Green Proofs, Hidden Bugs

What a Class Formal Verification of the CY62256N SRAM Teaches, Reviewed Under RAG-SVA, with Fixes and a Proposed Timed Model

Prof. Venkatesh Patil (Portland State University, venpatil@pdx.edu) · **Ben Cohen** (systemverilog.us)

November 2026

Abstract

In Fall 2026, the students of ECE-574 Assertion Based Verification at Portland State University formally verified a cycle model of the Cypress/Infineon CY62256N 32K x 8 SRAM with Synopsys VC Formal. They wrote assumptions, assertions and covers from the datasheet truth table, injected 44 bugs of their own (43 caught), and ran over-constraint experiments. The consolidated class report reaches one result above all: **every over-constraint left every assertion green**, and at least six reports showed a real bug turned into a proof. Only unreachable covers and vacuity flags revealed the problem. This paper reviews that report against the RAG-SVA rules (Cohen and Chibani, 2026), reproduces its main formal findings with open-source tools, re-runs its bug list in simulation against both the class checks and a RAG-SVA rewrite, and proposes a timed model in which the right assumptions differ from the class model. We address the use of `let` instead of `logic` + `assign` for assertion-only expressions, and show one class check, the shadow-model read check, that passed 1,367 times on a corrupted read path because its expected value was learned from the design itself.

1. Introduction

Assignment 3 of ECE-574 asked each student to formally verify a simplified, clocked model of the CY62256N SRAM, using the truth table on page 11 of the datasheet as the requirement. The professor's consolidated report (Patil, 2026) merges the student reports, A to L, into one reference set of SVA, results and lessons. It is unusual evidence: independent teams, one commercial formal tool (VC Formal V-2023.12-SP2-3), 44 injected bugs, and deliberate over-constraint experiments, with failures and setup errors reported honestly.

This paper does four things with that report:

- summarizes what the class did and found, including the formal analysis (Sections 2 and 3);
- examines that formal analysis and reproduces its central findings with open-source tools (Section 4);
- reviews the code against the RAG-SVA rules and shows the fixes, including `let` for assertion-only expressions and a reference model built only from inputs (Sections 5 and 6);
- re-runs the bug list against both checkers, and proposes a timed model with its own checker, from the authors' earlier datasheet-to-requirements work (Sections 7 and 8; Cohen and Chibani, 2026b).

2. The assignment and the class model

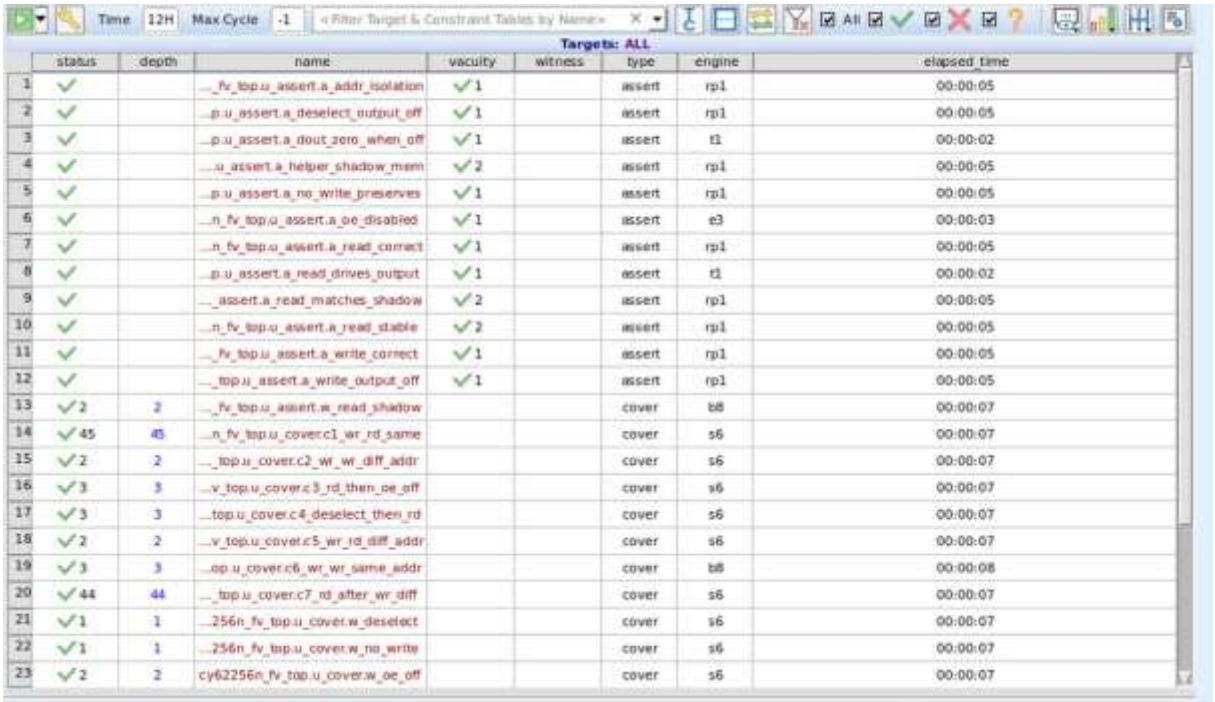
The model decodes the three active-low control pins into two requests. Every one of the eight pin combinations maps to exactly one datasheet mode, so no control combination is illegal:

```
assign write_req = !CE_n && !WE_n;           // write: CE=0, WE=0, OE=X
assign read_req  = !CE_n && WE_n && !OE_n;   // read:  CE=0, WE=1, OE=0
assign dout_oe   = read_req;
assign dout      = dout_oe ? mem[addr] : '0; // '0 stands for High-Z
```

Mode	CE_n WE_n OE_n	dout_oe / dout	Model behavior
Deselect	1 X X	0 / 0	no request; memory kept
Read	0 1 0	1 / mem[addr]	combinational, same cycle
Write	0 0 X	0 / 0	mem[addr] <= din at the next edge
Output disabled	0 1 1	0 / 0	OE_n blocks the read, WE_n the write

The model departs from the part in five ways, all stated in the report: a sampling clock (the part is asynchronous), a reset (the part has none; the array is cleared only if `INIT_ON_RESET = 1`), High-Z modeled as `dout_oe = 0` with `dout = 0`, the bidirectional bus split into `din` and `dout`, and no timing at all, with the address cut to 6 bits for fast proofs.

Data integrity uses a free symbolic address `sym_addr`, held constant, so a property proven for one word (`mem_sym`, a white-box tap of `mem[sym_addr]`) holds for every address. A shadow register records the last byte written to `sym_addr` for the black-box read check B3.


