Same Design, New Node

What Must Change in the SVA and the RTL When a Verified 5 nm Design Moves to 3 nm — and What an AI Assistant Can and Cannot Re-Derive from Technology Data

---

Ben Cohen

Kais Chibani

Cohen Publishing

systemverilog.us

September 2026

Abstract

A design that works in a 5 nm-class technology arrives with a set of assertions and an RTL description whose numbers were chosen for that technology: window bounds such as `##[1:3]`, exact latencies such as `##3`, pipeline depths, counter limits, synchronizer depths, and memory read latencies. Moving to a 3 nm-class technology invites two mistakes. The first is to assume nothing changes because the RTL is technology independent. The second is to scale every delay by a single factor. Both are wrong, for different reasons.

This paper explains what changes and what does not. Cycle-based protocol rules do not change. Time-based limits do not change, but the number of cycles they allow does, whenever the clock changes. Implementation latencies change only when the pipeline depth changes, and the depth changes only when the clock, the path delays, or the per-stage overhead move enough to cross an integer step. Gate delay, wire delay, register overhead, and variation margins scale by different amounts, so a single factor cannot describe the migration. We give the arithmetic, a catalogue of the numbers that move, a technology data sheet that captures what is needed, and a decision rule for keeping or re-pipelining the design.

We then answer the practical question: if the technology data is supplied, can an AI assistant rewrite the assertions? For the mechanical part, yes, provided every fixed integer is tagged with its basis and the delays come from timing reports, not from a guessed ratio. A complete worked example, in integer picoseconds and integer percentages, takes a request, ready, acknowledge block from a 5 nm-class technology to a 3 nm-class technology at the same clock and at a faster clock. It is simulated in every configuration, including three deliberately mismatched ones, and it shows that the assertions and the RTL stay consistent only when both derive from one parameter block. The limits are equally important: the assistant does not run synthesis or static timing analysis, so it cannot produce the delays it needs, and it cannot decide whether a latency change is acceptable to the rest of the system.

All technology numbers in this paper are assumptions chosen for the examples. They are not process data.

Keywords: technology migration, 5 nm, 3 nm, SystemVerilog Assertions, SVA, pipeline depth, static timing analysis, setup time, clock-to-q, timing closure, AI-assisted verification.

Contents

1. The situation

A team has a block that works in a 5 nm-class technology. It has an RTL description, a set of assertions, and a constraints file that closes timing. The target is now a 3 nm-class technology. Four questions follow immediately.

1. What must change in the RTL?
2. What must change in the assertions?
3. If the technology numbers (setup times, path delays, and so on) are supplied to an AI assistant, can it rewrite the assertions?
4. What can go wrong?

1.1 The running example

The running example is a request, ready, acknowledge handshake. A request pulse `req` produces a ready pulse `rdy`, which produces an acknowledge pulse `ack`. The specification gives time limits, not cycle counts: `rdy` within 8 ns of `req`, and `ack` within 12 ns of `rdy`. At the 5 nm design point the block runs with a 2 ns clock, with three register stages on the req to rdy path and five on the rdy to ack path. The assertions for that design point are:

```
$rose(req) |-> strong(##[1:4] rdy);    $rose(req) |-> strong(##3 (rdy ##1 !rdy));
$rose(rdy) |-> strong(##[1:6] ack);    $rose(rdy) |-> strong(##5 (ack ##1 !ack));
```

In these lines, `$rose(x)` marks the cycle where `x` rises, `|->` means "if the left side happens, the right side must follow", `##[1:4] rdy` means `rdy` occurs between one and four cycles later, `##3` means exactly three cycles later, and `strong(...)` requires that the event really occurs and is not merely "not yet failed" when the trace ends. The first line of each pair is the **specification window**: it comes from the time limit and is what a user of the block cares about. The second is the **implementation check**: it states the exact latency of this pipeline and that the pulse is one cycle wide.

Nothing in the four lines says why the numbers are 4, 3, 6 and 5. That is the root of every problem in this paper.

1.2 Answers in brief

- The RTL is technology independent in function, but its pipeline depth, its counters, its memory wrappers, and its synchronizers are technology dependent in cycle counts.
- The assertions change only where a number is a consequence of the clock, the delays, or the depth.
- An assistant can re-derive those numbers automatically when it is given the basis of each number and the physical data behind it. It cannot derive the physical data.
- The main risks are scaling the delays by one factor, forgetting the non-datapath latencies, letting the RTL and the assertions diverge, and treating an estimate as a measurement.

1.3 Conventions

- **Integers only.** Times are integer picoseconds (ps). Scaling between technologies uses integer percentages. Divisions are written with explicit rounding: **up** (ceiling) for delays and stage counts, **down** (floor) for the number of cycles that a time limit allows. The RTL and the assertions use the same integer arithmetic, so the paper and the code agree exactly.
- **Assumption numbers.** Every technology number (delays, register overhead, percentages, voltages, memory times) is an assumption made for the examples and is labelled as such. Replace them with values from your own timing reports.
- **Classes of latency.** A *cycle-based* latency is a protocol rule (for example, "three beats") and does not depend on the clock. A *time-based* latency is a limit in time (for example, "12 ns") and its cycle count depends on the clock. An *implementation* latency is the exact number of cycles the pipeline as built takes.

2. Why a node change is not a scale factor

Moving from one process generation to the next changes several quantities, and they do not change by the same amount. Foundry headlines quote speed, power and density gains for reference designs under stated conditions. Those numbers describe the process, not your paths. What matters here is the breakdown of a path into parts that scale differently.

2.1 Cell delay and wire delay

A path delay is the sum of the delays of the cells on the path and of the wires between them. Cell delay generally improves with a newer node. Wire delay does not improve at the same rate, because narrower and more closely spaced wires have higher resistance per unit length, and a larger share of a path delay is spent in interconnect. Both figures appear separately in a timing report, so the correct estimate of a migrated path is

D_new = (g × D_cell) / 100 + (w × D_net) / 100

where g is the cell-delay percentage and w the net-delay percentage (for example 80 and 95), both measured for your library and your routing, and w is typically closer to 100 than g. A single percentage applied to the total path delay hides this and is the most common source of over-optimistic pipeline plans.

2.2 The register overhead is not one number

The per-stage overhead in our model, t_ovh, is the sum of the clock-to-q delay of the launching register, the setup time of the capturing register, and the timing margins: clock skew, clock uncertainty (which includes jitter), and derating for on-chip variation. The first two shrink with a faster technology. The margins often do not, and can grow as a fraction of the period because variation is larger at advanced nodes and because uncertainty and jitter are absolute times that do not shrink with the clock period. The consequence, which the worked example reproduces, is that the overhead can barely change even when every cell is faster.

