From Datasheet to Traceable Requirements: The CY62256N SRAM Example

One AI-assisted process:

Requirements → Claude Opus 5.5 + RAG-SVA → Verified SVA checker, RTL model and testbench → read-back spec → commercial tools

Ben Cohen · Kais Chibani · systemverilog.us · October 30, 2026
Ben_Cohen@SystemVerilog.us

Summary

This paper describes a single process, shown in Figure 1, that starts from requirements documents in different formats (*for example, a vendor's PDF datasheet*) and ends with requirements written in a form that a commercial verification flow can track. The example is the Infineon/Cypress CY62256N 32K × 8 asynchronous SRAM.[i]

The process has two paths. The **forward path** turns the datasheet into three files: an SVA checker[ii], an RTL model of the RAM[iii] and a testbench. Claude generates them under the RAG-SVA rule framework[iv]. The **return path** branches from the checker alone. The checker is read back into a new specification written as line-item requirements[v], and that specification is handed to commercial tools that carry it through their own design and verification flow: importing the requirements in a format they accept (*for example, a specification broken into a user-specified format*), planning, traceability and sign-off.

The return path is what makes the result usable by a team. An AI-generated checker is a file someone has to trust; a numbered list of requirements, each tied to a datasheet reference and an SVA label, is something a team can review, import and track.

The process

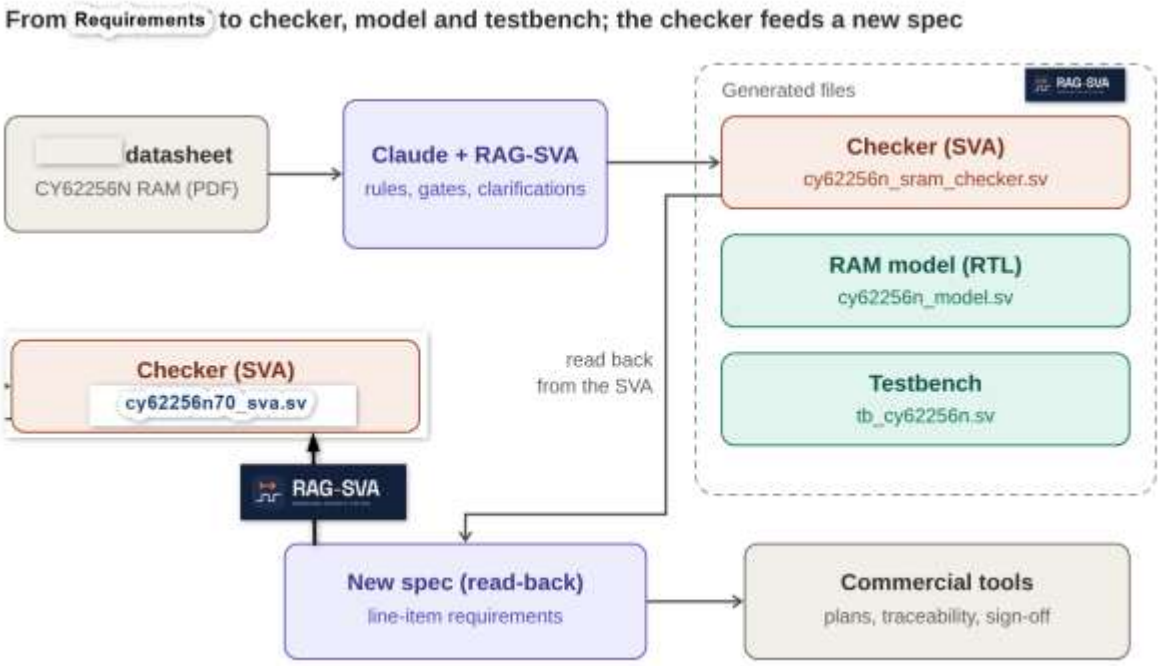


Figure 1. The process. The forward path (left to right) generates the checker, RTL model and testbench. The return path branches from the checker only: it is read back into a new spec, which feeds commercial tools. The new spec also goes back through RAG-SVA to generate a second checker, cy62256n70_sva.sv (left).

Figure 1 reads in two directions.

Left to right, generation. The datasheet is the only design input. Claude, running the RAG-SVA rules, does not start writing code. It first raises the questions the datasheet leaves open, as gates with options, and the engineer settles them. Only then does it produce the three generated files, elaborate them together and simulate the model and testbench.