Time 12H Max Cycle -1 « Filter Target & Constraint Tables by Name »

Targets: ALL

	status	depth	name	vacuity	witness	type	engine	elapsed_time
1	✓		..._fv_top.u_assert.a_addr_isolation	✓1		assert	rp1	00:00:05
2	✓		...p.u_assert.a_deselect_output_off	✓1		assert	rp1	00:00:05
3	✓		...p.u_assert.a_dout_zero_when_off	✓1		assert	t1	00:00:02
4	✓		...u_assert.a_helper_shadow_mem	✓2		assert	rp1	00:00:05
5	✓		...p.u_assert.a_no_write_preserves	✓1		assert	rp1	00:00:05
6	✓		...n_fv_top.u_assert.a_oe_disabled	✓1		assert	e3	00:00:03
7	✓		...n_fv_top.u_assert.a_read_correct	✓1		assert	rp1	00:00:05
8	✓		...p.u_assert.a_read_drives_output	✓1		assert	t1	00:00:02
9	✓		..._assert.a_read_matches_shadow	✓2		assert	rp1	00:00:05
10	✓		...n_fv_top.u_assert.a_read_stable	✓2		assert	rp1	00:00:05
11	✓		..._fv_top.u_assert.a_write_correct	✓1		assert	rp1	00:00:05
12	✓		..._top.u_assert.a_write_output_off	✓1		assert	rp1	00:00:05
13	✓2	2	..._fv_top.u_assert.w_read_shadow			cover	b8	00:00:07
14	✓45	45	...n_fv_top.u_cover.c1_wr_rd_same			cover	s6	00:00:07
15	✓2	2	..._top.u_cover.c2_wr_wr_diff_addr			cover	s6	00:00:07
16	✓3	3	...v_top.u_cover.c3_rd_then_oe_off			cover	s6	00:00:07
17	✓3	3	...top.u_cover.c4_deselect_then_rd			cover	s6	00:00:07
18	✓2	2	...v_top.u_cover.c5_wr_rd_diff_addr			cover	s6	00:00:07
19	✓3	3	...op.u_cover.c6_wr_wr_same_addr			cover	b8	00:00:08
20	✓44	44	..._top.u_cover.c7_rd_after_wr_diff			cover	s6	00:00:07
21	✓1	1	...256n_fv_top.u_cover.w_deselect			cover	s6	00:00:07
22	✓1	1	...256n_fv_top.u_cover.w_no_write			cover	s6	00:00:07
23	✓2	2	cy62256n_fv_top.u_cover.w_oe_off			cover	s6	00:00:07

Figure 1. Baseline result from the class (Report A, VC Formal, AW = 6): all 12 assertions proven and non-vacuous, all covers and vacuity witnesses reached. Screenshot from the class report.

3. What the class found

Item	Class result (from the report)
Assumptions	2 needed: symbolic address constant (D0) and reset; known-inputs (D3) as documentation
Assertions	B1–B10, the kit example and an optional helper: 11 or 12 per report, all proven and non-vacuous
Covers	C1–C7 scenarios + 10 vacuity witnesses = 17, all reached at both widths
Runtime, full width	10–15 min typical; one B3 proof stalled at 66 min until a helper was added
Bugs injected	3 to 7 per report, 44 in total; 1 missed (reset clear removed)
Over-constraint	every report saw unreachable covers or vacuous assertions; at least 6 a real bug hidden
AI use disclosed	5 reports (4 Claude, 1 ChatGPT)

The ten assertions check the truth table from both sides. B4–B6 force the output off when deselected, during a write and when OE_n is high; B10 checks that a read does drive the output, which closes a gap the class identified: a dead output passes B4–B6. B1, B7 and B8 check storage through the white-box tap. B2 compares `dout` with the stored word; B3 compares it with the shadow, from the pins only. B9 checks that a held read is stable. Each assertion has a witness cover on its antecedent; a proven assertion whose witness is unreachable has passed vacuously.

The report's lesson list is worth quoting in substance: check covers before trusting a proof; black-box checks catch what a peek at storage cannot; output-off checks need an output-on check; true is not the same as inductive; most early failures are in the setup; and a missed bug points to an untested configuration.

4. The formal analysis in the report, examined

4.1 Proof status and vacuity

The class used two independent lines of non-vacuity evidence: a witness cover per assertion and VC Formal's own vacuity check (the vacuity column in Figure 1). That is stronger than most industrial practice. Two cautions from the report deserve emphasis. First, a cover reported at depth 45 is still covered; engines report the first trace found, not the shortest, so depth is not a measure of difficulty. Second, the GUI's flagged-goals count includes vacuous and unreachable goals, not only falsified ones (Report G). A sign-off script should report the three separately.

4.2 Over-constraint: green proofs, hidden bugs

Each over-constraint in Figure 2 is a plausible engineering habit: bus turnaround, address hold, an OE_n tied low on the board, a read-only use. Each removed legal stimulus. None produced a failing assertion.

Class over-constraint experiments: what each tool view showed

Over-constraint	Assertions	Covers unreachable / vacuous	Bug it hid
No read after write	all proven	c1, c5, c6, c7	read-after-write defects
Address stable across writes	all proven	c2, c7	2nd back-to-back write dropped
Selected + !WE means read	all proven	B6 vacuous; c3, w_oe_off	OE_n ignored
OE_n always low	all proven	B6 vacuous; c3, w_oe_off	OE_n ignored
WE_n always high	all proven	~half the covers	write lands at addr+1
CE_n implies unknown WE/OE	all proven	B4 vacuous; c4, w_deselect	write while deselected

Source: class report (VC Formal V-2023.12-SP2-3). No over-constraint produced a failing assertion.

Figure 2. The class over-constraint experiments, as reported. All assertions stayed proven; only covers and vacuity flags changed.

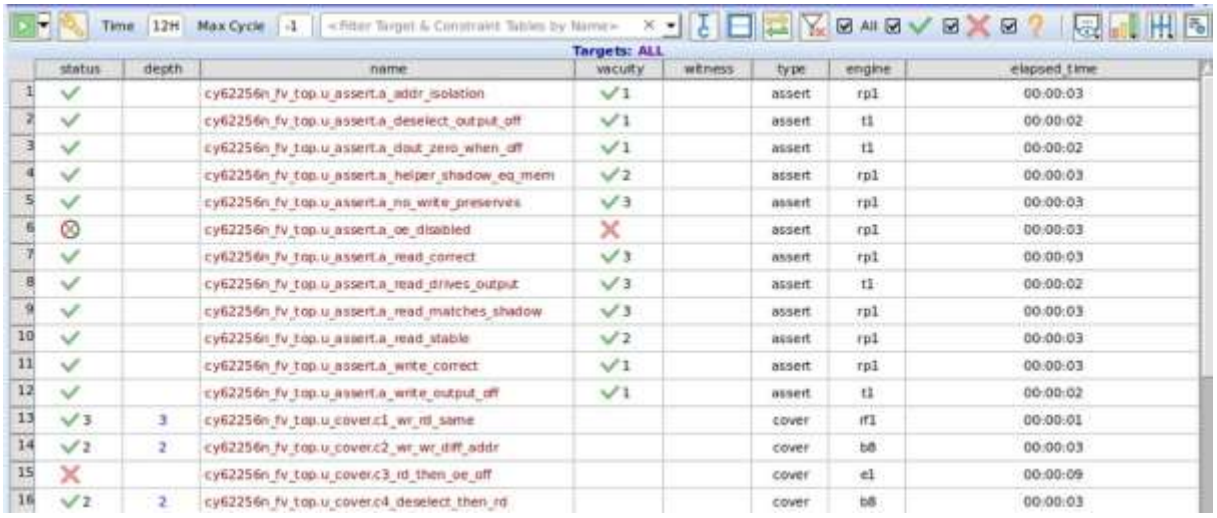


Figure 3. The OE_n-ignored bug under the OE_n-always-low over-constraint (Report I, VC Formal): a_oe_disabled is vacuous, not falsified, and c3_rd_then_oe_off is uncoverable. Screenshot from the class report.