Component	Direction with a newer node	Depends on
Clock-to-q	Usually smaller	Flop cell, Vt flavour, slew, corner
Setup time	Usually smaller	Flop cell, Vt flavour, slew, corner
Clock skew	Design dependent	Clock tree, floorplan
Clock uncertainty and jitter	Often not smaller in absolute terms	PLL, supply noise, methodology
On-chip variation derate	Often larger	Node, library, signoff methodology
Hold time	Design dependent	Not part of cycle counts

2.3 Corners, threshold flavours, voltage

A newer technology often brings a different set of signoff corners, more threshold-voltage flavours, and different supply voltages. The delay numbers used for pipeline planning must state the corner and the flavour. The slow corner limits setup and therefore the stage count; the fast corner limits hold and does not change cycle counts. At low supply voltage some technologies show a temperature-inversion effect, where delay is worse at low temperature, which changes which corner is the worst case. A path delay quoted without its corner is not usable data.

2.4 A different foundry means a different library

A migration from a 5 nm-class technology of one foundry to a 3 nm-class technology of another also changes the cell library, the flop and clock-gate cell names, the memory compilers, the standard-cell characterisation, and the constraint and back-annotation formats. Even within one foundry, the flops and clock-gating cells are new cells with new timing. The RTL may reference none of them, but the synthesis, timing, and equivalence flows do.

2.5 Memories and other macros

Memory macros and hard IP have access times set by their compilers and datasheets. The speed of SRAM macros often scales less favourably than the speed of logic. A memory read that took one cycle at the old clock may need two at a faster new clock, or its wrapper may need an output register. This is a latency change in the RTL and therefore in the assertions on the memory interface, and it is independent of the logic path delays.

2.6 Summary of what moves

Quantity	Moves with the node?	Moves with the clock?
Protocol rules counted in cycles	No	No
Time-based limits in the spec	No	Cycles allowed change
Cell delay	Yes	No
Net delay	Somewhat	No
Register overhead	Partly	No
Stage count N	Through delay and overhead	Yes
Memory access time	Yes, differently from logic	Cycles change
Synchronizer depth	Through metastability data	Through frequency

3. Two goals, one decision

A migration has one of two goals, sometimes both.

- **Goal A: same clock, better power or area.** The technology is faster, so paths have slack. The slack can be spent on smaller cells or lower-leakage flavours, or on fewer pipeline stages.
- **Goal B: higher clock.** The technology gain is spent on frequency. Depth may stay the same or grow.

The decision that determines how much changes is whether the cycle-level behaviour of the block is a contract with its neighbours.

3.1 Keep the depth

If software, a neighbouring block, or a certification depends on the block's latency in cycles, the safe migration keeps N. The RTL does not change at all. The assertions do not change at all. What changes is the timing report, which shows more slack, and the physical implementation, which uses the slack. This is the lowest-risk migration and the one to choose when Goal A is sufficient.

3.2 Reshape the depth

If the latency is only bounded by the specification window, the depth can be re-derived from the new data:

$N_{new} = \text{ceil}(D_{new} / (T_{clk} - t_{ovh\_new}))$,

and it is legal only if $1 \le N_{new} \le MAX$, where $MAX = \text{floor}(T_{spec} / T_{clk})$. This changes the RTL and the exact assertions together. The spec window changes only if the clock changed.

3.3 The rule

Keep N if a neighbour depends on it or the slack is sufficient for the goal. Re-derive N if the clock changed, if slack is negative, or if latency reduction has been requested. In either case the assertions and the RTL are updated from the same parameter block, and the choice is recorded in the specification, so it is not an accident of who edited which file.

4. The numbers that move

The catalogue below lists the numbers in a typical design that depend on the clock or the technology. In an existing design they appear as literals in assertions and RTL, which is why a migration is error prone.

Category	Example	Basis	What moves it	Data needed
Protocol cycles	"ack the cycle after req"	CYCLES	Nothing	None
Time-based window	`##[1:4] rdy`	TIME	Clock period	Limit in ps, new clock
Exact pipeline latency	`##3 (rdy ##1 !rdy)`	Implementation	Depth N	Path delay, overhead, clock
Pulse width	`##1 !rdy`	CYCLES	Nothing	None
Timeouts and watchdogs	"no response in 1000 cycles"	TIME	Clock period	Limit in ps
Counters derived from time	"wait 100 ns" coded as a cycle count	TIME	Clock period	Time in ps
Memory read latency	`##2 data_valid`	Macro	Macro access time, clock	Datasheet access time
Synchronizer latency	`##[2:3] sync_ack`	Implementation	Metastability data, frequency	MTBF target, technology data
Clock-gate enable timing	enable to gated clock	Implementation	Gate cell timing, clock	Cell timing
Reset release sequences	"release after 16 cycles"	TIME	Clock, reset tree	Reset tree delay
Environment spacing	`!req[*MAX_TOT]`	Derived	MAX values	Time limits
Multicycle exceptions	2-cycle setup path	Implementation	Clock, delay	Timing report

Two items deserve emphasis.

Counters derived from time. A constant such as `localparam WAIT = 10` that was chosen as 100 ns at 100 MHz is a time-based value written as a cycle count. The value must be re-derived from the original time for the new clock, and the assertion that checks it must follow. An RTL description that computes `WAIT` from a time parameter does not need to be edited; one that hard-codes it does.

Synchronizers. The depth of a clock-domain-crossing synchronizer is chosen to meet a mean time between failures target, and that depends on the clock frequencies and the technology's metastability parameters. A faster clock, or a different technology, can change the depth, and the depth is a latency that appears in the handshake assertions across the crossing.

5. The migration arithmetic

5.1 The inputs

For each path: the cell delay and the net delay from a timing report at the worst setup corner, and the register overhead broken into its components. For each requirement: its basis (TIME or CYCLES) and limit. For the block: the new clock period.

5.2 The steps

1. Recompute each path delay, preferably from a synthesis or post-route timing report for the new technology, or as an estimate using D_new = (g × D_cell) / 100 + (w × D_net) / 100.
2. Recompute the overhead from its components.
3. Compute the usable time per stage, T_use = T_clk − t_ovh.
4. Compute N = ceil(D / T_use) for each path and MAX = floor(T_spec / T_clk) for each limit.
5. Check 1 ≤ N ≤ MAX. If not, report infeasible.
6. Apply the keep or reshape rule of Section 3.3.
7. Re-derive every assertion bound and every RTL constant from the block.

5.3 Why floor for limits and ceiling for stages

The two roundings go in opposite directions, and mixing them up is a common mistake.

- **Stages: round up.** A path of delay D split into N equal stages puts about D / N of logic in each stage, and each stage must fit in the usable time T_use. So D / N ≤ T_use, which gives N ≥ D / T_use. A fractional stage does not exist, and rounding down would violate the clock.
- **Cycles allowed by a limit: round down.** A limit of T_spec means the latency must never exceed T_spec. With a clock T_clk, the largest number of whole cycles that fits is floor(T_spec / T_clk). For a limit of 13000 ps at a 4000 ps clock this is 3 cycles (12000 ps), not 4 cycles (16000 ps). Rounding up would silently loosen the requirement by 3000 ps.

