

Using RAG-RTL with RAG-SVA: A Pipelined Multiplier Example

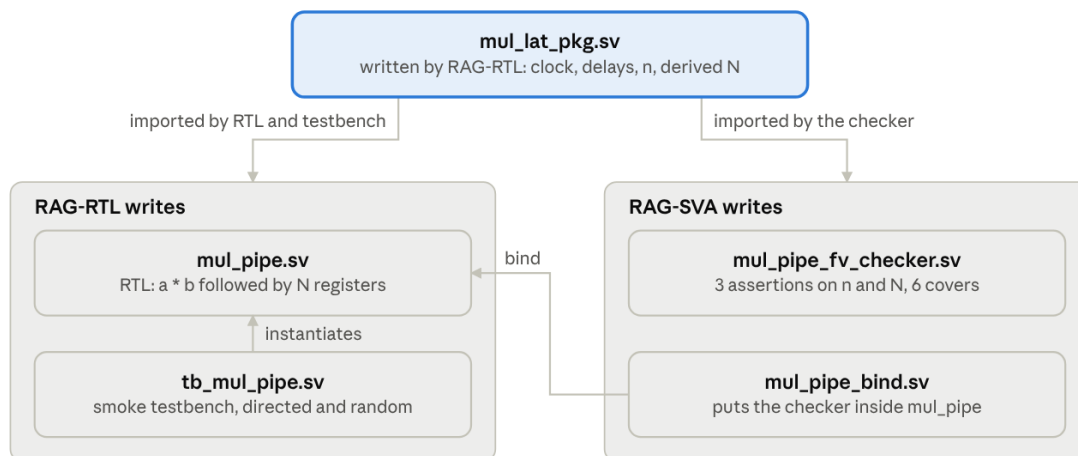
Oct 11, 2026 · Ben_Cohen@SystemVerilog.us

Ben Cohen, Dr. Kais Chibani · systemverilog.us

RAG-RTL and RAG-SVA generate a design and its assertions from one requirement and one shared latency package, so the RTL and the checker cannot disagree on timing. For a multiplier required to run at 2 GHz, the package derives 9 pipeline stages from the assumed delays, so the requirement is feasible only when n is 9 cycles or more. This paper shows the files to load, the settings to state, how the technology enters, the files that come out, and the condition under which the workflow declares a requirement not feasible and stops.

How the files fit together

One package sets the timing for the RTL, the checker and the testbench



file dependencies · 1 shared package, 2 files from each framework

RAG-RTL writes the latency package first. The RTL, the testbench and the checker import it, so the pipeline depth and the assertion bounds come from the same numbers: the clock period, the two delay estimates and n . The bind file attaches the checker to the RTL without editing the RTL.

RAG-RTL

One latency package. The RTL and its assertions cannot disagree on timing.

STEP 1

You state

Clock frequency
Logic delay
Cycle limit, n



STEP 2

It derives

Stages needed, N
Feasible if $N \leq n$
If not, it stops



STEP 3

It generates

RTL with N stages
SVA on n and N
Bind and testbench

$$N = \lceil \text{delay} \div (\text{clock period} - \text{register overhead}) \rceil$$

Computed once, in the package. The RTL depth and the assertion bounds both read it.

Example: 8×8 multiplier, assuming a 2,600 ps delay and 200 ps of register overhead

1 GHz



4 stages

Product out in 4 ns

2 GHz



9 stages

Product out in 4.5 ns

2 GHz, $n = 4$



Needs 9, allowed 4

Not feasible: the build stops

✓ Asks before it guesses

✓ Finishes every module

✓ Elaborates before delivery

Ben Cohen · Kais Chibani
systemverilog.us

Version 0.1 draft, not yet validated on a benchmark. Delays in the example are assumed, not measured.

Framework files to load

Five RAG files must be read in full before the first requirement; the rest are consulted as the workflow reaches them. On a conflict, the file higher in each list wins.