We reproduced this with Yosys 0.33 on the same model (rebuilt from the report, AW = 4), using an immediate-assertion form of B6 in a formal-only harness. With the OE_n-ignored bug and no over-constraint, the base case finds a counterexample at depth 2. With `OE_n` assumed low, the same check is **proved**, and a reachability probe on the B6 trigger (selected, WE_n high, OE_n high) is proved unreachable. The proof is green because the bug can no longer be stimulated.

The report concludes that covers are the only alarm. We would put it more precisely: covers and vacuity checks are the only alarms **the tool raises on its own**. Two more exist, and both are process steps. A documented assumption (RAG-SVA's DOC9 header: ASSUMPTION, TYPE, JUSTIFICATION, RISK LEVEL) can be reviewed against the datasheet before any run; "OE_n is always low" has no datasheet justification. And an assumption can be checked as an assertion against a real environment: in simulation, where

simulators check `assume` like `assert`, or as an obligation of the neighboring block in assume-guarantee reasoning.

4.3 True is not the same as inductive: B3

B3 holds on every reachable trace but was not inductive at full width. An induction step may start from an unreachable state where the shadow is valid but differs from memory; idle cycles and a read then break B3 (Report K found shadow 0x80, memory 0x08 with yosys/z3). The helper `exp_valid |-> (exp_data == mem_sym)` removes those states and is itself provable one cycle at a time.

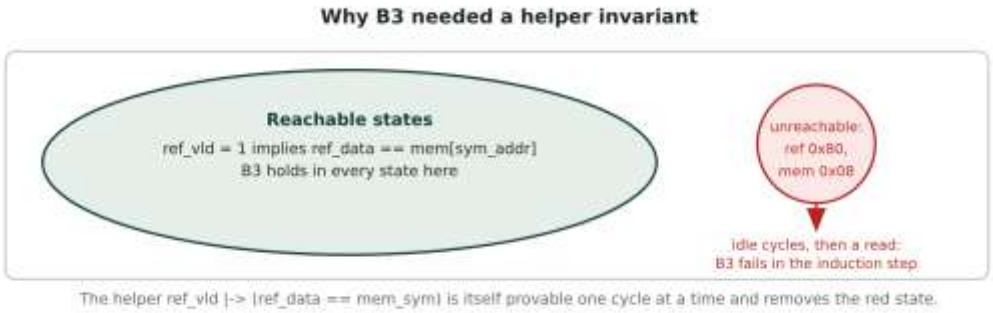


Figure 4. Why B3 needed a helper invariant. The state on the right is unreachable but is a legal start for an induction step.

Our Yosys run reproduces it at AW = 4: without the helper, induction does not close in 12 steps and no counterexample exists (it is true, not inductive); with the helper, both properties are proved by induction. With the addr+1 bug, B3 gives a real counterexample at depth 3. The practical rule: when a checker keeps reference state (a shadow, a scoreboard counter), write the invariant that ties it to the design state at the same time, rather than after a 66-minute stall. VC Formal proved B3 without a helper in two reports (D, G); the report suggests the engines found a strengthening invariant on their own, which a sign-off should not rely on.

4.4 Setup errors and who is to blame

Most early failures were in the environment: a symbolic address that moved (five properties falsified at depth 2), a reset declared twice (CONSTRAINT_CONFLICT), a reset never applied (failures at depth 0), and a launch directory that broke a relative path. The class labeled these correctly as environment errors. RAG-SVA makes that triage mechanical: a failing `mp_` points at the environment, a failing `ap_` at the design, and the PORT-INTENT table says which signals belong to which.

4.5 The missed bug is a configuration finding

The only escape, "reset clear removed", lives in a path enabled only when `INIT_ON_RESET = 1`, which no report ran. We ran it in simulation (Section 7): with `INIT_ON_RESET = 1` the bug is caught, but over three seeds only by the white-box helper (13 failures). The black-box B3 needs a read of `sym_addr` before the first write to it, which the random stimulus rarely produced; a formal run would find such a trace directly. Every model parameter needs at least one run, and a cover such as "read `sym_addr` before any write" makes the gap visible.

4.6 What the abstraction removed

The proofs are about the model, not the part. Removing timing removes every switching characteristic (tAA, tPWE, tSD, tHZWE). Two-state formal makes `asm_known_inputs` remove nothing, so X-propagation is not examined, as two reports noted. Modeling High-Z as `dout = 0` means a legitimate read of 0x00 and an off output differ only in `dout_oe`. And splitting the bus removes bus contention. None of this is wrong for a course, but a paper title of "SRAM formal verification" should say "untimed model" (Section 8).

5. Conformance to RAG-SVA

Much of the class practice is what RAG-SVA requires:

- **Assumptions only on inputs.** The report states that an assumption on `dout`, `dout_oe` or `mem_sym` would delete exactly the failing traces. That is the INTENTION boundary (§4.1 of the RAG-SVA INTENTION reference).
- **A cover for every assertion** (UR-5), and assumptions justified with "why", "evidence" and "can it hide a bug": the content of a DOC9 header, in prose.
- **Symbolic address and one-byte shadow**, the same data-integrity technique as the authors' CY62256N checker.
- **OPS1**: Boolean contexts use `!`, `&&`, `||`; no banned operators (`until`, `within`, `first_match`, `##[0:$]`).

Where it departs:

RAG-SVA rule	Class code	RAG-SVA form
`|=>` banned (INC-3)	B1, B7, B8, two over-constraints	`|-> ##1`
Labels ap_ / mp_ / cp_ and named p_ properties (DOC3)	a_, asm_, c1_, w_; inline properties	ap_write_stores, mp_sym_addr_const, cp_*_trigger
CHECK 21: assertion-only expressions use `let`	logic + assign for wr, rd, wr_sym, ...	`let wr = !CE_n && !WE_n;`
DOC9 header on each assumption	prose in the report	header in the code
PORT-INTENT table	implied	table in the checker header
Reference from assume signals only	B3 shadow learns from dout	ref_data from din only (Section 6.3)
White-box checks labeled	mixed with black-box	separate section, labeled WHITE-BOX
Line-item requirements and read-back	truth table cited	REQ IDs and read-back (V5 paper)
TOOL EXECUTION: compile and simulate	formal only, one tool	slang + simulation + formal

6. The fixes with RAG-SVA

The complete rewrite, `rag_sva.sv`, binds to the same model and ports as the class checker. Its header carries the PORT-INTENT table and the property inventory:

```
// PORT-INTENT
//   | Signal              | Dir | INTENTION  | Note                              |
//   |---------------------|-----|------------|-----------------------------------|
//   | rst_n               | in  | assume     | reset applied by the flow         |
//   | CE_n, WE_n, OE_n    | in  | assume     | all 8 combinations legal          |
//   | addr, din           | in  | assume     |                                   |
//   | sym_addr            | in  | assume     | free symbolic address (checker)   |
//   | dout, dout_oe       | out | obligation |                                   |
//   | mem_sym             | tap | obligation | WHITE-BOX: dut.mem[sym_addr]      |
//
// PROPERTY INVENTORY (class label in brackets)
//   ap_write_stores [B1, white-box]     ap_read_copies_mem [B2, white-box]
//   ap_read_matches_write [B3]          ap_read_consistent [new; replaces learning]
//   ap_mem_changes_only_on_write [B7+B8, white-box]
//   ap_deselect_hz [B4] ap_write_hz [B5] ap_oe_off_hz [B6] ap_read_drives [B10]
//   ap_hz_is_zero [kit] ap_read_stable [B9]  ap_helper_shadow_mem [helper]
//   mp_sym_addr_const [D0] mp_known_inputs [D3]   cp_* : one per ap_/mp_ + scenarios
```