Down and right, verification of the requirement. The return path starts at the SVA checker, not at the model or the testbench. The checker is the formal statement of what the device (*RAM in this example*) must do; the model and testbench exist to exercise it. Reading the checker back answers one question: *does the SVA say what the datasheet says?* The answer takes the form of a new spec, and the new spec goes to the commercial tools that will carry it to sign-off.

Back up, regeneration. The new spec is not only a document for review. Fed back into Claude + RAG-SVA as a requirements document (*`CY62256N-70_Verification_Requirements.docx`*), it produced a second SVA checker, *`cy62256n70_sva.sv`*, generated from the new requirements alone, without the datasheet. That second checker is shown on the left of Figure 1 and described in Stage 4b. Two checkers written from two different sources, the vendor PDF and the requirements derived from the first checker, can then be compared property by property.

The five stages, what goes in, what comes out and where the engineer spends time:

#	Stage	Input	Output	Engineer's role
1	Vendor datasheet	—	Timing tables, truth table, waveforms	Pick the part and speed grade
2	Claude + RAG-SVA	PDF, request, framework settings	Clarification questions, then code	Answer each question in a word or two
3	Generated files	Settled decisions	Checker (SVA), RTL model, testbench; Claude then elaborates all three and simulates the model and testbench	Review the files and the elaboration and simulation results
4	New spec (read-back)	The checker only	Line-item requirements	Compare each line with the PDF
4b	Second checker	The new spec only	cy62256n70_sva.sv, traced to requirement IDs	Compare the two checkers, requirement by requirement
5	Commercial tools	The new spec as data	Verification plan with live status	Track every line to sign-off

Time to create each file

Times are the session's own timestamps (Pacific time). "Elapsed" runs from the engineer's request to the delivered file and includes the engineer's own answers to the clarification questions. Claude elaborated and simulated each round before delivering it.

File	Generated from	Clock time	Elapsed	Engineer input in that time
`cy62256n_sram_checker.sv`, first version (5 ap_, 4 mp_, 14 cp_)	Vendor datasheet	Oct 20, 13:48 to 14:10	22 min (14 min from the first answers at 13:56)	Ten clarification answers, 13:56 to 14:07
`cy62256n_sram_checker.sv`, revision with the 0x0005 = 0xFE split (6 ap_, 5 mp_, 17 cp_)	First checker plus the new request	Oct 20, 14:12 to 14:26	14 min, shared with the next two files	CASE D split, data window and testbench choices, 14:19 and 14:21
`cy62256n_model.sv` (RTL model)	Vendor datasheet	Oct 20, 14:12 to 14:26	Same round	
`tb_cy62256n.sv` (testbench)	Checker and model	Oct 20, 14:12 to 14:26	Same round	
Elaboration (slang) and simulation (Verilator, 3 seeds)	All three files	Inside the 14:23 to 14:26 delivery	A few minutes, no engineer input	
Read-back spec (line-item requirements)	First checker only	Oct 28	~ 4 minutes *estimate*	
`CY62256N-70_Verification_Requirements.docx`	Read-back from `cy62256n_sram_checker.sv` to create RAG-SVA's clarified original specification	Later on	~ 7 minutes *estimate*	
`cy62256n70_sva.sv` (second checker)	From `CY62256N-70_Verification_Requirements.docx`	Later on	~ 10 minutes *estimate*	

The three files of the forward path took 36 minutes of wall-clock time from upload to the last delivery, of which about 28 minutes ran after the engineer's first answers. The times for the read-back spec, the requirements document and the second checker were not recorded in the session that produced the first three files; they are marked by the authors as estimates.

Stage 1 Vendor datasheet

The input for this example was the vendor's datasheet for the CY62256N (Infineon/Cypress document 001-06511) and a one-line request: "`write formal-verification assertions from this definition, packaged as a checker`". The datasheet supplies what any SRAM datasheet does: pin descriptions, a truth table, read and write switching characteristics in nanoseconds for each speed grade, and timing waveforms. It also leaves out what a verification engineer needs: there is no clock, no reset, and no statement of which side is the device and which is its environment.