RAG-SVA (assertions)

Precedence	File	Role	Load
1	Last_Minute_requests.md	Overrides and reminders; decomposition mode; strong() on bounded consequents	In full, first
2	DoThisFirst.md	Workflow: CHECK 0, Phases 1 to 7, gates, halt rules	In full, first
3	rag_make.md	Knowledge graph of the rules and the numbered checks	In full, first
4	step0.md	System initialization: source of truth, anti-hallucination rules, strict format such as no =>, explicit delays, label prefixes	In full, first
5	Intention_Boundary_Property_Type_gold.md	Which signals are assumed and which are obligations	At Phase 2
6	Pre_instructions_for_FV.md	Formal rules, used because FV MODE is YES	At setup
6	settings.md, settingsCombined.txt	Numbered settings items and their defaults	At CHECK 0
6	RAG_RULES.md, PropertyNamingQuickRef.md, Lessons_learned.md	Rule corpus, naming, known failure patterns	On demand

RAG-RTL (design)

Precedence	File	Role	Load
1	RTL_DoThisFirst.md	Workflow: CHECK 0, Phases 1 to 7, halt rules	In full, first
2	RTL_RULES.md	Rule set: spec, ports, coding, completeness, pipeline and latency, verification pairing	At Phases 1 to 6
3	RTL_settings.md	The 19 settings resolved at CHECK 0	At CHECK 0
4	RTL_Lessons_learned.md	Bad and good pairs for known failures	On demand
5	scripts/rtl_postcheck.py	Phase 5 text scan; a FAIL stops delivery	At Phase 5

Three items are produced once and used by both frameworks: the ambiguity gate (RAG-SVA RQM0), the port table, and the latency package.

Settings

RAG-SVA takes four lines, stated in the first message; RAG-RTL resolves its settings at CHECK 0, and only four of them differ from the defaults in this example.

RAG-SVA, as typed

```
FV MODE:    YES          (formal rules on; determinism enforced)
DECOMP:     STAGE=3      (pipeline; up to 3 stages chained in the consequent)
GATE:       ABBREVIATED (pause at CHECK 0, Phase 1, Phase 3, Phase 7)
VERBOSITY:  !C          (code only)
```

Every RAG-SVA reply then opens with the mode line `MODE: STAGE=3 (PIPELINE form, up to 3 stages)` • Item 39: SEPARATED • FV: YES.

RAG-RTL, as resolved at CHECK 0

```
CHECK 0: TARGET=ASIC  HDL=SV2012  RESET=ASYNC/ACTIVE_LOW
        CLK_PS=500 (2 GHz)  TECH=UNKNOWN
        REG_OVH_PS=200  DELAY_PROVENANCE=ESTIMATE  RETIMING=YES
        LATENCY_POLICY=KEEP  CONTEXT_SPLIT=6
        LINT_TOOL=slang + Verilator  GATE_MODE=ABBREVIATED
```

Setting	Value used	Default	Why it was set
CLK_PS	500 ps	UNKNOWN	The requirement names 2 GHz; the period is $10^{12} / f$ in hertz, shown at CHECK 0 for confirmation
REG_OVH_PS	200 ps, an estimate	UNKNOWN	Needed to size the pipeline; no technology data
RETIMING	YES	NO	The RTL is a * b plus stacked registers that synthesis moves
LINT_TOOL	slang and Verilator	iverilog	The tools available in the session

One value is not a setting but must be supplied because TECH is UNKNOWN: the multiplier delay, **2,600 ps**, also an estimate. Both estimates live in the latency package and nowhere else. The next section says what they stand in for.

How the technology is addressed

Only indirectly, and that is a real gap in this example: the technology enters as two numbers in the latency package, and both are assumptions. Users must replace them with actual values for the technology they are using or plan to use.

What stands in for the technology today

Item	Value in this example	What it should be
TECH setting	UNKNOWN	The library or FPGA family name
D_MUL_PS, multiplier delay	2,600 ps , assumed	From a synthesis or timing report of an 8×8 multiplier in that library
REG_OVH_PS, per-stage overhead	200 ps , assumed	Clock-to-q, setup, skew , uncertainty and variation margin for that library
DELAY_PROVENANCE	ESTIMATE	POST-SYNTH, then POST-ROUTE
TARGET	ASIC (the default)	ASIC or FPGA; this changes the coding rules and whether a hard multiplier block is used

Neither framework has a technology library, and neither runs synthesis or static timing analysis. Every stage count in this paper, and every "feasible" or "not feasible" verdict, is a statement about 2,600 ps and 200 ps, not about any real process.

