lane 0, applied to lane 1.
               Section 4.4 states this bound holds unconditionally outside
               the contention window.
RISK LEVEL:    Low
*/
property p_req1_to_rdy1;
  @(posedge clk) disable iff (!rst_n)
    $rose(req[1]) |-> ##[1:2] rdy[1];
endproperty
mp_req1_to_rdy1: assume property (p_req1_to_rdy1);

/*
ASSUMPTION:    While req[0] is asserted and rdy[0] not yet granted,
               rdy[1] must not assert.
TYPE:          Input Constraint / Arbitration
JUSTIFICATION: Section 4.4 — lane 0 gets a same-cycle tie-break, NOT a full
               lockout. Once rdy[0] is granted, lane 1 is unrestricted.
RISK LEVEL:    Medium — cross-lane by design. The index-symmetry rule
               (Last_Minute_requests.md) targets accidental index mismatch
               inside a per-lane template, not an intentional inter-lane
               arbitration property. Flagged explicitly.
*/
property p_rdy_priority;
  @(posedge clk) disable iff (!rst_n)
    (req[0] && !rdy[0]) |-> !rdy[1];
endproperty
mp_rdy_priority: assume property (p_rdy_priority);

// ------------------------------------------------------------------
// F1-A HOLD-UNTIL-RELEASE — req[i]/rdy[i] hold high until ack[i]
// (Section 4.3). ack[i] is an OBLIGATION signal in the antecedent;
// F1-A permits this only as a pure release trigger. User confirmed.
// ------------------------------------------------------------------

/*
ASSUMPTION:    req[0] holds high while ack[0] is absent.
TYPE:          Input Constraint / Hold Stability
JUSTIFICATION: Section 4.3 — req is a level signal held until its own ack.
```

```
  RISK LEVEL:    Low
  NOTE:          ack[0] (obligation) in antecedent per F1-A — NOT constrained.
  */
  property p_req0_holds_until_ack;
    @(posedge clk) disable iff (!rst_n)
      (req[0] && !ack[0]) |-> ##1 req[0];
  endproperty
  mp_req0_holds_until_ack: assume property (p_req0_holds_until_ack);

  /*
  ASSUMPTION:    req[1] holds high while ack[1] is absent.
  TYPE:          Input Constraint / Hold Stability
  JUSTIFICATION: Section 4.3, applied to lane 1.
  RISK LEVEL:    Low
  NOTE:          ack[1] (obligation) in antecedent per F1-A — NOT constrained.
  */
  property p_req1_holds_until_ack;
    @(posedge clk) disable iff (!rst_n)
      (req[1] && !ack[1]) |-> ##1 req[1];
  endproperty
  mp_req1_holds_until_ack: assume property (p_req1_holds_until_ack);

  /*
  ASSUMPTION:    rdy[0] holds high while ack[0] is absent.
  TYPE:          Input Constraint / Hold Stability
  JUSTIFICATION: Section 4.3 — hold applies to rdy as well as req. This is
                 also what keeps mp_rdy_priority's block on rdy[1] coherent.
  RISK LEVEL:    Low
  NOTE:          ack[0] (obligation) in antecedent per F1-A — NOT constrained.
  */
  property p_rdy0_holds_until_ack;
    @(posedge clk) disable iff (!rst_n)
      (rdy[0] && !ack[0]) |-> ##1 rdy[0];
  endproperty
  mp_rdy0_holds_until_ack: assume property (p_rdy0_holds_until_ack);

  /*
  ASSUMPTION:    rdy[1] holds high while ack[1] is absent.
  TYPE:          Input Constraint / Hold Stability
  JUSTIFICATION: Section 4.3, applied to lane 1.
  RISK LEVEL:    Low
  NOTE:          ack[1] (obligation) in antecedent per F1-A — NOT constrained.
  */
  property p_rdy1_holds_until_ack;
    @(posedge clk) disable iff (!rst_n)
      (rdy[1] && !ack[1]) |-> ##1 rdy[1];
  endproperty
  mp_rdy1_holds_until_ack: assume property (p_rdy1_holds_until_ack);