6.1 let instead of assign

The class declares its helper conditions as nets:

```
  logic wr, rd, oe_off_sel, wr_sym, rd_sym, wr_other;
  assign wr         = !CE_n && !WE_n;
  assign rd         = !CE_n && WE_n && !OE_n;
  assign oe_off_sel = !CE_n && WE_n && OE_n;
  assign wr_sym     = wr && (addr == sym_addr);
  assign rd_sym     = rd && (addr == sym_addr);
  assign wr_other   = wr && (addr != sym_addr);
```

RAG-SVA CHECK 21 requires `let` for any expression whose only consumers are assertions, assumptions and covers:

```
  // Named expressions (CHECK 21: let, not logic + assign)
  //------------------------------------------------------------------------------
  let wr         = !CE_n && !WE_n;           // Truth Table: Write
  let rd         = !CE_n &&  WE_n && !OE_n;  // Truth Table: Read
  let oe_off_sel = !CE_n &&  WE_n &&  OE_n;  // Truth Table: Output disabled
  let at_sym     = (addr == sym_addr);
  let wr_sym     = (wr && at_sym);
  let rd_sym     = (rd && at_sym);
```

The reasons, from CHECK 21 and IEEE 1800 §11.12:

- **No spurious nets.** `logic` + `assign` declares a real signal in the checker or bind scope that has no design purpose. A `let` declares nothing; it is a named expression.
- **Assertion sampling, not continuous-assignment scheduling.** A `let` is expanded where it is used, so inside a property it is evaluated with the property's sampled values and clock. A net is evaluated by continuous assignment; when the expression mixes `$past`, `$rose` or `$fell` with sampled signals, the two can differ. (Here the expressions are purely combinational, so the class nets are correct; the rule prevents the cases where they would not be.)
- **Arguments.** A `let` can take arguments, such as `let at(a) = (addr == a)`, which a net cannot.

- **Clarity of intent.** A reader sees at once that `wr` is a name for a datasheet condition, not logic that might be read by something else. When a net really is needed elsewhere (a waveform tap, another monitor), CHECK 21 permits it with an `// AUX-NET:` justification.

One practical note from our compile: slang 12.1.0 warned "&& within ||" where `wr_sym`, a `let` of an && expression, was used inside an || condition. A `let` behaves as one operand, so the logic was correct, but we parenthesized both the `let` body and the use site to keep the compile at zero diagnostics.

6.2 Documented assumptions

```
// ASSUMPTIONS
//============================================================================
/*
ASSUMPTION:    sym_addr keeps the value chosen after reset.
TYPE:          State reduction (symbolic address)
JUSTIFICATION: One free, constant address stands for every address.
RISK LEVEL:    Low -- sym_addr drives no DUT logic; its value stays free.
*/
property p_sym_addr_const;
  $stable(sym_addr);
endproperty
mp_sym_addr_const: assume property (@(posedge clk) disable iff (!rst_n) p_sym_addr_const);

/*
ASSUMPTION:    Inputs are never X or Z.
TYPE:          Input constraint (documentation)
JUSTIFICATION: Datasheet inputs are driven levels.
RISK LEVEL:    Low in formal (two-state: removes no traces); in 4-state
               simulation it flags an X-driving testbench.
*/
property p_known_inputs;
  !$isunknown({CE_n, WE_n, OE_n, addr, din, sym_addr});
endproperty
mp_known_inputs: assume property (@(posedge clk) disable iff (!rst_n) p_known_inputs);
```

6.3 A reference built only from inputs

The class shadow records `din` on each write to `sym_addr`, and on the first read of an address never written it **learns** `dout`: the DUT's own output becomes the expected value it is checked against (Figure 5). Under RAG-SVA INTENTION rules, the reference for a correctness check must come from assume signals; `dout` is an obligation. The datasheet does not define power-up contents, so RQM0 would stop and ask; the answer "contents are undefined" means **no correctness claim until a write**. What remains is a weaker, honest property, which we state separately and name for what it is:

```
// Reference byte for sym_addr
// AUX: INPUT-DRIVEN ONLY (CE_n, WE_n, addr, sym_addr, din) -- never from dout.
// Valid after a write to sym_addr, or after reset when INIT_ON_RESET = 1.
//----------------------------------------------------------------------------
logic       ref_vld;
logic [7:0] ref_data;
always_ff @(posedge clk)
  if (!rst_n)      begin ref_vld <= INIT_ON_RESET; ref_data <= '0;  end
  else if (wr_sym) begin ref_vld <= 1'b1;          ref_data <= din; end
```

```
  //------------------------------------------------------------------------
  // Last byte read from sym_addr since its last write
  // AUX: OBLIGATION-DRIVEN (dout). Used ONLY by ap_read_consistent, never as
  // the expected value of a correctness check.
  //------------------------------------------------------------------------
  logic       seen_vld;
  logic [7:0] seen_data;
  always_ff @(posedge clk)
    if (!rst_n || (wr_sym)) seen_vld <= 1'b0;
    else if (rd_sym)      begin seen_vld <= 1'b1; seen_data <= dout; end
```

```
  property p_read_matches_write;   // B3: a read returns the last byte written
    (rd_sym && ref_vld) |-> (dout_oe && (dout == ref_data));
  endproperty
  ap_read_matches_write: assert property (@(posedge clk) disable iff (!rst_n)
    p_read_matches_write) else $error("ap_read_matches_write: dout=%h expected=%h", dout,
ref_data);

  property p_read_consistent;      // new: two reads with no write between agree
    (rd_sym && seen_vld) |-> (dout == seen_data);
  endproperty
  ap_read_consistent: assert property (@(posedge clk) disable iff (!rst_n) p_read_consistent);
```

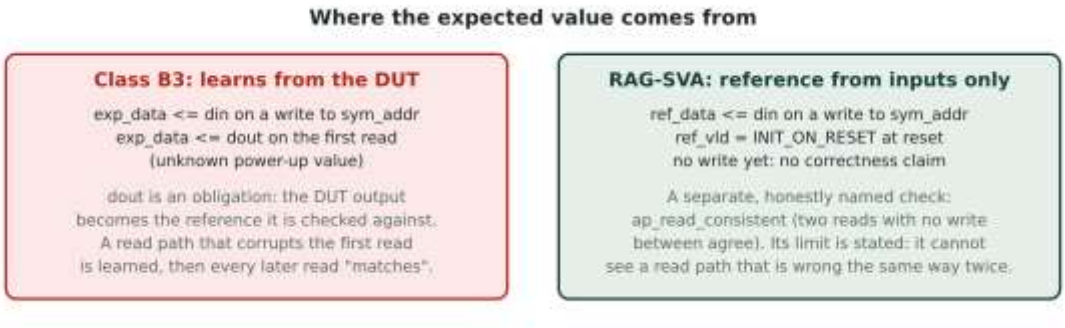


Figure 5. Two sources for the expected value. Learning from dout makes the check partly circular; RAG-SVA separates the correctness check from a labeled self-consistency check.