In the examples of this paper the limits (8000 ps and 12000 ps) divide the clock periods used exactly, so both roundings would agree for the limits. That is the special case, not the rule.

5.4 What cell delay and net delay represent

Both terms in D = D_cell + D_net are parts of the same path delay.

- **Cell delay** is the time spent inside the gates on the path (AND, OR, XOR, mux, adder cells, buffers).
- **Net delay** is the time spent in the wires between those gates.
- **D** is the total delay of the logic between two register boundaries if it were not pipelined at all. It excludes the register overhead (clock-to-q and setup), which is counted separately in t_ovh.

A timing report lists the path arc by arc, each with an incremental delay. Adding the cell arcs and the net arcs separately gives D_cell and D_net. The following path is an assumed illustration for the req to rdy logic at 5 nm:

Step	Cell (ps)	Net after it (ps)
Input buffer	200	100
AND-OR gate	350	150
Compare (XOR chain)	550	250
Mux	400	200
Adder carry	600	180
Output buffer	300	120
Total	2400	1000

So D_RDY = 2400 + 1000 = 3400 ps. For a real design, take the cell and net sums from the post-synthesis or post-route timing report at the signoff corner. The rdy to ack path (5600 + 2400 = 8000 ps) is derived the same way.

The two sums are kept apart because cells and wires scale differently. Applying an assumed 80 percent to the cells and 95 percent to the nets gives (80 × 2400) / 100 + (95 × 1000) / 100 = 1920 + 950 = **2870 ps**. Applying the cell percentage to the total instead would give (80 × 3400) / 100 = 2720 ps and understate the wire share.

5.5 The assumed technology data

All numbers in this table are assumptions chosen for the example.

	5 nm-class	3 nm-class
Clock-to-q	150 ps	120 ps
Setup	50 ps	40 ps
Clock skew	50 ps	50 ps
Uncertainty and jitter	80 ps	90 ps
On-chip variation derate	70 ps	80 ps
Overhead t_ovh	400 ps	380 ps
req to rdy: cell + net	2400 + 1000 = 3400 ps	1920 + 950 = 2870 ps
rdy to ack: cell + net	5600 + 2400 = 8000 ps	4480 + 2280 = 6760 ps
Cell-delay percentage of the 5 nm value	100	80
Net-delay percentage of the 5 nm value	100	95

The overhead falls by only 20 ps out of 400 (5 percent), although clock-to-q and setup each fall by about 20 percent, because the margin terms (uncertainty and variation) grow by 20 ps.

5.6 Applying the steps

The time limits are 8000 ps for req to rdy and 12000 ps for rdy to ack.

Case	Clock	Usable per stage	MAX (rdy, ack)	N (rdy, ack)	Feasible
5 nm baseline	2000 ps	1600 ps	4, 6	3, 5	Yes (ack path has 0 ps slack)
3 nm, same clock	2000 ps	1620 ps	4, 6	2, 5	Yes
3 nm, faster clock	1400 ps	1020 ps	5, 8	3, 7	Yes

Three observations. First, **the ack path stays at five stages at the same clock** even though the logic is faster, because four stages would need 1690 ps of logic per stage plus 380 ps overhead, which is 70 ps over the 2000 ps clock period. Quantization protects the old depth. Second, only the req to rdy path shortens. Third, the time-based windows are unchanged at 2000 ps, and they change at 1400 ps because the clock changed, not because the technology did.

Figure 1 shows the stage count against the clock period for both technologies on both paths. Where the stage curve is above the dashed line, the target is infeasible. The 3 nm curve lies on or below the 5 nm curve, and the region of feasible clocks is larger, but the curves have the same stepped shape, which is why an estimate that lands near a step is unreliable.

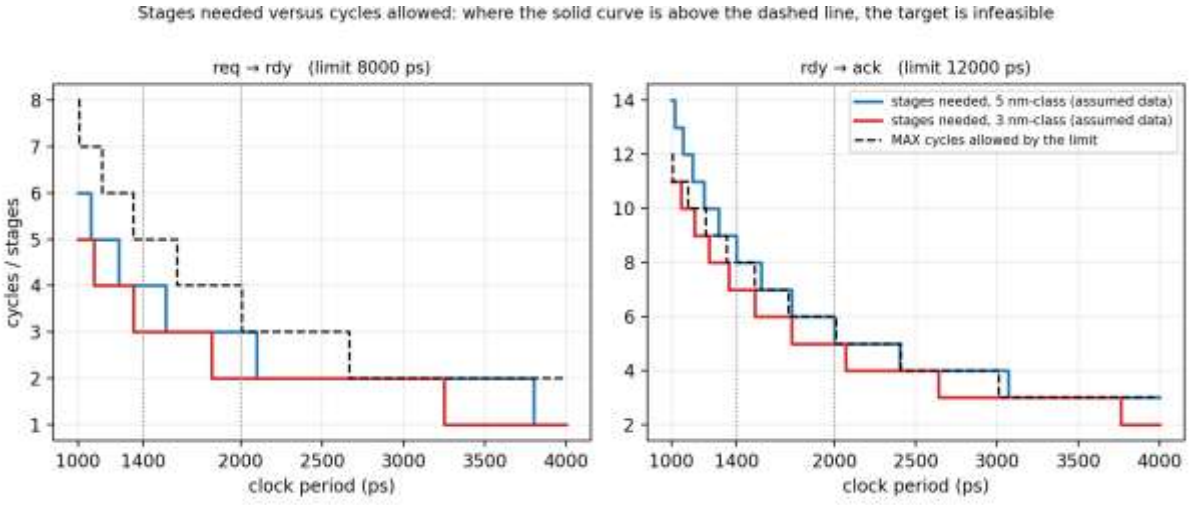

Figure 1. Stages needed (solid) versus cycles allowed by the spec (dashed) for the two paths of the example, for the assumed 5 nm-class and 3 nm-class data of Section 5.5.

5.7 Slack: D, N, and the equal-split formula

In the slack formula below, **D** is the total combinational delay of the path if it were not pipelined (the cell delay plus net delay above: 3400 ps for req to rdy and 8000 ps for rdy to ack at 5 nm). **N** is the number of register stages that logic is split into (3 and 5 in the 5 nm example).

"Equal split" means each of the N stages gets the same share of the logic, so the logic per stage is L = ceil(D / N), rounded up to stay conservative. For one stage:

slack = T_clk − (t_ovh + L)

- **T_clk** is the time the stage has.
- **t_ovh + L** is the time it uses: the register overhead plus its share of the logic.
- **slack** is what is left over. Positive means the stage meets timing; negative means it fails.

Worked example, 5 nm at 2000 ps:

- req to rdy: L = ceil(3400 / 3) = 1134 ps, so slack = 2000 − (400 + 1134) = **466 ps**.
- rdy to ack: L = ceil(8000 / 5) = 1600 ps, so slack = 2000 − (400 + 1600) = **0 ps**.

The zero on the ack path means five stages is the minimum depth that works. Any extra skew or wire delay would push it negative.