  // ---------------------------------------------------------------------
  // OBLIGATIONS (ap_*) — ack is INTENTION=obligation (DUT output).
  // SAFETY class, bounded [1:3]; strong() applied (assert side only).
  // ---------------------------------------------------------------------
  // CLASS: SAFETY
  property p_rdy0_to_ack0;
    @(posedge clk) disable iff (!rst_n)
      $rose(rdy[0]) |-> strong(##[1:3] ack[0]);
  endproperty
  ap_rdy0_to_ack0: assert property (p_rdy0_to_ack0)
    else $error("%m ap_rdy0_to_ack0 FAILED at time=%0t: ack[%0d] not seen within 3 cycles of rdy
rise",  $realtime, 0);
```

```
  // CLASS: SAFETY
  property p_rdy1_to_ack1;
    @(posedge clk) disable iff (!rst_n)
      $rose(rdy[1]) |-> strong(##[1:3] ack[1]);
  endproperty
  ap_rdy1_to_ack1: assert property (p_rdy1_to_ack1)
    else $error("%m ap_rdy1_to_ack1 FAILED at time=%0t: ack[%0d] not seen within 3 cycles of rdy
rise", $realtime, 1);

  // ------------------------------------------------------------------
  // COVERAGE (cp_*) — Item 9 (antecedent reachability) + Item 10 (full
  // sequence) for every ap_*; UR-5 cp_*_hit for every mp_*.
  // ------------------------------------------------------------------

  // Item 9 — antecedent reachability
  cp_rdy0_to_ack0_trigger: cover property (@(posedge clk) disable iff (!rst_n)
    $rose(rdy[0]));
  cp_rdy1_to_ack1_trigger: cover property (@(posedge clk) disable iff (!rst_n)
    $rose(rdy[1]));

  // Item 10 — full sequence execution
  cp_rdy0_to_ack0_full: cover property (@(posedge clk) disable iff (!rst_n)
    $rose(rdy[0]) ##[1:3] ack[0]);
  cp_rdy1_to_ack1_full: cover property (@(posedge clk) disable iff (!rst_n)
    $rose(rdy[1]) ##[1:3] ack[1]);

  // UR-5 — one cp_*_hit per mp_*
  cp_req0_to_rdy0_hit: cover property (@(posedge clk) disable iff (!rst_n)
    $rose(req[0]) ##[1:2] rdy[0]);
  cp_req1_to_rdy1_hit: cover property (@(posedge clk) disable iff (!rst_n)
    $rose(req[1]) ##[1:2] rdy[1]);
  cp_rdy_priority_hit: cover property (@(posedge clk) disable iff (!rst_n)
    req[0] && !rdy[0] && req[1]);   // contention actually exercised
  cp_req0_holds_until_ack_hit: cover property (@(posedge clk) disable iff (!rst_n)
    req[0] && !ack[0]);
  cp_req1_holds_until_ack_hit: cover property (@(posedge clk) disable iff (!rst_n)
    req[1] && !ack[1]);
  cp_rdy0_holds_until_ack_hit: cover property (@(posedge clk) disable iff (!rst_n)
    rdy[0] && !ack[0]);
  cp_rdy1_holds_until_ack_hit: cover property (@(posedge clk) disable iff (!rst_n)
    rdy[1] && !ack[1]);

  // ------------------------------------------------------------------
  // FAIRNESS / LIVELOCK (FV-FA L3-A/L3-B, FV-LL-3, FV-LL-6)
  // Included per Section 4.5 (user-directed; see FV-FA-3 deviation note).
  // ------------------------------------------------------------------

  /*
  ASSUMPTION:    req[0] eventually asserts.
  TYPE:          Fairness/Activation
  JUSTIFICATION: L3-B — req[0] held low forever makes every downstream
                 property vacuously true. Section 4.5 requires fairness.
  RISK LEVEL:    Low
  */
  mp_fair_req0: assume property (@(posedge clk) disable iff (!rst_n)
    s_eventually req[0]);
  cp_fair_req0_hit: cover property (@(posedge clk) disable iff (!rst_n) req[0]);
```

```
/*
ASSUMPTION:    req[1] eventually asserts.
TYPE:          Fairness/Activation
JUSTIFICATION: L3-B — same as lane 0.
RISK LEVEL:    Low
*/
mp_fair_req1: assume property (@(posedge clk) disable iff (!rst_n)
  s_eventually req[1]);
cp_fair_req1_hit: cover property (@(posedge clk) disable iff (!rst_n) req[1]);