Section 7 shows what the difference does in practice: with no writes to `sym_addr`, the class B3 passed 1,367 checks on a corrupted read path; the RAG-SVA trigger cover for `ap_read_matches_write` reported zero, which says plainly that the black-box check never ran.

6.4 The remaining rewrites

- `|=>` becomes `|-> ##1` (B1, B7, B8).
- B7 and B8 overlap; together they say that `mem_sym` changes only on a write to `sym_addr`, which is one property: `!wr_sym |-> ##1 $stable(mem_sym)`.
- B9 needed a `past_valid` flag so `$past` would not look into the reset cycle. Written as a sequence antecedent, `rd ##1 (rd && $stable(addr)) |-> $stable(dout)`, both cycles lie after reset under `disable iff`, and the flag is not needed.
- White-box checks (B1, B2, B7/B8, the helper) are kept, but in their own section, labeled, so a reader never mistakes a peek at storage for a pin-level requirement.

```
  property p_write_stores;                                          // B1
    wr_sym |-> ##1 (mem_sym == $past(din));
  endproperty
  property p_read_copies_mem;                                       // B2
    rd_sym |-> (dout_oe && (dout == mem_sym));
  endproperty
  property p_mem_changes_only_on_write;                             // B7 + B8
    !wr_sym |-> ##1 $stable(mem_sym);
  endproperty
  property p_helper_shadow_mem;                                     // induction helper
    ref_vld |-> (ref_data == mem_sym);
  endproperty
  ap_write_stores:   assert property (@(posedge clk) disable iff (!rst_n) p_write_stores);
  ap_read_copies_mem: assert property (@(posedge clk) disable iff (!rst_n) p_read_copies_mem);
  ap_mem_changes_only_on_write: assert property (@(posedge clk) disable iff (!rst_n)
    p_mem_changes_only_on_write);
  ap_helper_shadow_mem: assert property (@(posedge clk) disable iff (!rst_n) p_helper_shadow_mem);
```

7. The bug list, re-run against both checkers

We rebuilt the class model from the report (it is not the students' file) with the twelve distinct bugs of the report's table behind defines, bound the class checker and the RAG-SVA checker side by side, and simulated each configuration with Verilator 5.055 (development version, built from source) for three seeds of 4,000 random cycles (AW = 6, random power-up memory contents). Both checkers compile with slang 12.1.0 with 0 diagnostics. On the correct model, neither checker fails and all 34 covers (17 per checker) are hit, with `INIT_ON_RESET` = 0 and = 1.

Bug (report table)	Class checks that failed	RAG-SVA checks that failed
Write lands at addr+1	a_write_correct 1942, a_addr_isolation 500, a_read_matches_shadow 1571, helper 11968	ap_write_stores 1942, ap_mem_changes_only_on_write 500, ap_read_matches_write 1571, ap_read_consistent 71, helper 11968
Write while deselected	a_no_write_preserves 438, a_read_matches_shadow 278, helper 2204	ap_mem_changes_only_on_write 438, ap_read_matches_write 278, ap_read_consistent 65, helper 2204
Output on during write	a_write_output_off 4886	ap_write_hz 4886
OE_n ignored on read	a_oe_disabled 1165	ap_oe_off_hz 1165
WE_n ignored on output	a_write_output_off 1217	ap_write_hz 1217
Write-data LSB corrupted	a_write_correct 997, a_read_matches_shadow 821, helper 6255	ap_write_stores 997, ap_read_matches_write 821, helper 6255
dout_oe broken	a_read_drives_output 3746, a_read_correct 1574, a_read_matches_shadow 1574	ap_read_drives 3746, ap_read_copies_mem 1574, ap_read_matches_write 1574
Read data bit 0 flipped	a_read_correct 1574, a_read_matches_shadow 1574	ap_read_copies_mem 1574, ap_read_matches_write 1574
Second back-to-back write dropped	a_write_correct 457, a_read_matches_shadow 381, helper 2638	ap_write_stores 457, ap_read_matches_write 381, helper 2638
Word inverted while deselected	a_no_write_preserves 883, a_read_matches_shadow 316, helper 2644	ap_mem_changes_only_on_write 883, ap_read_matches_write 316, ap_read_consistent 86, helper 2644

Read after write returns 0	a_read_correct 462, a_read_matches_shadow 462, a_read_stable 241	ap_read_copies_mem 462, ap_read_matches_write 462, ap_read_stable 241, ap_read_consistent 282
Reset clear removed, INIT_ON_RESET = 0	none (path not used)	none (path not used)
Reset clear removed, INIT_ON_RESET = 1	helper 13	helper 13
Read bit 0 flipped, sym_addr never written	a_read_correct 1370, helper 11951; a_read_matches_shadow PASSES (witness hit 1,367 times)	ap_read_copies_mem 1370; ap_read_matches_write trigger cover = 0 (check never ran); ap_read_consistent passes, as stated

Table 4. Assertions that failed, summed over three seeds. "helper" is the induction helper. The last row uses a stimulus mode that never writes `sym_addr`.

Three observations. Both checkers catch the same eleven bugs, so the rewrite loses nothing. The new `ap_read_consistent` adds a pin-level failure on four storage bugs (addr+1, write while deselected, inverted while deselected, read after write returns 0). And the last row is the circular-reference case: the class B3 passed while its witness fired 1,367 times; only the white-box checks caught the corrupted read path.

8. The proposed model

The class model is right for teaching the truth table, but its abstractions decide which assumptions are legitimate. The authors' earlier work (Cohen and Chibani, 2026b) built a timed model of the -70 part from the same datasheet, with RAG-SVA: a 20 ns sampling clock, registered outputs, nanosecond limits rounded up to whole samples (tAA 70 ns = 4 samples, tPWE 50 ns = 3), the bus split into `dq_i`, `dq_o`, `dq_oe` with a pad adapter for simulation, and no reset, as on the part.

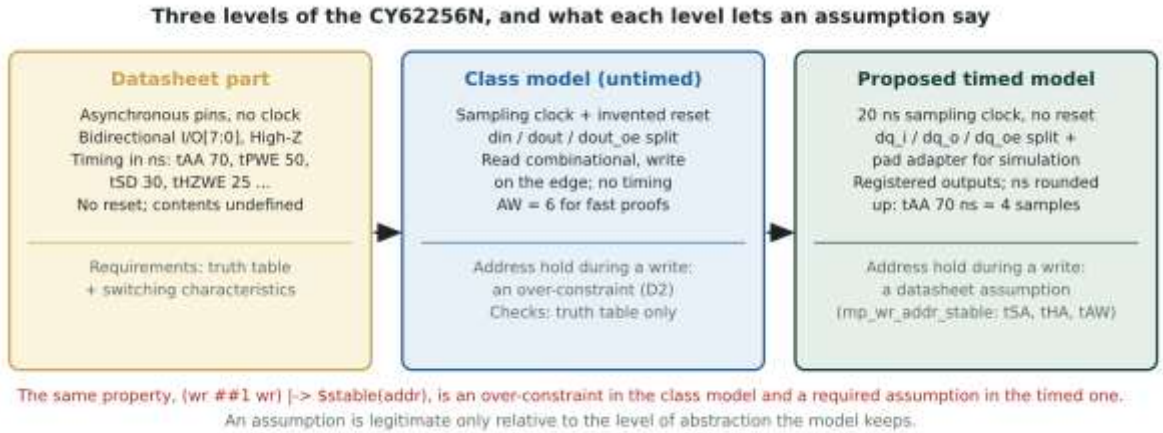


Figure 6. Three levels of the CY62256N. The same address-hold property is an over-constraint in the class model and a datasheet assumption in the timed model.