Stage 2 Claude + RAG-SVA

Claude ran with the RAG-SVA framework (Cohen and Chibani) loaded as a Claude skill: a corpus of rule files that fixes SVA style, operator use, decomposition and labeling. Settings: `FV mode on, decomposition stage 3, abbreviated gates, compact output`. The framework's rules decide the style questions the same way every time:

- Inputs are assumed (`mp_`), outputs are asserted (`ap_`), and every assumption gets a cover (`cp_`).
- A property that mixes an input constraint with an output obligation is split, never written as an assumption over an output.
- Banned forms such as `|=>`, unbounded `##[0:$]`, `until` and `first_match` never appear.

What the rules cannot decide comes back to the engineer as questions. The first gate raised ten, each with options. The main ones:

Question	Decision
Clocking (the part is asynchronous)	Sample on a 20 ns `clk`; active-low `rstn` `Claude pointed out a flag about a 5ns minimum check`
Which side is the device	The SRAM is the DUT: controls, address and write data are assumptions; read data and output enable are obligations
Speed grade and rounding	-70; nanoseconds rounded up to whole 20 ns samples
Sub-sample parameters	tOHA and the low-Z times (5 ns) dropped
Bidirectional I/O	Split into `dq_i`, `dq_o`, `dq_oe`
Data integrity over 32K addresses	One free symbolic address, `watch_addr`, with a one-byte shadow
Power, supply, retention	Excluded

A later request, that address 0x0005 hold 0xFE, would have constrained read data in an assumption. The framework stopped it and proposed the split that was adopted: an assumption on writes to 0x0005 and an assertion on reads from it.

Stage 3 Generated files: checker, RTL model and testbench

Claude generates three files and then checks its own output before handing it over: it elaborates the checker, model and testbench together, and runs a simulation of the model and testbench. Each step is reported separately, so the engineer can see what was compiled, what was simulated and what was not executed.

Why the data bus is split for formal verification

The CY62256N has one bidirectional data bus, I/O[7:0]. The same eight pins carry write data into the chip, driven by the controller, and read data out of it, driven by the RAM. The truth table decides who owns the pins at each moment, and when neither side drives them they float in high impedance. That works on a board, but it does not work as the interface of a formal checker, for three reasons.

1. Intent. Every signal in a formal property must have one owner. An input to the device is *assumed* (the environment promises it), and an output is *asserted* (the device must deliver it). A single inout signal is both at once, so any property written on it is half assumption and half obligation. The RAG-SVA rules stop exactly that case (CASE D): an assumption may never constrain a value the device itself drives, because the tool would then prove the design correct by forbidding the very behavior it was meant to check.

2. Tri-state and resolution. Formal tools work with two-state logic. A high-impedance 'z, weak drivers and the resolution of two drivers on one wire are not modeled faithfully, so "the output is in high Z" cannot be checked directly on the shared wires.

3. Driver identity. The tool must know which values are free inputs it may choose and which are produced by the design. On a shared bus that depends on the cycle; on separate buses it is fixed by the port direction.

The checker therefore sees the bus as three signals, shown in Figure 2:

Signal	Direction	Driven by	Intention	Replaces
`dq_i[7:0]`	Into the RAM	Controller	Assume	I/O[7:0] during a write
`dq_o[7:0]`	Out of the RAM	RAM model	Assert	I/O[7:0] during a read
`dq_oe`	Out of the RAM	RAM model	Assert	The tri-state enable: 1 = driving, 0 = high Z