How the workflow brings the technology in

1. **CHECK 0.** The user names TECH and supplies the delays, or accepts estimates. With TECH=UNKNOWN and a multiplier in the design, the workflow must ask before it can size the pipeline.
2. **Package.** The numbers go in one place, labelled with their provenance.
3. **Phase 7, timing hand-off.** The report lists each assumed delay and states that depth is not signed off until timing analysis confirms it.
4. **After synthesis.** The user returns the reported delay or the slack. Only the package is changed; the stage count, the RTL depth and the assertion bounds are regenerated from it.

What a single delay number does not capture

- **Corners.** The delay must be the worst case for process, voltage and temperature, not the typical one.
- **Uneven stages.** Dividing the delay by N assumes retiming balances the multiplier perfectly.
- **Wires.** Post-route delay is longer than post-synthesis delay, so depth can change late.
- **Width.** The delay is for 8×8 . Changing W needs a new number.

The request and the gate answers

One sentence of requirement was enough to start, because the gates asked for everything it left out.

Requirement, as given:

1. Write a model of a multiplier.
2. If rq , then within n cycles, c equals a times b .
3. Write that property. Show the timing, assuming a 2 GHz clock.

The requirement gives the clock as a frequency, but no value for n , no widths, no reset, no technology, and no way to tell a result from an idle output. Both frameworks halt on such gaps and write no code until they are closed.

Gate	Question	Answer recorded
CHECK 0	Clock	2 GHz , converted to CLK_PS=500
CHECK 0	Delay estimates	Multiplier 2,600 ps , register overhead 200 ps
CHECK 0	How the multiplier is cut	Stacked registers for retiming (RETIMING=YES)
Phase 1, R1	The limit n	9 cycles, the fewest that fit at 2 GHz with these estimates
Phase 1, R2	Arithmetic	Unsigned, 8 × 8 into 16 bits, so no overflow case
Phase 1, R3	How the result is marked	An added v1d output, high for the result cycle
Phase 1, R4	Throughput	A new request allowed every cycle, no back-pressure
Phase 1, R5	c without a result	Don't-care; meaningful only with v1d
Phase 3	Architecture plan	Accepted: 9 stages, product out 4.5 ns after rq
Step 7	What to generate	Option A: assertions and coverage

The value of **n** is the one answer the clock forces. Any **n** below 9 is not feasible at 2 GHz with these estimates, and the workflow says so before it writes code; the section on infeasibility gives the condition.

Taken as given at Phase 1: **rq** in cycle T0 allows the result from T0+1 through T0+n, never in T0; **rstn** is active-low with asynchronous assert and synchronous release; **v1d** and **c** reset to 0.

The session that produced the files answered these gates for a **1 MHz** clock and **n = 4**. The table restates them for the 2 GHz requirement; only the clock and n differ.

Generated files

Six files come out, and the first one is the contract: every cycle count in the RTL, the checker and the testbench is read from the package.

#	File	Produced by	Role	Lines
1	mul_lat_pkg.sv	RAG-RTL, shared	Clock period, delay estimates, n , derived stage count, widths	47
2	mul_pipe.sv	RAG-RTL	The multiplier	85
3	mul_pipe_fv_checker.sv	RAG-SVA	3 assertions and 6 covers	164
4	mul_pipe_bind.sv	RAG-SVA	Attaches the checker to the RTL without editing it	17
5	tb_mul_pipe.sv	RAG-RTL	Smoke testbench: directed and random products, self-checking	243
6	check_mul_trace.py	Support script	Re-checks the assertions on the simulation trace	121

Compile files 1 to 5 in that order. The clock is the only build option:

+define+MUL_CLK_PS=<period in ps>.

The listings below are the delivered code with the header comments shortened. For the **2 GHz** requirement two constants in the package differ from the delivered file, which holds a 1 MHz default and **n = 4**: **MUL_CLK_PS** is **500** and **MAX_MUL** is **9**. No other file changes.