/*
ASSUMPTION:    rdy[0] eventually asserts.
TYPE:          Fairness/Blocking-Signal
JUSTIFICATION: L3-A — rdy[0] held low forever stalls the rdy->ack
               obligation AND permanently blocks rdy[1] via mp_rdy_priority.
RISK LEVEL:    Medium — this signal gates both lanes. HUMAN REVIEW
               RECOMMENDED if lane 1 coverage is unexpectedly low.
*/
mp_fair_rdy0: assume property (@(posedge clk) disable iff (!rst_n)
  s_eventually rdy[0]);
cp_fair_rdy0_hit: cover property (@(posedge clk) disable iff (!rst_n) rdy[0]);

/*
ASSUMPTION:    rdy[1] eventually asserts.
TYPE:          Fairness/Blocking-Signal
JUSTIFICATION: L3-A — rdy[1] held low forever stalls the lane 1
               rdy->ack obligation.
RISK LEVEL:    Low
*/
mp_fair_rdy1: assume property (@(posedge clk) disable iff (!rst_n)
  s_eventually rdy[1]);
cp_fair_rdy1_hit: cover property (@(posedge clk) disable iff (!rst_n) rdy[1]);

// FV-LL-6 HOLD-UNTIL-COMPLETION PROTOCOL — req[i] holds until ack[i]
// (Section 4.3), so `s_eventually always !req[i]` would contradict the
// required hold. Auxiliary new_req[i] captures only a genuinely NEW
// transaction: a first rising edge, or a re-request right after its own
// ack completes. Declared with `let` per CHECK 21 (assertion-only
// auxiliary — no synthesizable net, correct sampling semantics).
let new_req0 = $rose(req[0]) || (req[0] && $past(ack[0], 1));
let new_req1 = $rose(req[1]) || (req[1] && $past(ack[1], 1));

/*
ASSUMPTION:    No new lane 0 transactions are initiated indefinitely,
               allowing an in-flight transaction to complete.
TYPE:          Fairness/Blocking-Signal (FV-LL-3)
JUSTIFICATION: Tool can re-trigger req[0] indefinitely; new_req0 prevents
               that without forbidding the Section 4.3 hold behavior.
RISK LEVEL:    Low
*/
mp_livelock_req0: assume property (@(posedge clk) disable iff (!rst_n)
  s_eventually always !new_req0);
cp_livelock_req0_hit: cover property (@(posedge clk) disable iff (!rst_n) !new_req0);
cp_new_req0_hit:      cover property (@(posedge clk) disable iff (!rst_n) new_req0);
// cp_new_req0_hit is the FV-LL-6 reachability diagnostic. If UNREACHABLE,
// loosen new_req0: try `$rose(req[0])` alone before adding $past(ack[0],1).
```

```
 /*
  ASSUMPTION:    No new lane 1 transactions are initiated indefinitely.
  TYPE:          Fairness/Blocking-Signal (FV-LL-3)
  JUSTIFICATION: Same as lane 0.
  RISK LEVEL:    Low
  */
  mp_livelock_req1: assume property (@(posedge clk) disable iff (!rst_n)
    s_eventually always !new_req1);
  cp_livelock_req1_hit: cover property (@(posedge clk) disable iff (!rst_n) !new_req1);
  cp_new_req1_hit:      cover property (@(posedge clk) disable iff (!rst_n) new_req1);
  // cp_new_req1_hit is the FV-LL-6 reachability diagnostic (see lane 0 note).

endmodule : req_rdy_ack_checker

// ============================================================================
// BIND
//
// PLACEHOLDER: replace YOUR_DUT_MODULE_NAME with the actual DUT module name.
// Binding into the DUT scope makes rdy — an internal local variable — visible
// by name without needing to be a DUT port. If the internal signals are not
// literally named req/rdy/ack, adjust the right-hand side of each connection.
// ============================================================================
bind YOUR_DUT_MODULE_NAME req_rdy_ack_checker u_req_rdy_ack_checker (
  .clk   (clk),
  .rst_n (rst_n),
  .req   (req),
  .rdy   (rdy),
  .ack   (ack)
);
```

7. Timing Diagrams

1. Request

```
> "back to my request for module req_rdy_ack_checker
> give me a timing diagram for the cases when req[0] and req[1] occur at the
> same and at different times"

Diagrams were rendered in the session default style (Option 3, T-column
edge-annotation, per Last_Minute_requests.md).