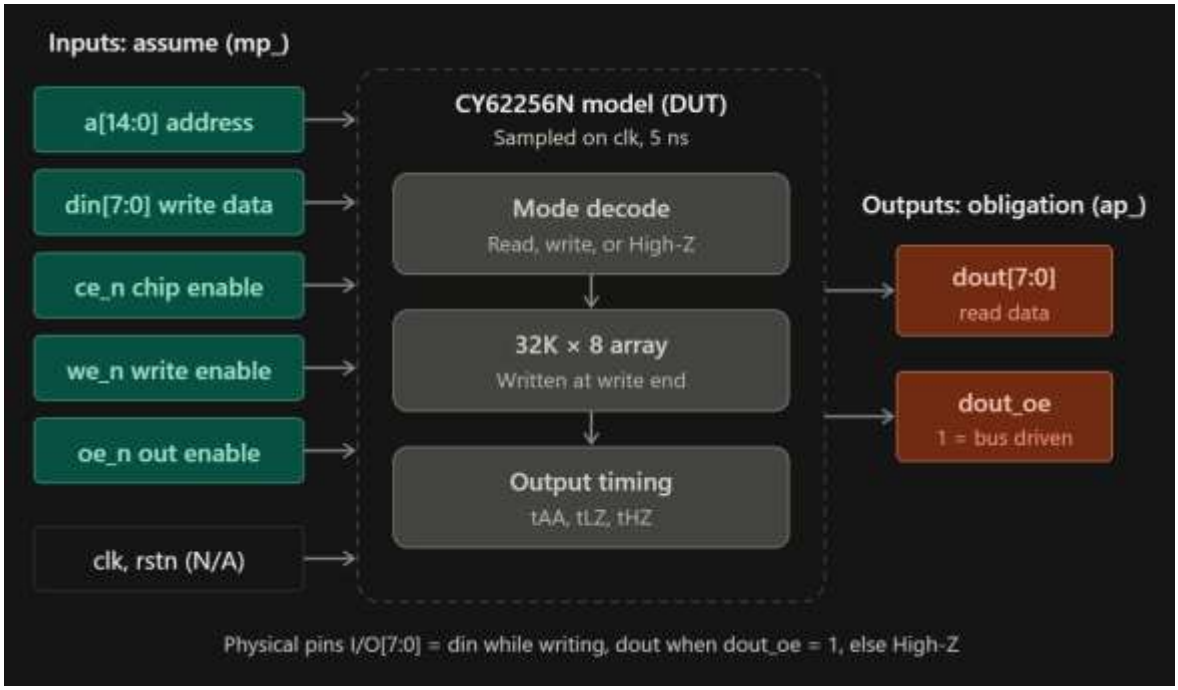


Figure 2. RAM architecture for formal verification. The controller drives the control signals, the address and the input bus dq_i; the RAM drives the output bus dq_o and its enable dq_oe. The checker is bound to the RAM and sees every signal. A pad adapter rebuilds the real inout pins for simulation, and is not part of the formal model.

With the split, the datasheet's high-Z timing becomes a plain two-state property: tHZCE, tHZOE and tHZWE are checked as `dq_oe` falling to 0 (REQ-DEV-001 to 003), and read data is checked on `dq_o` whenever `dq_oe` is 1 (REQ-DEV-004, 005). The second checker, `cy62256n70_sva.sv`, uses the same split under the names `din`, `dout` and `dout_oe`.

Reconnecting to the pins. Where the real inout bus is needed, in a board-level testbench or around a silicon model, a small adapter rebuilds it: the pins carry `dq_o` when `dq_oe` is 1 and float otherwise, and `dq_i` simply reads the pins. The adapter stays outside the formal model.

What the split gives up. On the real bus, the controller must not drive the pins while the RAM's outputs are on (datasheet Note 19). With separate buses the two drivers can never collide, so that rule is no longer visible to the checker. It is tracked as REQ-GAP-004, and is closed by adding a controller drive-enable to the adapter and one assertion that it is never active together with `dq_oe`.

Checker (SVA)

`cy62256n_sram_checker.sv` contains:

- **Assertions (ap_):** 6. Three high-Z releases (CE high, OE high, WE low), output drive in a read window, read data equal to the last write at `watch_addr`, and read data at 0x0005 equal to 0xFE.
- **Assumptions (mp_):** 5. Write-pulse width, address stability, write-data setup, the 0x0005 = 0xFE write constraint, and a constant `watch_addr`.
- **Covers (cp_):** 17. A trigger cover and a full cover for each assertion, and a hit cover for each assumption.

Every property is bounded safety, so the proof needs no fairness assumptions.

RTL model

`cy62256n_model.sv` is a cycle-based behavioral model with a 32K × 8 array. It writes on every sample with CE and WE low, and drives `dq_o` and `dq_oe` one cycle after the address and controls. It is deliberately simple: a stand-in DUT that gives the checker something to run against.

Testbench

`tb_cy62256n.sv` runs five directed tests (0x0005 = 0xFE write and read-back, corner addresses 0x0000 and 0x7FFF, a CE-controlled write, release to high Z on OE and on CE), then 500 weighted-random operations: write 40%, read 40%, idle 20%, with 0x0005 chosen 30% of the time. A reference memory checks every read, and all stimulus satisfies the checker's assumptions.

Elaboration and simulation by Claude

Elaboration. Claude elaborated all three files together in slang (pyslang 12.0.0): parsing, binding the checker to the model's ports, and resolving every parameter, sequence and property. Result: zero errors and zero warnings.