Relation to the stage-count formula. N = ceil(D / (T_clk − t_ovh)) is this same inequality solved for N. Setting slack ≥ 0 gives D / N ≤ T_clk − t_ovh, so N ≥ D / (T_clk − t_ovh), and N is rounded up.

The caveat. An equal split is an idealization. Real logic does not divide evenly, so the actual slack is set by the worst stage, not the average. Use the equal-split number as a planning estimate, and take the real slack from the timing report.

Case	rdy path slack	ack path slack
5 nm at 2000 ps, depth (3, 5)	466 ps	0 ps
3 nm at 2000 ps, depth reshaped (2, 5)	185 ps	268 ps
3 nm at 2000 ps, depth kept (3, 5)	663 ps	268 ps
3 nm at 1400 ps, depth reshaped (3, 7)	63 ps	54 ps
3 nm at 1400 ps, depth kept (3, 5)	63 ps	−332 ps

The last row is the important one. If the design is taken to the faster clock without adding stages, the specification window and the old assertions may still pass, but the ack path violates setup by 332 ps. Only timing analysis reports that.

6. Can an assistant rewrite the assertions?

6.1 The short answer

Yes, for the arithmetic; no, for the data. Given the basis of each bound and the physical inputs, the conversion is deterministic: floor for windows, ceiling for depths, a feasibility check, and a change log. What the assistant cannot do is produce the timing data. It does not run synthesis or static timing analysis, so the path delays and overhead must come from the team's tools.

6.2 Three levels of automation

Level	What is converted	What must be supplied
1	Time-based windows and derived spacing constants	Old and new clock, the TIME limit or CYCLES flag of each bound
2	Exact latencies and RTL pipeline depth	Level 1, plus per-path cell and net delays at the signoff corner, register overhead components, freedom to add stages, keep or reshape decision
3	Non-datapath latencies	Level 2, plus memory access times, synchronizer data, timer and reset specifications, clock-gate timing, multicycle exceptions

6.3 Estimation mode and its limit

If only percentages are available, an assistant can compute an estimate, but it should report a range and flag it. Suppose the cell-delay percentage is known only to lie between 75 and 85 and the net-delay percentage between 90 and 105. The delay range gives a range of stage counts, computed with integer arithmetic:

Path and clock	Delay range in ps (low, mid, high)	N (low, mid, high)	Verdict
req to rdy at 2000 ps	2700, 2870, 3090	2, 2, 2	Stable: 2
rdy to ack at 2000 ps	6360, 6760, 7280	4, 5, 5	Straddles a step
req to rdy at 1400 ps	2700, 2870, 3090	3, 3, 4	Straddles a step
rdy to ack at 1400 ps	6360, 6760, 7280	7, 7, 8	Straddles a step

For example, the low delay for req to rdy is (75 × 2400) / 100 + (90 × 1000) / 100 = 1800 + 900 = 2700 ps, and the high delay is (85 × 2400) / 100 + (105 × 1000) / 100 = 2040 + 1050 = 3090 ps.

Three of the four estimates straddle an integer step. That is a typical outcome, and the right response is to leave those exact bounds marked "needs timing data" rather than pick a value. A stage count decided from an estimate that straddles a step is a coin toss written as an assertion.

6.4 What the assistant does with missing data

The assertion-generation framework described in the authors' book (reference 1) halts on ambiguity instead of guessing. In practice, an unknown basis is asked back, an unknown delay leaves the exact latency flagged, and an infeasible combination is reported and not widened. The output includes a change log that lists each line, its old value, its new value, and the data that justified it.

6.5 What the assistant should also return

- The four properties in terms of parameters, not literals, so the next migration is a data edit.
- The new values for each target.
- A per-path feasibility check.
- A list of RTL constants that changed (stage counts, counters, memory wrappers) so that the RTL and the assertions cannot diverge.
- A list of items it could not determine.

7. Issues and risks

Issue	Symptom	Mitigation
One scale factor applied to total path delay	Over-optimistic depth, late timing failure	Split into cell and net delay; use measured values
Overhead assumed to scale with the cells	Zero or negative slack at signoff	Rebuild overhead from components, including margins
Estimate near an integer step	Wrong depth chosen	Report a range; mark undecided; get timing data
Window passes, depth wrong	Assertions silent, STA fails	Assert the exact latency as well as the window
Assertions re-derived, RTL not	Exact assertion fails on a correct RTL	One parameter block for both
RTL re-pipelined, assertions not	Exact assertion fails after a correct change	One parameter block for both
Time-based counters hard-coded in RTL	Wrong timeouts at the new clock	Derive counters from a time parameter
Memory latency unchanged in the wrapper	Data valid too early or late	Recompute from macro access time
Synchronizer depth unchanged	Reduced MTBF at a faster clock	Recompute MTBF; update depth and assertions
Clock-gate enable path critical	Timing failure on the enable	Pipeline the enable; update the latency contract
Multicycle exception not re-derived	Silent silicon failure	Mirror every exception with an assertion
Depth changed but neighbours rely on it	System-level failure	Decide keep or reshape explicitly
Retiming moves the register carrying `rdy`	Internal assertions meaningless on netlist	State which latencies are external; use equivalence checking
Corner not stated	Under-counted stages	Record the corner with every delay
Constraints file period and package period differ	Two clocks, one truth	Generate one from the other
Neglecting gate-level timing simulation	RTL assertions pass, gate-level does not	Keep gate-level simulation with back-annotated delays in the plan

7.1 Estimates as measurements

The single **most damaging habit is treating an estimate as a measurement**. An assertion with a number in it looks authoritative. The number should carry its provenance in a comment or, **better, come from a block that names the source: estimate, post-synthesis, or post-route.**

7.2 Verification plan for the migrated design

1. Regenerate the parameter block for the new node and clock.
2. Rebuild the RTL and the assertions from it.
3. Run the feasibility check.
4. Run simulation and formal with the new assertions, including coverage.
5. Run synthesis and static timing analysis at the signoff corners; confirm slack.
6. If slack is negative or thin, change the block, not the files, and repeat.
7. Run equivalence checking between RTL and netlist if retiming or optimisation may have moved registers.
8. Run gate-level simulation with back-annotated delays on a representative subset.
9. Record the technology data and its provenance with the release.

8. A complete worked example

This section follows the handshake block from a specification to a 5 nm baseline and then to two 3 nm configurations, using only integers. All technology numbers are the assumptions of Section 5.5 and Appendix A.