---
```

2. Case A — simultaneous rise, `req[0]` and `req[1]` at T0

```
                T0     T1     T2     T3     T4     T5
clk              ↑      ↑      ↑      ↑      ↑      ↑
─────────────────────────────────────────────────────────
req[0]          0→1     1      1      1     1→0     0
req[1]          0→1     1      1      1      1     1→0
rdy[0]           0     0→1     1      1     1→0     0
rdy[1]           0      0     0→1     1      1     1→0
ack[0]           0      0      0     0→1    1→0     0
ack[1]           0      0      0      0     0→1    1→0
─────────────────────────────────────────────────────────
                ↑req0  ↑rdy0  ↑rdy1  ↑ack0  ↑ack1
                ↑req1
```

- **T0** — both lanes rise. `mp_rdy_priority` is active (`req[0]=1, rdy[0]=0`), so `rdy[1]` is held off. The tie-break does its job.
- **T1** — `rdy[0]` granted 1 cycle after `$rose(req[0])`, inside `##[1:2]`. The priority antecedent goes false, releasing lane 1.
- **T2** — `rdy[1]` granted 2 cycles after `$rose(req[1])` — still inside `##[1:2]`.
- **T3/T4** — acks land 2 cycles after their respective `rdy` rises, inside `##[1:3]`. Each lane's `req`/`rdy` releases the cycle after its own ack.

Tight-case note. If the CPU took the full 2 cycles to grant `rdy[0]` (T2 instead of T1), `rdy[1]` would be blocked at T1 and forced to exactly T2 — the last legal cycle in its own `##[1:2]` window. The priority constraint and the lane-1 timing bound only just coexist.

---

3. Case B1 — `req[0]` first at T0, `req[1]` at T2

```
                T0     T1     T2     T3     T4     T5     T6
clk              ↑      ↑      ↑      ↑      ↑      ↑      ↑
──────────────────────────────────────────────────────────────
req[0]          0→1     1      1      1     1→0     0      0
req[1]           0      0     0→1     1      1      1     1→0
rdy[0]           0     0→1     1      1     1→0     0      0
rdy[1]           0      0      0     0→1     1      1     1→0
ack[0]           0      0      0     0→1    1→0     0      0
ack[1]           0      0      0      0      0     0→1    1→0
──────────────────────────────────────────────────────────────
                ↑req0  ↑rdy0  ↑req1  ↑rdy1         ↑ack1
                              ↑ack0
```

Clean. Lane 0 is already granted when lane 1 arrives, so the priority antecedent is false from T1 onward and lane 1 runs unrestricted. This is the "not a full lockout" behavior §4.4 asked for.

---

4. Case B2 — `req[1]` first at T0, `req[0]` arrives mid-flight at T2

```
                T0     T1     T2
clk              ↑      ↑      ↑
──────────────────────────────────
req[1]          0→1     1      1
rdy[1]           0     0→1     ??   ← contradiction
req[0]           0      0     0→1
rdy[0]           0      0      0
ack[1]           0      0      0
──────────────────────────────────
                ↑req1  ↑rdy1  ↑req0
```

This case does not resolve. At T2 two assumptions demand opposite values for `rdy[1]`:

Assumption	Evaluated at	Demands
`mp_rdy1_holds_until_ack`	T1: `rdy[1]=1 && !ack[1]`	`rdy[1] = 1` at T2
`mp_rdy_priority`	T2: `req[0]=1 && !rdy[0]`	`rdy[1] = 0` at T2

Contradictory assumptions do not produce a failure — they produce an empty constraint set, and every property in that region passes **vacuously**. Precisely the failure mode the UR checks exist to catch, and it was not caught during generation.

Root cause — level-vs-edge flattening

§4.4 states that `rdy[1]` "must not **assert**" during the contention window. "Assert" means a new grant — a rising edge. The generated property encoded it as a **level** constraint, which also forbids an already-granted lane 1 from **holding** its grant. That contradicts both the §4.3 hold rule and the explicit "not a full lockout" wording in §4.4.