Simulation. Claude then compiled the model and testbench in Verilator 5.020 and ran them on three seeds: the five directed tests plus 500 weighted-random operations, with the testbench's reference memory checking every read. All three runs passed.

What the simulation could not do. Verilator 5.020 cannot run the checker's SVA sequences, so the checker was elaborated but not executed in simulation. Claude exercised the property logic instead through a Python stand-in, shown in Figure 3.

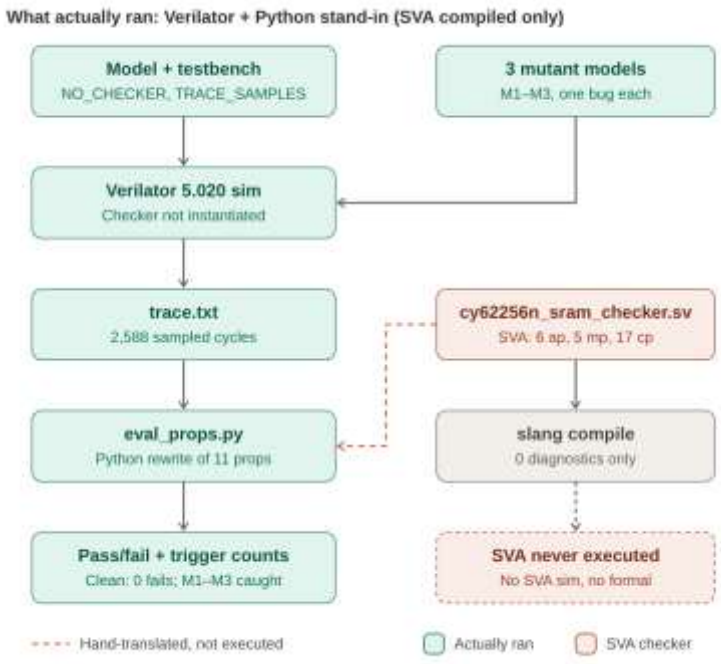


Figure 3. What actually ran. Green boxes executed; the SVA checker was only compiled. The dashed line is a manual translation, not an execution.

1. The model and testbench were simulated without the checker, with a trace of the sampled signals on every clock edge (2,588 cycles; the testbench's own checks passed on three seeds).
2. A short Python script re-implemented the 11 properties as window checks over that trace and counted how often each one triggered. Result: no failures, with each assertion triggering 61 to 1,439 times.
3. Three deliberately broken models were run the same way. Each was caught by exactly the property it targeted: `ap_ce_hz`, `ap_rd_data` with `ap_rd_addr5_data`, and `ap_we_hz`.

This shows the intended behavior is consistent and catches seeded bugs. It does not test the SVA text itself. That needs a simulator with full SVA support or a formal tool, which is why every requirement in Stage 4 carries an explicit status.

Stage 4 New spec: reading the checker back

This is the branch in Figure 1. A new specification is written from the checker alone, without consulting the PDF, stating in plain words what each property actually requires. Comments in an SVA file say what the author meant; the read-back states what the code does, with its windows, sample counts and conditions. When the read-back and the datasheet disagree, the read-back is right about the code and the datasheet is the referee.

Translation rules

- Each `mp_` assumption becomes an environment requirement (ENV).
- Each `ap_` assertion becomes a device requirement (DEV).
- The `cp_` covers become reachability requirements (REACH).
- Each datasheet item the checker does not constrain becomes a gap (GAP), so silence is never read as agreement.
- Engineer-requested scenarios that are not in the datasheet go in their own plan (SCN).

Conventions. One cycle is one rising edge of `clk`, every 20 ns. No requirement applies while `rstn` is low. A *read window* is CE low and WE high for 5 consecutive cycles, with OE low for the last 3, judged on the 5th cycle. Each requirement is one atomic "shall" statement with a stable ID.

Status. C = compiled; S = passed the Python stand-in; M = caught a seeded mutant; SVA-sim = passed in an SVA simulator; FV = proven formally. No requirement has yet reached SVA-sim or FV.