8.1 The block and its specification

Signal	Direction	Role	Notes
`clk`	input	clock	rising edge
`reset_n`	input	reset	active-low
`req`	input	environment	one-cycle pulse
`rdy`	DUT-internal	design obligation	one-cycle pulse; observed by the assertion module
`ack`	output	design obligation	one-cycle pulse

Requirements:

- **REQ-1 (time-based).** `rdy` shall follow `req` within 8000 ps, and not earlier than the next cycle.
- **REQ-2 (time-based).** `ack` shall follow `rdy` within 12000 ps, and not earlier than the next cycle.
- **REQ-3 (implementation).** The design has a fixed latency: exactly N_RDY register stages from `req` to `rdy` and N_ACK from `rdy` to `ack`, each within its REQ-1 or REQ-2 window.
- **REQ-4 (pulses).** `req`, `rdy` and `ack` are each exactly one clock cycle wide.
- **REQ-5 (single outstanding).** After each `req` pulse, `req` stays low for MAX_RDY + MAX_ACK cycles, so the acknowledge of one transaction always precedes the next request. This is a constant, because the assumption may not refer to the design's own outputs.
- **REQ-6 (reset).** While `reset_n` is low, all pulses are inactive and the pipeline is flushed; no `rdy` or `ack` results from a request issued before reset.
- **REQ-7 (feasibility).** If N exceeds MAX for any path at any clock, that target is infeasible. It is reported and no bound is widened.
- **REQ-8 (single source).** The clock, limits, delays and overhead are written once, in a parameter block, and both the RTL and the assertions derive from it.

8.2 Baseline: 5 nm, 2000 ps

Using the data of Section 5.5 and integer arithmetic:

Step	req to rdy	rdy to ack
Time limit	8000 ps	12000 ps
MAX = floor(limit / 2000)	4	6
D = cell + net	2400 + 1000 = 3400 ps	5600 + 2400 = 8000 ps
T_use = 2000 − 400	1600 ps	1600 ps
N = ceil(D / 1600)	ceil(3400 / 1600) = 3	ceil(8000 / 1600) = 5
Feasible: 1 ≤ N ≤ MAX	3 ≤ 4 yes	5 ≤ 6 yes
Achieved latency N × 2000	6000 ps	10000 ps
Spacing MAX_TOT = MAX_RDY + MAX_ACK	4 + 6 = 10 cycles	

The four assertions and the assumption for this design point are shown in Section 1.1 and in Appendix D, with `MAX_RDY = 4`, `MAX_ACK = 6`, `N_RDY = 3`, `N_ACK = 5`, `MAX_TOT = 10`.

8.3 Migration to 3 nm at the same clock, 2000 ps

Migrated path delays (integer percentages 80 and 95):

- req to rdy: (80 × 2400) / 100 + (95 × 1000) / 100 = 1920 + 950 = 2870 ps.
- rdy to ack: (80 × 5600) / 100 + (95 × 2400) / 100 = 4480 + 2280 = 6760 ps.

Overhead: 120 + 40 + 50 + 90 + 80 = 380 ps, so T_use = 2000 − 380 = 1620 ps.

Step	req to rdy	rdy to ack
MAX = floor(limit / 2000)	4	6
N = ceil(D / 1620)	ceil(2870 / 1620) = 2	ceil(6760 / 1620) = 5
Feasible	2 ≤ 4 yes	5 ≤ 6 yes
Slack, L = ceil(D / N)	2000 − (380 + 1435) = 185 ps	2000 − (380 + 1352) = 268 ps

Two choices follow, and both are legal.

- **Keep the depth (3, 5).** Nothing in the RTL or the assertions changes. The slack grows to 663 ps on the req to rdy path and 268 ps on the ack path, and the physical implementation can spend it on power or area.
- **Reshape the depth (2, 5).** The req to rdy path loses one stage. Only one assertion number changes (`##3` becomes `##2`), together with the RTL depth. The ack path stays at five stages, because four stages would leave a slack of 2000 − (380 + 1690) = −70 ps.

The decision follows the rule of Section 3.3: keep the depth if a neighbour depends on the latency; reshape it if latency reduction is wanted.

8.4 Migration to 3 nm at a faster clock, 1400 ps

Step	req to rdy	rdy to ack
MAX = floor(limit / 1400)	floor(8000 / 1400) = 5	floor(12000 / 1400) = 8
T_use = 1400 − 380	1020 ps	1020 ps
N = ceil(D / 1020)	ceil(2870 / 1020) = 3	ceil(6760 / 1020) = 7
Feasible	3 ≤ 5 yes	7 ≤ 8 yes
Slack, L = ceil(D / N)	1400 − (380 + 957) = 63 ps	1400 − (380 + 966) = 54 ps
Achieved latency N × 1400	4200 ps	9800 ps
Spacing MAX_TOT	5 + 8 = 13 cycles	

Here the depth cannot be kept: with the old (3, 5), the ack path would use L = ceil(6760 / 5) = 1352 ps and have a slack of 1400 − (380 + 1352) = **−332 ps**. The ack path must go to seven stages. The windows change because the clock changed.

8.5 What changes in the files

Item	5 nm, 2000 ps	3 nm, 2000 ps, keep	3 nm, 2000 ps, reshape	3 nm, 1400 ps
req to rdy window	`##[1:4]`	`##[1:4]`	`##[1:4]`	**`##[1:5]`**
req to rdy exact	`##3`	`##3`	`##2`	**`##3`**
rdy to ack window	`##[1:6]`	`##[1:6]`	`##[1:6]`	**`##[1:8]`**
rdy to ack exact	`##5`	`##5`	`##5`	**`##7`**
Spacing, req low for	10 cycles	10 cycles	10 cycles	**13 cycles**
RTL depth (rdy, ack)	3, 5	3, 5	2, 5	**3, 7**
Edits to the RTL and assertion files	none	none	none	none

The last row is the point of the parameter block. The assertion file and the RTL are identical in all four columns; only two compile-time settings differ: the technology selection and the clock period. For example, `+define+HS_TECH_3NM +define+HS_CLK_PS=1400` produces the last column.

8.6 The non-datapath items, with the same integers

The handshake pipeline is only part of a real block. Applying the same arithmetic to the assumed non-datapath data of Appendix A:

Item	5 nm, 2000 ps	3 nm, 2000 ps	3 nm, 1400 ps
Memory read latency, ceil((access + capture margin) / clock)	ceil((1500 + 250) / 2000) = 1	ceil((1400 + 260) / 2000) = 1	ceil((1400 + 260) / 1400) = 2
Minimum 100 ns wait, ceil(100000 / clock)	50 cycles	50 cycles	72 cycles
Reset hold 200 ns, ceil(200000 / clock)	100 cycles	100 cycles	143 cycles
Synchronizer depth (assumed from an MTBF analysis)	2	2	3

At 1400 ps, four latencies that a datapath-only migration would miss change at once: the memory read (an assertion such as `##1 data_valid` becomes `##2`), two time-derived counters, and the synchronizer depth. The memory latency changes even though the logic got faster, because the assumed macro access time improves by only 100 ps.

Note on the values. The memory access times, synchronizer depths, and timer values are assumptions for the example; they are derived here by arithmetic only and are not part of the simulated design.