```systemverilog
  // WRONG (as generated) — forbids an in-flight rdy[1] from holding
  (req[0] && !rdy[0]) |-> !rdy[1];

  // CORRECT — forbids only a NEW grant during the contention window
  (req[0] && !rdy[0]) |-> !$rose(rdy[1]);
```

Why the fix is safe for the other cases

With `$rose`, Case B2 resolves: at T2 `rdy[1]` was already high at T1, so `$rose(rdy[1])` is false, the constraint is satisfied, and the hold still holds. Cases A and B1 are unaffected — in both, `rdy[1]` is low when the window is active, so blocking the edge and blocking the level are equivalent there.

---

5. Scope of the Defect

The same `!rdy[1]` form appears in the published checker in *From Ambiguous Requirement to Verified Assertion* (Cohen & Chibani), so this is not confined to the copy generated in this session — the paper's Section 5 listing carries it too.

Observations on why the checks did not catch it

- The generation-time review classified `mp_rdy_priority` as Medium risk, but for the wrong reason — cross-lane index asymmetry (a naming/index concern), not assumption satisfiability.
- **No check in the corpus currently tests for **mutually contradictory assumptions** across separate `mp_*` properties. UR-1 through UR-5 test each assumption in isolation (implication form, standalone sequences, edge triggers, cover pairing); none tests whether two individually-valid assumptions can both hold in some reachable state.**
- The contradiction is order-dependent — it only manifests when the lower-priority lane is granted *before* the higher-priority request arrives. Cases A and B1, the two orderings a reviewer reaches for first, both look correct.

---

6. Open Items

- **Regenerate `req_rdy_ack_checker.sv`** with the corrected `!$rose(rdy[1])` form. Offered; not yet performed.
- **Add a cover for the B2 ordering** specifically — something along the lines of `rdy[1] && !ack[1] && $rose(req[0])` — so the contention case that exposed this is exercised rather than assumed reachable.
- **Consider a new check** for cross-assumption satisfiability. The existing UR family is per-assumption; this defect needed a pairwise analysis. Would sit alongside CHECK 20 / CHECK 21 as a corpus addition.
- **Correct the paper's Section 5 listing** if the checker there is intended to be reference-quality.

---

7. Files Referenced

File	Role
`req_rdy_ack_checker.sv`	Checker under review — contains the defect
`Requirement_Template_Paper.pdf`	§4 worked example the checker was generated from; §5 listing carries the same defect
`DoThisFirst.md`	UR-1..UR-5, CHECK 20, CHECK 21
`Last_Minute_requests.md`	Option 3 T-column diagram style

---

8. Mutually Contradictory Assumption Analysis

Framework: RAG-SVA (Ben Cohen & Kais Chibani, systemverilog.us)
Subject: `req_rdy_ack_checker.sv` — two-lane priority handshake checker
Analysis type: pairwise assumption satisfiability across all 13 `mp_*` properties — a check the UR family does not perform, since UR-1 through UR-5 test each assumption in isolation.
Status: one hard contradiction confirmed, one over-constraint requiring a requirement decision (HALT), one degenerate-trace risk flagged.

---

1. Request

> "For these requirements and assertions, check for mutually contradictory
> assumptions"

---

2. The Assumption Set Under Analysis

	Label	Constraint
1	`mp_req0_to_rdy0`	`$rose(req[0]) |-> ##[1:2] rdy[0]`
2	`mp_req1_to_rdy1`	`$rose(req[1]) |-> ##[1:2] rdy[1]`
3	`mp_rdy_priority`	`(req[0] && !rdy[0]) |-> !rdy[1]`
4	`mp_req0_holds_until_ack`	`(req[0] && !ack[0]) |-> ##1 req[0]`
5	`mp_req1_holds_until_ack`	`(req[1] && !ack[1]) |-> ##1 req[1]`
6	`mp_rdy0_holds_until_ack`	`(rdy[0] && !ack[0]) |-> ##1 rdy[0]`
7	`mp_rdy1_holds_until_ack`	`(rdy[1] && !ack[1]) |-> ##1 rdy[1]`
8–11	`mp_fair_req0/1`, `mp_fair_rdy0/1`	`s_eventually sig`
12–13	`mp_livelock_req0/1`	`s_eventually always !new_req[i]`

---

3. C1 — HARD CONTRADICTION: #3 vs #7

Severity: blocking. Produces an empty constraint set.

Minimal trigger condition:

- At cycle **T**: `rdy[1]=1`, `ack[1]=0` → **#7** forces `rdy[1]=1` at T+1
- At cycle **T+1**: `req[0]=1`, `rdy[0]=0` → **#3** forces `rdy[1]=0` at T+1

Two assumptions demand opposite values for the same signal at the same cycle. Contradictory assumptions do not fail — they yield an empty constraint set, and every property in that region passes **vacuously**.

3.1 The recurrence is the real problem

This is worse than the earlier Case-B2 walkthrough suggested. **#3's antecedent is `req[0]` as a level, not `$rose(req[0])`.** After lane 0 completes a transaction and re-requests, the block re-engages.

So this is not a rare startup ordering. It fires on **every lane-0 request that arrives while lane 1 has an outstanding grant** — and under the fairness assumptions (#8–#11), that is guaranteed to happen infinitely often.

3.2 Root cause — level-vs-edge flattening

§4.4 says `rdy[1]` "must not **assert**" during the contention window. "Assert" means a new grant — a rising edge. The generated property encoded it as a **level** constraint, which also forbids an already-granted lane 1 from **holding** its grant, contradicting both the §4.3 hold rule and §4.4's own "not a full lockout" wording.