The comparison makes one point the class report could not: **an assumption is legitimate only relative to the level of abstraction the model keeps**. Report D's "address stable across back-to-back writes" hid a real bug in the untimed model, where every cycle can be a new write. In the timed model a write lasts at least three samples, and the address must be stable for all of them (tSA, tHA, tAW), so the same property is the documented assumption `mp_wr_addr_stable`. Figure 7 shows the proposed checker organization for either level.

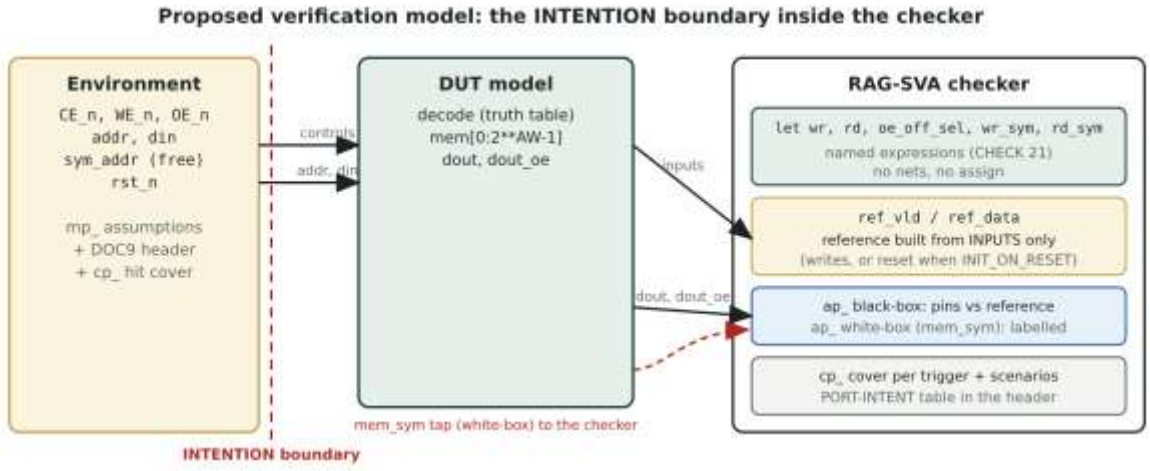


Figure 7. Proposed verification model: environment signals constrained by documented mp_ assumptions, a reference built from inputs only, black-box and labeled white-box assertions, and a cover per trigger.

We simulated the timed model, its RAG-SVA checker (6 ap_, 5 mp_, 17 cp_) and its directed-plus-random testbench with Verilator 5.055, three seeds each, and added three of the class bugs to the timed model. Two findings came first, both from covers. The four data-read covers were never hit, although the testbench reads address 5 many times. A probe showed the reason: Verilator 5.055 never matched a sequence that has a formal argument (`s_rd_at_window(watch_addr)`); the same sequence written inline was hit 60 times in one run. For simulation we therefore used a copy of the checker with that sequence expanded by hand; the delivered checker is unchanged. Second, slang required a `timescale` in the checker, because the model and testbench declare one; we added it to the copy we ran.

Timed model	Checker result (3 seeds)	Notes
Correct	0 failures; 17 of 17 covers hit	testbench reference memory: 0 errors
Write lands at a+1	ap_rd_data and ap_rd_addr5_data: 60, 56, 54 failures	testbench reference memory also fails
OE_n ignored	ap_oe_hz: 194, 191, 193 failures	also seen once per seed by the testbench
Write while deselected	no failure	missed in simulation by the checker and the testbench (see text)

The last row is a finding against our own checker, and the class work shows the remedy. The random stimulus never pulls WE_n low while deselected at an address and data that would change a watched byte, and the timed checker has no property equivalent to the class B7/B8 (memory unchanged unless written). The data check `ap_rd_data` should expose the bug in a formal run, where the tool chooses the stimulus, but we have not run one. Adding a pin-level form of B7 to the timed checker, and a cover for "WE_n low while deselected", closes the gap.

9. Recommendations

1. **Review assumptions before running them.** Require a DOC9-style header with a datasheet justification and a risk level for every assumption. "OE_n is always low" fails that review in seconds.
2. **Report proof status in three columns**: proven, vacuous, unreachable cover. Do not let a goal count merge them.
3. **Build references from inputs only.** Where the specification leaves a value undefined, make no correctness claim; state any self-consistency check separately and name its limit.
4. **Plan helper invariants** with every piece of reference state, and keep them separate from requirement checks.
5. **Run every configuration** (here `INIT_ON_RESET` = 0 and 1), and add covers for corner orders such as a read before the first write.
6. **Use `let` for assertion-only expressions**, `|-> ##1` for next-cycle implication, and the ap_/mp_/cp_ labels, so the code itself says what is required, assumed and covered.
7. **Match assumptions to the model's abstraction**, and say in the title which model was verified.
8. **Check simulation tools with covers.** A cover that is never hit may be a tool limitation, as in Section 8, not only a stimulus gap.

10. Conclusion

The class report shows, at a scale no single engineer produces, what formal verification looks like when assumptions go wrong: nothing fails. The teams found that only covers and vacuity flags spoke up. RAG-SVA turns the class's hard-won habits into rules that apply before a run: assumptions documented and reviewed, labels that separate the environment from the design, references built only from inputs, a cover for every trigger, and real tool runs reported truthfully. The bug re-run shows the rewrite keeps every detection and adds visibility where the class check was quietly circular, and the timed model shows that the right assumptions depend on the abstraction. The students' work is the evidence; the rules are how the next class, and the next project, gets the same results without first being surprised by a green proof.

Acknowledgments and disclosure

We thank the students of ECE-574, Fall 2026, whose anonymized reports (A to L) are the basis of this paper. Screenshots in Figures 1 and 3 are from those reports. This draft, its code and its tool runs were prepared with AI assistance (Claude) under the authors' direction, using the RAG-SVA rule files.

Appendix A. Tool results

Step	Result
Compile	slang 12.1.0: class model, class checker, RAG-SVA checker, testbench: 0 diagnostics. Timed model, checker (with `timescale` added) and testbench: 0 diagnostics.
Simulation, untimed	Verilator 5.055, 15 configurations x 3 seeds x 4,000 cycles, random power-up memory; class and RAG-SVA checkers bound together; covers via --coverage-user: 34 of 34 hit on the correct model.
Simulation, timed	Verilator 5.055, 4 configurations x 3 seeds; checker copy with the formal-argument sequence expanded inline (tool limitation).
Formal	Yosys 0.33 sat -tempinduct, AW = 4, immediate-assertion equivalents in a formal-only harness (open-source Yosys does not read SVA sequences). B3 without helper: induction not closed in 12 steps, no counterexample; with helper: proved; addr+1 bug: counterexample, depth 3; B6 with OE_n-ignored bug: counterexample, depth 2; same with OE_n assumed low: proved; B6 trigger reachability: reachable, and unreachable under the over-constraint. Not a commercial sign-off.
Class results	VC Formal V-2023.12-SP2-3 results are quoted from the class report; we did not run VC Formal.