8.7 What was run

The RTL of Appendix C was simulated with the package of Appendix B for each case. The sampled trace was checked against the assumption and the four assertions of Appendix D by a cycle-accurate implementation of their semantics (attempt start, `disable iff (!reset_n)`, and strong completion inside the trace). A commercial SVA simulator or formal tool was not used, so the assertion file should be run in the team's own tool too. The stimulus includes tightest-legal request spacing, longer gaps, and a reset applied while a request is in flight.

Case	RTL depth	Assertion depth	Windows	Exact rdy	Exact ack
C1: 5 nm, 2000 ps, derived	(3, 5)	(3, 5)	pass	pass	pass
C2: 3 nm, 2000 ps, derived (reshape)	(2, 5)	(2, 5)	pass	pass	pass
C3: 3 nm, 1400 ps, derived	(3, 7)	(3, 7)	pass	pass	pass
C4: 3 nm, 2000 ps, RTL not updated	(3, 5)	(2, 5)	pass	**fail**	pass
C5: 3 nm, 2000 ps, assertions not updated	(2, 5)	(3, 5)	pass	fail	pass
C6: 3 nm, 1400 ps, RTL not re-pipelined	(3, 5)	(3, 7)	pass	pass	**fail**
C7: 3 nm, 2000 ps, keep strategy	(3, 5)	(3, 5)	pass	pass	pass

Table 1. Results. C1 to C3 and C7 are consistent configurations. C4 to C6 are deliberate mismatches. The single-outstanding assumption held and no orphan pulses followed the reset in the passing cases.

Reading the table:

- **C1 to C3** confirm that one parameter block re-derives RTL and assertions consistently for both technologies and both clocks.
- **C4 and C5** show that a mismatch in either direction is caught by the exact assertion and not by the window, so the exact assertion is what keeps the RTL and the assertions honest.
- **C6** shows the exact assertion catching a stale depth at the faster clock. The slack table of Section 5.7 shows the same design failing setup on the ack path by 332 ps, so both the assertion and the timing analysis flag it, for different reasons.
- **C7** is the keep strategy. Nothing in the RTL or the assertions changes, and the design gains slack.

9. Checklist

9.1 Before migrating

- Every fixed integer in the assertions and the RTL is tagged TIME, CYCLES, or implementation.
- The time limit or the delay behind each number is recorded.
- The keep or reshape decision is made and written into the specification.
- The technology data sheet (Appendix A) is filled in, with source and corner.

9.2 During migration

- The parameter block is regenerated; no literal is edited by hand.
- The non-datapath latencies (memory, synchronizers, timers, clock gates, reset, multicycle exceptions) are reviewed.
- Feasibility is checked for every path.

9.3 After migration

- Timing analysis confirms slack with margin at the signoff corners.
- Simulation, formal, and coverage pass on the regenerated assertions.
- Equivalence checking is run if registers may have moved.
- The data and its provenance are archived with the release.

10. Conclusion

A technology migration changes the numbers in a design, but not all of them, and not by a common factor. Cycle-based protocol rules are constant. Time-based limits are constant in time and change in cycles only with the clock. Implementation latencies change only when the pipeline depth crosses an integer step, and the depth depends on cell delay, net delay, and an overhead whose margin components often do not shrink. A design that keeps its depth needs no RTL or assertion change. A design that reshapes its depth needs both to change together, from one block.

The complete example makes this concrete with integers throughout. At the same clock, the assumed 3 nm data removes one stage from one path and leaves the other unchanged. At a faster clock, the same data needs more stages than the 5 nm design had, and it changes the memory, timer, reset, and synchronizer numbers as well. In every case the edit is to the parameter block, not to the files.

An assistant can carry out that change reliably when it is given the basis of every number and the timing data behind it, and it should refuse to guess when it is not. It cannot generate the timing data, and it cannot decide whether a latency change is acceptable to the neighbours. Those remain the responsibility of the people who own the specification and run the timing tools.

Appendix A. Technology data sheet (completed with assumed numbers)

Every number in this appendix is an assumption chosen for the worked example. None is process data. Replace each with a value from your own libraries and timing reports.

A.1 Technology and operating point

Item	Old (5 nm-class)	New (3 nm-class)
Foundry, library, flavour	Foundry A, high-performance library, SVT and LVT cells	Foundry B, high-performance library, LVT cells
Nominal supply	750 mV	700 mV
Setup signoff corner	Slow process, 675 mV, 125 C	Slow process, 630 mV, 125 C
Hold signoff corner	Fast process, 825 mV, −40 C	Fast process, 770 mV, −40 C
Clock period, target 1	2000 ps	2000 ps
Clock period, target 2	not used	1400 ps
Separate build per clock	not applicable	Yes

A.2 Register overhead

Component	Old (ps)	New (ps)
Clock-to-q	150	120
Setup	50	40
Clock skew	50	50
Clock uncertainty and jitter	80	90
On-chip variation derate	70	80
Additional margin	0	0
Total overhead t_ovh	400	380
Hold time (not used for cycle counts)	20	15

A.3 Path delays (post-synthesis, setup corner, assumed)

Path	Cell old	Cell new	Net old	Net new	Total old	Total new
req to rdy	2400	1920	1000	950	3400	2870
rdy to ack	5600	4480	2400	2280	8000	6760

Cell-delay percentage of the old value: 80. Net-delay percentage of the old value: 95. All values in ps. Source: post-synthesis timing report at the setup corner (assumed); replace with post-route values when available.

A.4 Non-datapath latencies

Item	Old	New
Memory macro access time	1500 ps	1400 ps
Capture margin (setup + skew + uncertainty + variation)	250 ps	260 ps
Memory read latency at 2000 ps	1 cycle	1 cycle
Memory read latency at 1400 ps	not used	2 cycles
Synchronizer depth at 2000 ps	2	2
Synchronizer depth at 1400 ps	not used	3 (assumed from an MTBF analysis)
Minimum 100 ns wait	50 cycles	50 cycles at 2000 ps; 72 cycles at 1400 ps
Reset hold 200 ns	100 cycles	100 cycles at 2000 ps; 143 cycles at 1400 ps
Clock-gate enable setup	90 ps	80 ps
Multicycle exceptions	none	none

A.5 Specification limits and decision

Item	Value
req to rdy limit	8000 ps
rdy to ack limit	12000 ps
MAX (rdy, ack) at 2000 ps	4, 6
MAX (rdy, ack) at 1400 ps	5, 8
Spacing: req low for MAX_RDY + MAX_ACK	10 cycles at 2000 ps; 13 cycles at 1400 ps
Latency	Fixed
May add stages	Yes; up to MAX on each path
Maximum total latency	20000 ps
Decision at 2000 ps	Keep depth (3, 5) because neighbours depend on the latency; reshape to (2, 5) only if the latency reduction is approved
Decision at 1400 ps	Reshape to (3, 7); keeping (3, 5) fails setup by 332 ps on the ack path