```systemverilog
  // WRONG (as generated) — forbids an in-flight rdy[1] from holding
  (req[0] && !rdy[0]) |-> !rdy[1];

  // CORRECT — forbids only a NEW grant during the contention window
  (req[0] && !rdy[0]) |-> !$rose(rdy[1]);
```

Resolution: mechanical. No requirement input needed.

---

4. C2 — OVER-CONSTRAINT: #3 vs #2 vs #1

Severity: silent loss of verification coverage. Not empty, but the satisfying set collapses to a single trace.

When `req[1]` precedes `req[0]`:

```
T0: $rose(req[1])            → #2 requires rdy[1] at T1 or T2
T1: $rose(req[0]), rdy[0]=0  → #3 blocks rdy[1] at T1
T2: if rdy[0] still 0        → #3 blocks rdy[1] at T2 as well
```

#1 permits `rdy[0]` at T2 **or** T3. But if the environment picks T3, `rdy[1]` is blocked at both T1 and T2, and **#2** becomes unsatisfiable.

The environment is therefore silently forced to grant `rdy[0]` at exactly T2 — `##[1:2]` is effectively narrowed to `##[1:1]` in every such scenario.

Consequence: the formal engine will never explore the slower-grant case. Any bug that only manifests when lane 0 takes its full 2 cycles goes unexamined, and nothing reports this.

4.1 The C1 fix does NOT resolve C2

Changing #3 to `!$rose(rdy[1])` still prevents `rdy[1]` from *rising* at T1 and T2. The over-constraint survives intact. These are independent defects.

4.2 This is an RQM0 ambiguity, not a coding bug — HALT

§4.4 states that lane 1's bound "still applies unconditionally outside of that contention window," implying it does *not* apply inside. But the requirement never says what happens to a lane-1 request that **arrives during** contention. Two defensible readings:

Reading	Meaning
(A)	The 1–2 cycle bound is suspended during contention and measured from when the window clears
(B)	The bound holds absolutely, and lane 0 is obligated to grant early enough not to violate it

These produce different checkers. Per halt-don't-guess, this requires a user decision before #2 or #3 can be finalized. **Not resolved in this session.**

---

5. C3 — DEGENERATE-TRACE RISK: fairness + livelock + holds

Severity: advisory. Not a contradiction.

`assume property (@(posedge clk) s_eventually req[0])` is evaluated at **every** cycle, so it means "infinitely often," not "at least once."

Combined with #12 (`s_eventually always !new_req0`) and the hold assumptions, the traces that satisfy all three tend toward everything stuck high: `req[0]` rises once and never deasserts, `rdy[0]` likewise, `ack[0]` pulses once. That satisfies every assumption while exercising almost nothing.

Note on the FV-FA-3 deviation. These fairness assumptions were added under the user-authorized FV-FA-3 deviation specifically to *prevent* vacuity. In combination they may be narrowing the explored state space instead — a cost that was not visible at generation time.

Diagnostic: if `cp_new_req0_hit` and `cp_new_req1_hit` come back reachable but with very few distinct traces, this is what is happening.

---

6. Pairs Checked Clean

No conflicts found in:

- #4 / #5 against #1 / #2 — hold assumptions vs. req→rdy timing
- #6 against #3 — rdy[0] hold vs. priority
- #6 / #7 against #10 / #11 — rdy holds vs. rdy fairness
- #4 against #12 — req[0] hold vs. lane-0 livelock