1. mul_lat_pkg.sv: the shared latency package, set for 2 GHz

```
// mul_lat_pkg.sv : shared latency package for mul_pipe (used by RTL, SVA and TB)
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//  USABLE_PS = CLK_PS - REG_OVH_PS          usable logic time per pipeline stage
//  N_MUL      = ceil(D_MUL_PS / USABLE_PS)    stages the multiplier needs at this clock
//  Feasible   : 1 <= N_MUL <= MAX_MUL         otherwise mul_pipe refuses to elaborate
`timescale 1ps/1ps
`default_nettype none

package mul_lat_pkg;

`ifndef MUL_CLK_PS
  `define MUL_CLK_PS 500
`endif

  localparam int W          = 8;              // operand width (unsigned)
  localparam int PW         = 2 * W;          // product width: the full product always fits

  localparam int CLK_PS     = `MUL_CLK_PS;    // clock period
  localparam int REG_OVH_PS = 200;            // ESTIMATE: clk-to-q + setup + margin per stage
  localparam int D_MUL_PS   = 2600;          // ESTIMATE: combinational delay of the W x W multiplier
  localparam int MAX_MUL    = 9;             // spec: product within this many cycles of rq
```

```

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

// A clock no longer than the register overhead leaves no time for logic: report it as
// N_MUL = 0 so that the feasibility check in mul_pipe rejects the build.
localparam int N_MUL = (USABLE_PS > 0) ? ceil_div(D_MUL_PS, USABLE_PS) : 0;

endpackage : mul_lat_pkg

```

2. mul_pipe.sv: the RTL

```

// mul_pipe.sv : pipelined unsigned multiplier, depth derived from the clock
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
`timescale 1ps/1ps
`default_nettype none

module mul_pipe (
    input logic clk,
    input logic rstn,
    input logic rq,
    input logic [mul_lat_pkg:W-1:0] a,
    input logic [mul_lat_pkg:W-1:0] b,
    output logic vld,
    output logic [mul_lat_pkg:PW-1:0] c
);
    import mul_lat_pkg::*;

    // Depth used for declarations; stays legal when N_MUL is out of range so that the
    // feasibility check below is the message the user sees.
    localparam int NP = (N_MUL < 1) ? 1 : N_MUL;

    logic [PW-1:0] prod_pipe [NP]; // retiming stack for the product
    logic [NP-1:0] vld_pipe; // valid travels with the data

    // Feasibility: elaboration-time check in generate scope
    if (USABLE_PS <= 0) begin : g_infeasible_clk
        $fatal(1, "mul_pipe: INFEASIBLE: CLK_PS=%0d leaves no logic time after REG_OVH_PS=%0d",
            CLK_PS, REG_OVH_PS);
    end else if (N_MUL < 1 || N_MUL > MAX_MUL) begin : g_infeasible_mul
        $fatal(1, "mul_pipe: INFEASIBLE: multiplier needs N_MUL=%0d stage(s) at CLK_PS=%0d, spec
allows 1..%0d",
            N_MUL, CLK_PS, MAX_MUL);
    end

    always_ff @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            vld_pipe <= '0;
            for (int i = 0; i < NP; i++) prod_pipe[i] <= '0;
        end else begin
            vld_pipe[0] <= rq;
            prod_pipe[0] <= PW'(a) * PW'(b); // operands widened first: full product
            for (int i = 1; i < NP; i++) begin
                vld_pipe[i] <= vld_pipe[i-1];
                prod_pipe[i] <= prod_pipe[i-1];
            end
        end
    end
end

```

```

    assign vld = vld_pipe[NP-1];
    assign c    = prod_pipe[NP-1];

endmodule : mul_pipe