Appendix B. Package with integer technology data

hs_lat_pkg_tech.sv

```
// hs_lat_pkg_tech.sv : latency package with an integer technology data block
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//
// ALL TECHNOLOGY NUMBERS BELOW ARE ASSUMPTIONS chosen for a worked example. They are not
// process data. Replace them with post-synthesis / post-route timing-report values.
// Every quantity is an integer: times in picoseconds, ratios in whole percent.
//
// Select the target at compile time:
//   +define+HS_CLK_PS=2000       clock period in ps (default 2000)
//   +define+HS_TECH_3NM          3 nm-class data   (default: 5 nm-class data)
package hs_lat_pkg;

`ifndef HS_CLK_PS
  `define HS_CLK_PS 2000
`endif

  localparam int CLK_PS        = `HS_CLK_PS;
  localparam int T_RDY_SPEC_PS = 8000;    // req -> rdy limit, TIME-based, node independent
  localparam int T_ACK_SPEC_PS = 12000;   // rdy -> ack limit, TIME-based, node independent

  // ---- 5 nm-class base data (assumed) ------------------------------------------------
  localparam int B_CLK2Q_PS   = 150;
  localparam int B_SETUP_PS   = 50;
  localparam int B_SKEW_PS    = 50;
  localparam int B_UNCERT_PS  = 80;
  localparam int B_OCV_PS     = 70;
  localparam int B_RDY_CELL   = 2400;     // req -> rdy: sum of cell delays
  localparam int B_RDY_NET    = 1000;     // req -> rdy: sum of net delays
  localparam int B_ACK_CELL   = 5600;     // rdy -> ack: sum of cell delays
  localparam int B_ACK_NET    = 2400;     // rdy -> ack: sum of net delays

`ifdef HS_TECH_3NM
  // ---- 3 nm-class data (assumed): register terms given directly, path delays by percent ----
  localparam int CLK2Q_PS   = 120;
  localparam int SETUP_PS   = 40;
  localparam int SKEW_PS    = 50;
  localparam int UNCERT_PS  = 90;         // margins do not shrink; they grow
  localparam int OCV_PS     = 80;
  localparam int CELL_PCT   = 80;         // 3 nm cell delay = 80 % of the 5 nm value
  localparam int NET_PCT    = 95;         // 3 nm net delay  = 95 % of the 5 nm value
`else
  localparam int CLK2Q_PS   = B_CLK2Q_PS;
  localparam int SETUP_PS   = B_SETUP_PS;
  localparam int SKEW_PS    = B_SKEW_PS;
  localparam int UNCERT_PS  = B_UNCERT_PS;
  localparam int OCV_PS     = B_OCV_PS;
  localparam int CELL_PCT   = 100;
  localparam int NET_PCT    = 100;
`endif

  localparam int REG_OVH_PS = CLK2Q_PS + SETUP_PS + SKEW_PS + UNCERT_PS + OCV_PS;  // 400 or 380

  localparam int D_RDY_PS = (B_RDY_CELL * CELL_PCT) / 100 + (B_RDY_NET * NET_PCT) / 100;  // 3400 or
2870
  localparam int D_ACK_PS = (B_ACK_CELL * CELL_PCT) / 100 + (B_ACK_NET * NET_PCT) / 100;  // 8000 or
6760
```

```
  localparam int USABLE_PS = CLK_PS - REG_OVH_PS;

  function automatic int ceil_div(input int a, input int b);
    return (a + b - 1) / b;
  endfunction

  localparam int MAX_RDY = T_RDY_SPEC_PS / CLK_PS;   // floor: a limit is never exceeded
  localparam int MAX_ACK = T_ACK_SPEC_PS / CLK_PS;   // floor
  localparam int MAX_TOT = MAX_RDY + MAX_ACK;

  // Verification-only overrides (mismatch testing); never define in production.
`ifdef HS_FORCE_N_RDY
  localparam int N_RDY = `HS_FORCE_N_RDY;
`else
  localparam int N_RDY = ceil_div(D_RDY_PS, USABLE_PS);   // ceil: a stage never exceeds the clock
`endif
`ifdef HS_FORCE_N_ACK
  localparam int N_ACK = `HS_FORCE_N_ACK;
`else
  localparam int N_ACK = ceil_div(D_ACK_PS, USABLE_PS);   // ceil
`endif

endpackage : hs_lat_pkg
```

Appendix C. RTL

hs_pipe.sv

```
// hs_pipe.sv : req -> rdy -> ack pulse pipeline with clock-derived depth
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//
// req  : input, 1-cycle pulse, single outstanding transaction
// rdy  : DUT-internal 1-cycle pulse, exactly N_RDY cycles after req
// ack  : output 1-cycle pulse, exactly N_ACK cycles after rdy
//
// Pipeline depth comes from hs_lat_pkg (N_RDY, N_ACK). Where real logic exists, the
// D_RDY_PS / D_ACK_PS of combinational logic is split across the stages, i.e. the
// logic of stage k sits between rdy_pipe[k-1] and rdy_pipe[k] (same for ack_pipe).
module hs_pipe (
  input  logic clk,
  input  logic reset_n,   // active-low, asynchronous assert, synchronous release by design
  input  logic req,
  output logic ack
);
  import hs_lat_pkg::*;

  logic [N_RDY-1:0] rdy_pipe;
  logic [N_ACK-1:0] ack_pipe;
  logic             rdy;                // internal obligation signal (observed by bind)

  // REQ-8 feasibility: refuse a clock that cannot meet the time-based spec.
`ifndef HS_NO_FEAS_CHECK
  initial begin
    if (N_RDY < 1 || N_RDY > MAX_RDY)
      $fatal(1, "hs_pipe: INFEASIBLE req->rdy: N_RDY=%0d, MAX_RDY=%0d at CLK_PS=%0d",
             N_RDY, MAX_RDY, CLK_PS);
    if (N_ACK < 1 || N_ACK > MAX_ACK)
      $fatal(1, "hs_pipe: INFEASIBLE rdy->ack: N_ACK=%0d, MAX_ACK=%0d at CLK_PS=%0d",
             N_ACK, MAX_ACK, CLK_PS);
  end
`endif

  always_ff @(posedge clk or negedge reset_n) begin
```

```
    if (!reset_n) begin
      rdy_pipe <= '0;                  // REQ-5: flush in-flight work on reset
      ack_pipe <= '0;
    end else begin
      rdy_pipe[0] <= req;
      for (int i = 1; i < N_RDY; i++) rdy_pipe[i] <= rdy_pipe[i-1];
      ack_pipe[0] <= rdy;
      for (int j = 1; j < N_ACK; j++) ack_pipe[j] <= ack_pipe[j-1];
    end
  end

  assign rdy = rdy_pipe[N_RDY-1];
  assign ack = ack_pipe[N_ACK-1];

endmodule : hs_pipe
```