```
---

## 7. Remediation Order

| Priority | Item | Action | Blocked on |
|---|---|---|---|
| 1 | **C1** | Apply `!$rose(rdy[1])` | Nothing — mechanical |
| 2 | **C2** | Choose reading (A) or (B), then regenerate #2 and #3 | **User decision** |
| 3 | **C3** | Decide whether fairness should be scoped (e.g. gated on a transaction-active condition) rather than free-running | User judgment |

Regeneration of the checker was offered pending the C2 answer; **not performed in this session.**

---

## 8. Methodological Observation

The UR family (UR-1 through UR-5) validates each assumption **in isolation** — implication form, standalone sequences, edge triggers, cover pairing. None of these detects a contradiction that exists only **between** two individually well-formed assumptions.

C1 was individually valid under every UR check and still produced an empty constraint set. C2 passed every check and silently removed a legal timing path from the state space.

This argues for a corpus addition covering cross-assumption satisfiability, alongside CHECK 20 and CHECK 21 — noted as an open item in the prior session log and reinforced here.

---

## 9. Files Referenced

| File | Role |
|---|---|
| `req_rdy_ack_checker.sv` | Checker under analysis — contains C1 and C2 |
| `Requirement_Template_Paper.pdf` | §4.4 priority wording; §5 listing carries the same C1 defect |
| `DoThisFirst.md` | UR-1..UR-5, CHECK 20, CHECK 21 |
| `Intention_Boundary_Property_Type_gold.docx` | §4.2 FV-FA-0/FV-FA-3 fairness scope |
```

9. Conclusions

The gap between Section 1 and Section 4 is the first point of this paper: the same protocol, the same author's intent, but one version leaves five things unstated and the other leaves nothing unstated. The template in Section 3 is what closes that gap on the next handshake-style requirement, before a single property gets written.

Sections 5 through 8 make a second point that matters more. A complete requirement is necessary for a correct checker; it is not sufficient. The checker generated from the fully-filled template still contained a contradiction. **mp_rdy_priority** encoded "rdy[1] must not assert" as a level constraint rather than an edge, which forbade an already-granted lane 1 from holding its grant and directly contradicted the hold rule in §4.3. Two assumptions demanded opposite values for the same signal on the same cycle. Nothing failed — contradictory assumptions do not fail. They empty the constraint set and let everything downstream pass vacuously.

That defect was not found by inspection, and no per-property check would have found it. UR-1 through UR-5 validate each assumption in isolation — implication form, standalone sequences, edge triggers, cover pairing — and **mp_rdy_priority** passed all of them. It was found by asking two specific questions of the generated code.

The first was a timing question: show the waveform for req[0] and req[1] rising simultaneously, and for each arrival order. The simultaneous case and the lane-0-first case both look correct. The lane-1-first case does not resolve — the contradiction appears only when the lower-priority lane already holds a grant at the moment the higher-priority request arrives. Walking the arrival orders is what surfaced it.

The second was an inconsistency question: check every assumption against every other for mutual satisfiability. That found the same contradiction independently, established that it recurs on every lane-0 request rather than only at startup, and surfaced a second finding the timing walk had missed — a silent over-constraint in which the priority rule and lane 1's own 1–2 cycle bound together force lane 0 to be granted at exactly one cycle, narrowing ##[1:2] to ##[1:1] and removing a legal timing path from the state space with no diagnostic of any kind.

This is the working conclusion. RAG-SVA's contribution is not that the first generation pass is correct; this paper documents a first pass that was not. It is that the framework makes the defect findable while it is still cheap. FV-ready assertions come from the combination of a complete requirement, an explicit timing analysis across arrival orders, and a cross-assumption inconsistency search. Any one of the three alone would have shipped this checker with a vacuity hole in it.

The over-constraint remains open at the time of writing, and deliberately so. Resolving it requires knowing whether lane 1's timing bound is suspended during contention or holds absolutely — a question the requirement does not answer, and one the framework refuses to guess at. Halting there is not an incomplete result. It is the discipline of Section 2 applied one level deeper: to the assumptions, not just the requirement.