Environment requirements (assumptions)

ID	Requirement	Datasheet ref	Implemented by	Status
REQ-ENV-001	Every write (CE and WE both low) shall last at least 3 consecutive cycles (60 ns).	tPWE 50 ns, tSCE 60 ns	`mp_wr_min_overlap`	C, S
REQ-ENV-002	The address shall not change between two consecutive write cycles.	tAW, tSA, tHA	`mp_wr_addr_stable`	C, S
REQ-ENV-003	Write data shall not change on the last cycle of a write.	tSD 30 ns, tHD 0 ns	`mp_wr_data_setup`	C, S
REQ-ENV-004	The symbolic address `watch_addr` shall stay constant for a whole proof run.	Proof technique	`mp_watch_addr_const`	C

Device requirements (assertions)

ID	Requirement	Datasheet ref	Implemented by	Status
REQ-DEV-001	When CE has been high for 3 consecutive cycles, the SRAM shall not drive DQ.	tHZCE 25 ns; truth table	`ap_ce_hz`	C, S, M
REQ-DEV-002	When OE has been high for 3 consecutive cycles, the SRAM shall not drive DQ.	tHZOE 25 ns; truth table	`ap_oe_hz`	C, S
REQ-DEV-003	When WE has been low for 3 consecutive cycles, the SRAM shall not drive DQ.	tHZWE 25 ns; truth table	`ap_we_hz`	C, S, M
REQ-DEV-004	On the 5th cycle of a read window, the SRAM shall drive DQ.	tACE 70 ns, tDOE 35 ns	`ap_rd_drive`	C, S
REQ-DEV-005	On the 5th cycle of a read window at `watch_addr`, if that address has been written, DQ shall equal the last byte written there.	tAA 70 ns; functional description	`ap_rd_data`	C, S, M

Reachability requirements (covers)

ID	Requirement	Implemented by	Status
REQ-REACH-001	Each of REQ-DEV-001 to 005 shall be triggerable: its condition shall be reachable.	`cp_*_trigger` (5)	C; triggers counted
REQ-REACH-002	Each of REQ-DEV-001 to 005 shall be satisfiable: condition and result shall occur together.	`cp_*_full` (5)	C
REQ-REACH-003	Each of REQ-ENV-001 to 004 shall leave legal behavior reachable.	`cp_*_hit` (4)	C; `cp_wr_data_setup_hit` unreachable with this testbench

Not covered (gaps)

ID	Datasheet item not covered	Reason	Proposed action
REQ-GAP-001	tOHA, tLZOE, tLZCE, tLZWE (5 ns)	Below one 20 ns cycle	Accept, or add a finer sampling clock
REQ-GAP-002	Read cycles shorter than 5 cycles (tRC 70 ns)	Outside every read window	Add a cover and an assertion for short reads
REQ-GAP-003	Address change during a read	No requirement written	Define the expected output and add a property
REQ-GAP-004	Bus contention on the bidirectional I/O (Note 19)	Bus split into `dq_i`, `dq_o`, `dq_oe`	Add a pad-level adapter and a check
REQ-GAP-005	Data integrity at more than one address per proof	One symbolic address per run	Keep `watch_addr` free in FV
REQ-GAP-006	VCC, ICC, ISB, data retention, AC load	Analog; excluded by decision	Analog or characterization sign-off

Scenario plan: address 0x0005 = 0xFE

The engineer's 0x0005 = 0xFE request is a directed scenario, not a datasheet requirement. It lives in its own plan and is left out of the general data-integrity proof, where `watch_addr` stays free. Enabled in a general proof, REQ-SCN-001 would hide every bug that corrupts data at 0x0005.

ID	Requirement	Source	Implemented by	Status
REQ-SCN-001	Every write to address 0x0005 shall carry 0xFE on its last cycle.	Engineer's request	`mp_wr_addr5_data`; `cp_wr_addr5_data_hit`	C, S
REQ-SCN-002	On the 5th cycle of a read window at 0x0005, if 0x0005 has been written, DQ shall equal 0xFE.	Engineer's request	`ap_rd_addr5_data`; `cp_rd_addr5_data_trigger`, `_full`	C, S, M

What the comparison with the PDF found. Set against the datasheet, the new spec shows where the checker is stricter (tPWE, tSD), looser (tAA, tACE, tDOE and the high-Z times, all checked at 20 ns resolution) or silent (the GAP items). REQ-ENV-001 and REQ-ENV-003 are stricter or coarser than the datasheet because of the sampling clock, and a reviewer should accept that explicitly.