Appendix B. The complete RAG-SVA checker

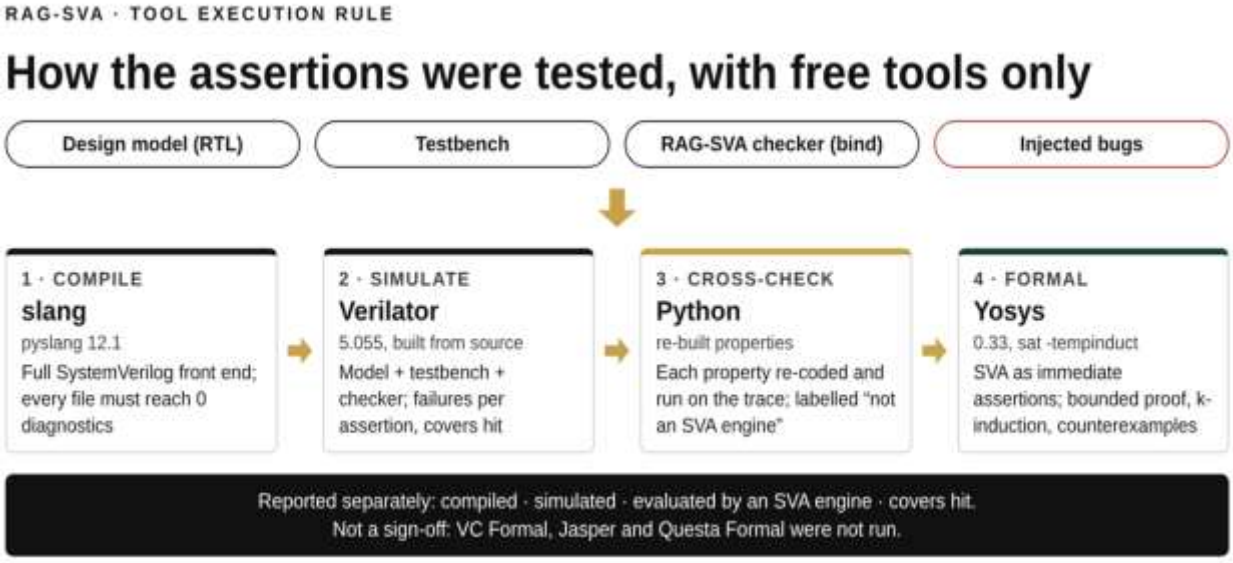
```
// rag_sva.sv -- RAG-SVA rewrite of the class checks for the untimed CY62256N model
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//============================================================================
// Framework : RAG-SVA (Cohen, Chibani) -- systemverilog.us
// Settings  : FV MODE=YES | Labels ap_/mp_/cp_/p_ | |-> ##1 (no |=>) | CHECK 21 (let)
// Source    : CY62256N datasheet Truth Table (p. 11); model abstractions per class report
//
// PORT-INTENT
//   | Signal              | Dir | INTENTION  | Note                              |
//   |---------------------|-----|------------|-----------------------------------|
//   | rst_n               | in  | assume     | reset applied by the flow         |
//   | CE_n, WE_n, OE_n    | in  | assume     | all 8 combinations legal          |
//   | addr, din           | in  | assume     |                                   |
//   | sym_addr            | in  | assume     | free symbolic address (checker)   |
//   | dout, dout_oe       | out | obligation |                                   |
//   | mem_sym             | tap | obligation | WHITE-BOX: dut.mem[sym_addr]      |
//
// PROPERTY INVENTORY (class label in brackets)
//   ap_write_stores [B1, white-box]     ap_read_copies_mem [B2, white-box]
//   ap_read_matches_write [B3]          ap_read_consistent [new; replaces learning]
//   ap_mem_changes_only_on_write [B7+B8, white-box]
//   ap_deselect_hz [B4] ap_write_hz [B5] ap_oe_off_hz [B6] ap_read_drives [B10]
//   ap_hz_is_zero [kit] ap_read_stable [B9]  ap_helper_shadow_mem [helper]
//   mp_sym_addr_const [D0] mp_known_inputs [D3]   cp_* : one per ap_/mp_ + scenarios
//============================================================================
module rag_sva #(parameter int AW = 6, parameter bit INIT_ON_RESET = 1'b0) (
  input logic          clk, rst_n,
  input logic          CE_n, WE_n, OE_n,   // assume
  input logic [AW-1:0] addr, sym_addr,     // assume
  input logic [7:0]    din,                // assume
  input logic [7:0]    dout,               // obligation
  input logic          dout_oe,            // obligation
  input logic [7:0]    mem_sym             // obligation (white-box tap)
);
```

```
//------------------------------------------------------------------------
// Named expressions (CHECK 21: let, not logic + assign)
//------------------------------------------------------------------------
let wr         = !CE_n && !WE_n;           // Truth Table: Write
let rd         = !CE_n &&  WE_n && !OE_n;  // Truth Table: Read
let oe_off_sel = !CE_n &&  WE_n &&  OE_n;  // Truth Table: Output disabled
let at_sym     = (addr == sym_addr);
let wr_sym     = (wr && at_sym);
let rd_sym     = (rd && at_sym);

//------------------------------------------------------------------------
// Reference byte for sym_addr
// AUX: INPUT-DRIVEN ONLY (CE_n, WE_n, addr, sym_addr, din) -- never from dout.
// Valid after a write to sym_addr, or after reset when INIT_ON_RESET = 1.
//------------------------------------------------------------------------
logic       ref_vld;
logic [7:0] ref_data;
always_ff @(posedge clk)
  if (!rst_n)      begin ref_vld <= INIT_ON_RESET; ref_data <= '0;  end
  else if (wr_sym) begin ref_vld <= 1'b1;          ref_data <= din; end

//------------------------------------------------------------------------
// Last byte read from sym_addr since its last write
// AUX: OBLIGATION-DRIVEN (dout). Used ONLY by ap_read_consistent, never as
// the expected value of a correctness check.
//------------------------------------------------------------------------
logic       seen_vld;
logic [7:0] seen_data;
always_ff @(posedge clk)
  if (!rst_n || (wr_sym)) seen_vld <= 1'b0;
  else if (rd_sym)      begin seen_vld <= 1'b1; seen_data <= dout; end

//========================================================================
// ASSUMPTIONS
//========================================================================
/*
ASSUMPTION:    sym_addr keeps the value chosen after reset.
TYPE:          State reduction (symbolic address)
JUSTIFICATION: One free, constant address stands for every address.
RISK LEVEL:    Low -- sym_addr drives no DUT logic; its value stays free.
*/
property p_sym_addr_const;
  $stable(sym_addr);
endproperty
mp_sym_addr_const: assume property (@(posedge clk) disable iff (!rst_n) p_sym_addr_const);

/*
ASSUMPTION:    Inputs are never X or Z.
TYPE:          Input constraint (documentation)
JUSTIFICATION: Datasheet inputs are driven levels.
RISK LEVEL:    Low in formal (two-state: removes no traces); in 4-state
               simulation it flags an X-driving testbench.
*/
property p_known_inputs;
  !$isunknown({CE_n, WE_n, OE_n, addr, din, sym_addr});
endproperty
mp_known_inputs: assume property (@(posedge clk) disable iff (!rst_n) p_known_inputs);

//========================================================================
// ASSERTIONS -- black-box (pins only)
//========================================================================
property p_read_matches_write;   // B3: a read returns the last byte written
  (rd_sym && ref_vld) |-> (dout_oe && (dout == ref_data));
endproperty
```

```
  ap_read_matches_write: assert property (@(posedge clk) disable iff (!rst_n)
    p_read_matches_write) else $error("ap_read_matches_write: dout=%h expected=%h", dout,
ref_data);

  property p_read_consistent;      // new: two reads with no write between agree
    (rd_sym && seen_vld) |-> (dout == seen_data);
  endproperty
  ap_read_consistent: assert property (@(posedge clk) disable iff (!rst_n) p_read_consistent);

  property p_deselect_hz;  CE_n       |-> !dout_oe;  endproperty   // B4
  property p_write_hz;     wr         |-> !dout_oe;  endproperty   // B5
  property p_oe_off_hz;    oe_off_sel |-> !dout_oe;  endproperty   // B6
  property p_read_drives;  rd         |->  dout_oe;  endproperty   // B10
  property p_hz_is_zero;   !dout_oe   |-> (dout == '0); endproperty // kit
  property p_read_stable;                                          // B9
    rd ##1 (rd && $stable(addr)) |-> $stable(dout);
  endproperty
  ap_deselect_hz: assert property (@(posedge clk) disable iff (!rst_n) p_deselect_hz);
  ap_write_hz:    assert property (@(posedge clk) disable iff (!rst_n) p_write_hz);
  ap_oe_off_hz:   assert property (@(posedge clk) disable iff (!rst_n) p_oe_off_hz);
  ap_read_drives: assert property (@(posedge clk) disable iff (!rst_n) p_read_drives);
  ap_hz_is_zero:  assert property (@(posedge clk) disable iff (!rst_n) p_hz_is_zero);
  ap_read_stable: assert property (@(posedge clk) disable iff (!rst_n) p_read_stable);

  //============================================================================
  // ASSERTIONS -- white-box (mem_sym tap), kept separate and labelled
  //============================================================================
  property p_write_stores;                                         // B1
    wr_sym |-> ##1 (mem_sym == $past(din));
  endproperty
  property p_read_copies_mem;                                      // B2
    rd_sym |-> (dout_oe && (dout == mem_sym));
  endproperty
  property p_mem_changes_only_on_write;                            // B7 + B8
    !wr_sym |-> ##1 $stable(mem_sym);
  endproperty
  property p_helper_shadow_mem;                                    // induction helper
    ref_vld |-> (ref_data == mem_sym);
  endproperty
  ap_write_stores:   assert property (@(posedge clk) disable iff (!rst_n) p_write_stores);
  ap_read_copies_mem: assert property (@(posedge clk) disable iff (!rst_n) p_read_copies_mem);
  ap_mem_changes_only_on_write: assert property (@(posedge clk) disable iff (!rst_n)
    p_mem_changes_only_on_write);
  ap_helper_shadow_mem: assert property (@(posedge clk) disable iff (!rst_n) p_helper_shadow_mem);

  //============================================================================
  // COVERS (UR-5): one per assumption and assertion trigger, plus scenarios
  //============================================================================
  cp_sym_addr_const_hit:   cover property (@(posedge clk) disable iff (!rst_n) wr_sym ##[1:4]
rd_sym);
  cp_known_inputs_hit:     cover property (@(posedge clk) disable iff (!rst_n) wr && (din != '0));
  cp_read_matches_write_trigger: cover property (@(posedge clk) disable iff (!rst_n) rd_sym &&
ref_vld);
  cp_read_consistent_trigger:    cover property (@(posedge clk) disable iff (!rst_n) rd_sym &&
seen_vld);
  cp_deselect_hz_trigger:  cover property (@(posedge clk) disable iff (!rst_n) CE_n);
  cp_write_hz_trigger:     cover property (@(posedge clk) disable iff (!rst_n) wr);
  cp_oe_off_hz_trigger:    cover property (@(posedge clk) disable iff (!rst_n) oe_off_sel);
  cp_read_drives_trigger:  cover property (@(posedge clk) disable iff (!rst_n) rd);
  cp_hz_is_zero_trigger:   cover property (@(posedge clk) disable iff (!rst_n) !dout_oe);
  cp_read_stable_trigger:  cover property (@(posedge clk) disable iff (!rst_n) rd ##1 (rd &&
$stable(addr)));
  cp_write_stores_trigger: cover property (@(posedge clk) disable iff (!rst_n) wr_sym);
  cp_read_copies_mem_trigger: cover property (@(posedge clk) disable iff (!rst_n) rd_sym);
```

```
  cp_mem_changes_only_on_write_trigger: cover property (@(posedge clk) disable iff (!rst_n) wr &&
!at_sym);
  cp_helper_shadow_mem_trigger: cover property (@(posedge clk) disable iff (!rst_n) ref_vld);
  cp_scn_wr_rd_same: cover property (@(posedge clk) disable iff (!rst_n)          // C1
    wr ##1 (rd && (addr == $past(addr)) && dout_oe && (dout == $past(din))));
  cp_scn_rd_then_oe_off: cover property (@(posedge clk) disable iff (!rst_n)      // C3
    rd ##1 oe_off_sel);
  cp_scn_overwrite_same: cover property (@(posedge clk) disable iff (!rst_n)      // C6
    wr ##1 (wr && (addr == $past(addr)) && (din != $past(din)))
       ##1 (rd && (addr == $past(addr)) && dout_oe && (dout == $past(din))));
endmodule
```