Appendix D. Assertions

hs_sva.sv

```
// +----------------------+------+------+------+------+------+
// | mp_req_pulse_spacing | UR-1 | UR-2 | UR-3 | UR-4 | UR-5 |
// |                      | PASS | PASS | PASS | N/A  | PASS |
// +----------------------+------+------+------+------+------+
//
// INTENTION: req = assume (antecedent side); rdy, ack = obligation (consequent side).
// Not covered (not requested): spurious rdy/ack without a req; fairness (Step 8 gate pending).
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.

module hs_sva (
  input logic clk,
  input logic reset_n,
  input logic req,
  input logic rdy,
  input logic ack
);
  import hs_lat_pkg::*;
  // ---------------------------------------------------------------------
  // ASSUMPTION
  // ---------------------------------------------------------------------
  /*
  ASSUMPTION: req is a 1-cycle pulse, and after each req pulse req stays low for
              MAX_RDY + MAX_ACK cycles (single outstanding transaction; the previous
              ack always lands before the next req can rise).
  TYPE: Temporal
  JUSTIFICATION: Requirement REQ-4 (1-cycle pulse) and REQ-6 (single outstanding). The
                 spacing is a constant, not a function of rdy/ack, because an obligation
                 signal may not appear in an assume consequent (CHECK 2, user-confirmed fix).
  RISK LEVEL: Medium (environment is over-constrained: next req waits the maximum spec
              latency even when ack arrives sooner).
  */
  property p_req_pulse_spacing;
    $rose(req) |-> ##1 !req[*MAX_TOT];
  endproperty : p_req_pulse_spacing

  mp_req_pulse_spacing: assume property (@(posedge clk) disable iff (!reset_n)
    p_req_pulse_spacing
  );
   // VALIDATION COVER (UR-5)
  cp_mp_req_pulse_spacing_hit: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(req) ##1 !req[*MAX_TOT]
  );
```

```
  // ---------------------------------------------------------------------
  // ASSERTIONS (obligations)
  // ---------------------------------------------------------------------

  // CLASS: LIVENESS-B (bounded, FORM A)  -- spec-level window, time-derived MAX_RDY
  property p_req_rdy_window;
    $rose(req) |-> strong(##[1:MAX_RDY] rdy);
  endproperty : p_req_rdy_window

  ap_req_rdy_window: assert property (@(posedge clk) disable iff (!reset_n)
    p_req_rdy_window
  ) else $error("FV-SIM: rdy missing after %0d cycles from req at %0t", MAX_RDY, $realtime);

  // CLASS: LIVENESS-B (bounded, FORM A)  -- spec-level window, time-derived MAX_ACK
  // EXCEPTION 7b: $rose(rdy) = pure timing anchor (user confirmed)
  // rdy is obligation/output -- used only as edge-trigger reference, not as constraint
  property p_rdy_ack_window;
    $rose(rdy) |-> strong(##[1:MAX_ACK] ack);
  endproperty : p_rdy_ack_window

  ap_rdy_ack_window: assert property (@(posedge clk) disable iff (!reset_n)
    p_rdy_ack_window
  ) else $error("FV-SIM: ack missing after %0d cycles from rdy at %0t", MAX_ACK, $realtime);

  // CLASS: LIVENESS-B (bounded, exact)  -- implementation-level: fixed pipeline depth N_RDY
  // rdy must be high at exactly N_RDY cycles after req and be a 1-cycle pulse.
  property p_req_rdy_exact;
    $rose(req) |-> strong(##N_RDY (rdy ##1 !rdy));
  endproperty : p_req_rdy_exact

  ap_req_rdy_exact: assert property (@(posedge clk) disable iff (!reset_n)
    p_req_rdy_exact
  ) else $error("FV-SIM: rdy not a 1-cycle pulse at exactly %0d cycles after req at %0t", N_RDY,
$realtime);

  // CLASS: LIVENESS-B (bounded, exact)  -- implementation-level: fixed pipeline depth N_ACK
  // EXCEPTION 7b: $rose(rdy) = pure timing anchor (user confirmed)
  // rdy is obligation/output -- used only as edge-trigger reference, not as constraint
  property p_rdy_ack_exact;
    $rose(rdy) |-> strong(##N_ACK (ack ##1 !ack));
  endproperty : p_rdy_ack_exact

  ap_rdy_ack_exact: assert property (@(posedge clk) disable iff (!reset_n)
    p_rdy_ack_exact
  ) else $error("FV-SIM: ack not a 1-cycle pulse at exactly %0d cycles after rdy at %0t", N_ACK,
$realtime);
```

```
  // ---------------------------------------------------------------
  // COVERAGE (Item 9 antecedent reachability, Item 10 full-sequence execution)
  // ---------------------------------------------------------------
  cp_req_rdy_window_trigger: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(req)
  );
  cp_req_rdy_window_full: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(req) ##[1:MAX_RDY] rdy
  );

  cp_rdy_ack_window_trigger: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(rdy)
  );
  cp_rdy_ack_window_full: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(rdy) ##[1:MAX_ACK] ack
  );

  cp_req_rdy_exact_trigger: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(req)
  );
  cp_req_rdy_exact_full: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(req) ##N_RDY rdy ##1 !rdy
  );

  cp_rdy_ack_exact_trigger: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(rdy)
  );
  cp_rdy_ack_exact_full: cover property (@(posedge clk) disable iff (!reset_n)
    $rose(rdy) ##N_ACK ack ##1 !ack
  );

endmodule : hs_sva

// bind-observed: rdy is an internal signal of hs_pipe, visible to the bound instance.
bind hs_pipe hs_sva u_hs_sva (
  .clk     (clk),
  .reset_n (reset_n),
  .req     (req),
  .rdy     (rdy),
  .ack     (ack)
);
```

Appendix E. Formula summary

Quantity	Formula	Rounding
Migrated path delay	D_new = (g × D_cell) / 100 + (w × D_net) / 100	none (integers)
Overhead	t_ovh = clk-to-q + setup + skew + uncertainty + variation + margin	none
Usable time	T_use = T_clk − t_ovh	none
Stages	N = ceil(D / T_use)	up
Legal cycles	MAX = floor(T_spec / T_clk)	down
Feasible	1 ≤ N ≤ MAX	
Stage logic	L = ceil(D / N)	up
Stage slack	T_clk − (t_ovh + L)	
Single-outstanding spacing	MAX_TOT = MAX_RDY + MAX_ACK	

Quantity	Formula	Rounding
Time-based counter (minimum wait)	cycles = ceil(t_required / T_clk)	up
Time-based counter (maximum wait)	cycles = floor(t_limit / T_clk)	down
Memory read latency	ceil((access time + capture margin) / T_clk)	up

References

1. B. Cohen and K. Chibani, *Knowledge-Driven AI for SystemVerilog Assertions*, VhdlCohen Publishing, 2026.
2. B. Cohen, *SystemVerilog Assertions Handbook*, 4th ed., VhdlCohen Publishing, 2023.
3. IEEE Std 1800, SystemVerilog: Unified Hardware Design, Specification, and Verification Language.
4. Foundry design-kit and library documentation, and the static timing analysis tool documentation, for the definitions of corners, derating, and uncertainty used at signoff.

Further material is available at systemverilog.us.