Stage 4b Second checker: regenerating SVA from the new requirements

This is the branch on the left of Figure 1: the new spec goes back through Claude + RAG-SVA and comes out as a second checker. The input was `CY62256N-70_Verification_Requirements.docx` and nothing else; the header of `cy62256n70_sva.sv` states that it was generated only from that document. The framework settings were the same as for the first checker: FV mode on, decomposition stage 3, abbreviated gates, with buses exempt from per-bit checks by the engineer's decision.

What the requirements changed

Because the requirements document numbers every requirement, every property in the second checker is named after the requirement it implements, and the file header maps each ID to its label:

Requirement IDs	Labels	What they check
ENV-01 to ENV-05	`mp_env01_wr_min_window` … `mp_env05_addr05_data_fe`	Write window length, write cycle time, address stability, data setup, 0x0005 = 0xFE on writes
§2.2 (proof aids)	`mp_sym_a_rigid`, `mp_addr05_value`	A rigid symbolic address, and 0x0005 held as a free variable so helper logic stays bitwise
REQ-01 to REQ-05	`ap_req01_lowz_not_early` … `ap_req05_rd_addr05_fe`	Low-Z not early, high-Z when not reading, address-access read data, OE-access read data, 0x0005 reads 0xFE
REQ-06	via REQ-03, REQ-04 and the D(x) model	Data integrity, implied by the two read-data assertions
COV-01 to COV-10	`cp_cov01` … `cp_cov09c`, plus trigger, full and hit covers	WE-, CE- and OE-controlled cycles, back-to-back writes, each high-Z source, the 0x0005 read

The header also carries the framework's PORT-INTENT table, which the first checker left out: each port and free variable is listed with its direction and its intention (assume or obligation), so the assumption-versus-obligation split can be reviewed at a glance.

First checker versus second checker

Topic	First: `cy62256n_sram_checker.sv`	Second: `cy62256n70_sva.sv`
Source	Vendor datasheet	Verification requirements derived from the first checker
Sample period	20 ns (tAA = 4 samples)	5 ns (tAA = 14 samples)
Traceability	Labels keyed to datasheet parameters	Labels keyed to requirement IDs
PORT-INTENT table	Missing	Present
Write cycle time (tWC)	Implied by other assumptions	Its own assumption, ENV-02
Low-Z (5 ns parameters)	Dropped, below one sample	Checked, REQ-01
High-Z	Three properties (CE, OE, WE)	One property: six non-read samples
Read data	Shadow byte compared at the end of the window	Local variable captures D(a) at the start
OE-controlled read	Covered by the general read window	Its own assertion, REQ-04
Scenario covers	Trigger, full and hit only	Adds COV-01 to COV-09
Status	Elaborated; model and testbench simulated	Elaborates with 0 diagnostics in slang (module alone); not simulated

Why this matters. The second checker is the strongest test of the read-back spec. If the spec were incomplete, the second checker would be weaker than the first; instead it is more complete, because the requirements document, written at a finer 5 ns resolution, restored items the first checker had dropped (tWC, the low-Z times) and named a dedicated OE-controlled read. Where the two checkers differ, the difference points at a single requirement ID to resolve, rather than at two files to compare by eye.

Open items. The second checker's `bind` targets a `cy62256n` module with `din`, `dout` and `dout_oe` ports, so it does not bind directly to the Stage 3 model, which uses `dq_i`, `dq_o` and `dq_oe`; a thin adapter or a renamed model is needed before it can be simulated. Like the first checker, it has not yet been run in an SVA simulator or a formal tool.

Stage 5 Commercial tools

The last arrow in Figure 1. Each row of the new spec has one ID, one statement and one or more implementing labels, so it maps directly onto the plan structures that verification-management tools use.

Mapping to a verification plan. In Synopsys's Hierarchical Verification Plan (HVP), used by Verdi Planner, a plan is organized as features linked to measures (coverage, tests, assertions), with results back-annotated from the coverage database and from assertion-status data. In that model:

- **Feature:** the requirement, named by its ID, with the statement as its description and the datasheet reference and type as user-defined attributes.
- **Measure:** the `ap_`, `mp_` and `cp_` labels in "Implemented by". Assertion results and cover hits flow back onto the feature, so the plan shows each requirement's status without anyone opening the SVA.
- **Hierarchy:** ENV, DEV, REACH and GAP, plus the SCN scenario plan, become sub-plans under one top-level plan for the device. GAP features have no measures, so they stay visibly unverified.