Appendix C. The Free Tools Used by RAG-SVA



References

[1] V. Patil, "Assignment 3: CY62256N SRAM Formal Verification, Consolidated Class Report," ECE-574 Assertion Based Verification, Portland State University, Fall 2026.

[2] B. Cohen, K. Chibani, *Knowledge-Driven AI for SystemVerilog Assertions*, Cohen Publishing, 2026.

[3] B. Cohen, K. Chibani, "From Datasheet to Traceable Requirements: The CY62256N SRAM Example," systemverilog.us/V5_Vendor_Datasheet_to_Traceable_Requirements_The_CY62256N_SRAM.pdf, 2026.

[4] B. Cohen, K. Chibani, "Can You Trust AI-Generated Assertions? Catching SVA That Passes but Does Not Match the Spec," systemverilog.us/Trusting_AI_Generated_SVA_CY62256N.pdf, 2026.

[5] B. Cohen, K. Chibani, "Assume or Assert? Separating the Environment from the Design in SVA," systemverilog.us, 2026.

[6] Infineon/Cypress, CY62256N 256-Kbit (32K x 8) Static RAM datasheet, Doc. 001-06511.

[7] IEEE Std 1800-2023, *IEEE Standard for SystemVerilog*, 11.12 (let construct), clause 16 (assertions).

[8] Yosys, yosyshq.net; Verilator, verilator.org; slang, sv-lang.com.