Requirements-management and traceability tools work the same way: the ID is the trace key and the SVA label is the link to the evidence.

Interchange format. A neutral carrier is a CSV with one row per requirement:

```
req_id,type,requirement,datasheet_ref,implemented_by,method,status,notes
REQ-ENV-001,ENV,"Every write (CE and WE both low) shall last at least 3
   consecutive cycles (60 ns).","tPWE 50 ns; tSCE 60 ns",mp_wr_min_overlap,
   assume,"C,S","Stricter than tPWE at 20 ns sampling"
REQ-DEV-005,DEV,"On the 5th cycle of a read window at watch_addr, if that
   address has been written, DQ shall equal the last byte written there.",
   "tAA 70 ns",ap_rd_data,assert,"C,S,M","Keep watch_addr free in FV"
REQ-GAP-002,GAP,"Read cycles shorter than 5 cycles are not checked.",
   "tRC 70 ns",,none,open,"Add cover and assertion"
```

(Records are wrapped here for print; each is one line in the file.) A short script or the tool's own import path turns the file into the plan. Exact column names and import options depend on the tool and its version, so confirm the mapping against the vendor documentation before the first import.

Keeping it in sync. The requirement IDs belong in the SVA too, as a comment or a suffix on each label, so a rename on either side shows up as a broken trace rather than a silent mismatch.

Limits and next steps

- **The SVA has not been executed.** It compiles cleanly, but no SVA simulator or formal tool has run it.
- **One author.** Model, checker, stand-in and read-back come from one reading of the datasheet. The line-item spec reduces that risk by putting every claim beside its datasheet reference for a human reviewer; it does not remove it.
- **Covers not measured.** Only triggers were counted.
- **Sampling resolution.** A 20 ns clock coarsens some limits and drops the 5 ns parameters (REQ-GAP-001).

1. Run the testbench with the checker in Questa, VCS or Xcelium with assertion and cover reporting, and compare with the Python stand-in.
2. Run a formal tool with `watch_addr` free and the SCN plan left out, for a general data-integrity proof.
3. Close REQ-GAP-002 and REQ-GAP-003 with new properties and new IDs.
4. Import the CSV into the verification plan and back-annotate results, so each requirement shows live status.
5. Remove the trace debug block from the testbench before release.

Conclusion

The process in Figure 1 has a short forward path and an essential return path. Going forward, Claude under the RAG-SVA rules turns a vendor PDF into a checker, an RTL model and a testbench, and the engineer's effort goes into decisions about the device rather than into writing assertions. Coming back, the checker is read into a new spec of numbered, referenced, status-tagged requirements, which commercial tools can import and track to sign-off.

That branch from the checker is the point. It turns AI-generated SVA into a statement of requirements that a team can check against the original datasheet, line by line, in the tools it already uses.

---

[i] infineon-cy62256n-256-kbit-32-k-8-static-ram-datasheet-en.pdf
[ii] cy62256n_sram_checker_2.sv from Infineon datasheet, 20 ns (tAA = 4 samples),
[iii] cy62256n_model.sv
[iv] RAG-SVA is distributed with the book *Knowledge-Driven AI for SystemVerilog Assertions*.
Commercial use requires a commercial license; the license is free for educational use.
[v] CY62256N-70_Verification_Requirements.docx

https://SystemVerilog.us/rqmts_to_sva_to_fixedRQMTs.tar.gz
Created with:
tar -czvf rqmts_to_sva_to_fixedRQMTs.tar.gz paper
a paper
a paper/CY62256N-70_Verification_Requirements.docx
a paper/CY62256N-70_Verification_Requirements.pdf
a paper/cy62256n70_sva-1.sv
a paper/cy62256n_model.sv

```
a paper/cy62256n_simulation_block_diagram.png
a paper/cy62256n_sram_checker_2.sv
a paper/HIGH_Z_ISSUE.png
a paper/infineon-cy62256n-256-kbit-32-k-8-static-ram-datasheet-en.pdf
a paper/ram_cmd.txt
a paper/replay70.py
a paper/SCORECARD.md
a paper/tb_cy62256n.sv
a paper/tb_cy62256n70-2.sv
a paper/V4_Vendor_Datasheet_to_Traceable_Requirements_The_CY62256N_SRAM.docx
```