```

3. mul_pipe_fv_checker.sv: the assertions

The first property is the requirement as worded, with $n = \text{MAX_MUL}$. The second is the tight check at the derived depth N_MUL . The third rejects a vld that no request caused. The six covers, a trigger and a full-sequence cover per assertion, are in the file and not shown.

```

// File: mul_pipe_fv_checker.sv -- FV checker for mul_pipe
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// MODE: STAGE=3 (PIPELINE form) | Item 39: SEPARATED | FV MODE: YES | GATE:
// ABBREVIATED
`timescale 1ps/1ps

module mul_pipe_fv_checker (
    input logic          clk,
    input logic          rstn, // active-low reset
    input logic          rq,   // INTENTION = assume
    input logic [mul_lat_pkg::W-1:0] a, // INTENTION = assume
    input logic [mul_lat_pkg::W-1:0] b, // INTENTION = assume
    input logic          vld,   // INTENTION = obligation
    input logic [mul_lat_pkg::PW-1:0] c // INTENTION = obligation
);
    import mul_lat_pkg::*;

    // CLASS: SAFETY  AXIS2: STRONG  AXIS3: BOUNDED
    // Spec: if rq, then in at most MAX_MUL cycles vld is high and c equals a times b.
    property p_mul_result_in_window;
        logic [W-1:0] v_a, v_b;
        (rq, v_a = a[W-1:0], v_b = b[W-1:0]) |->
            strong(##[1:MAX_MUL] (vld && (c[PW-1:0] == PW'(v_a) * PW'(v_b))));
    endproperty : p_mul_result_in_window

    ap_mul_result_in_window: assert property (@(posedge clk) disable iff (!rstn)
        p_mul_result_in_window)
        else $error("FV-SIM: ap_mul_result_in_window FAIL: no vld with c = a*b in 1..%0d
cycles after rq, time=%0t",
            MAX_MUL, $realtime);

    // CLASS: SAFETY  AXIS2: STRONG  AXIS3: BOUNDED
    // Implementation: the product is out at exactly N_MUL cycles after rq.
    property p_mul_result_exact;
        logic [W-1:0] v_a, v_b;
        (rq, v_a = a[W-1:0], v_b = b[W-1:0]) |->
            strong(##N_MUL (vld && (c[PW-1:0] == PW'(v_a) * PW'(v_b))));
    endproperty : p_mul_result_exact

    ap_mul_result_exact: assert property (@(posedge clk) disable iff (!rstn)
        p_mul_result_exact)
        else $error("FV-SIM: ap_mul_result_exact FAIL: vld=%0b c=%0d at exactly %0d
cycle(s) after rq, time=%0t",
            $sampled(vld), $sampled(c[PW-1:0]), N_MUL, $realtime);

```

```
// CLASS: SAFETY  AXIS2: STRONG  AXIS3: BOUNDED
// A cycle without a request produces no valid N_MUL cycles later.
property p_mul_no_spurious_vld;
  !rq |-> strong(##N_MUL !vld);
endproperty : p_mul_no_spurious_vld

ap_mul_no_spurious_vld: assert property (@(posedge clk) disable iff (!rstn)
  p_mul_no_spurious_vld)
  else $error("FV-SIM: ap_mul_no_spurious_vld FAIL: vld high %0d cycle(s) after a
cycle without rq, time=%0t",
  N_MUL, $realtime);

// ... six cover properties (cp_) follow in the delivered file ...

endmodule : mul_pipe_fv_checker
There are no assumptions: rq, a and b are unconstrained inputs.
```

4. mul_pipe_bind.sv: the bind

```
// File: mul_pipe_bind.sv -- binds the FV checker to mul_pipe
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// Compile after mul_lat_pkg.sv, mul_pipe.sv and mul_pipe_fv_checker.sv.

bind mul_pipe mul_pipe_fv_checker u_mul_pipe_fv_checker (
  .clk  (clk),
  .rstn (rstn),
  .rq   (rq),
  .a    (a),
  .b    (b),
  .vld  (vld),
  .c    (c)
);
```

Timing at 2 GHz

At 2 GHz the multiplier needs 9 pipeline stages, so the product is out 9 cycles, 4.5 ns, after rq. The stage count N is the multiplier delay divided by the usable time per stage, rounded up.

$$N = \left\lceil \frac{D_{\text{mul}}}{T_{\text{clk}} - T_{\text{ovh}}} \right\rceil = \left\lceil \frac{2600}{500 - 200} \right\rceil = 9$$

The simulated trace for 7×6 . X marks a value nobody may rely on.

Cycle :	T-1	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	
time :		0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	ns after T0
clk :	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	
rq :	0	1	0	0	0	0	0	0	0	0	0	0	
		↑ rose(rq)											
a :	X	7	X	X	X	X	X	X	X	X	X	X	
b :	X	6	X	X	X	X	X	X	X	X	X	X	
vld :	0	0	0	0	0	0	0	0	0	0	1	0	
			T0+9 = N = n: product out 4.5 ns after rq ↑										
c :	X	X	X	X	X	X	X	X	X	X	42	X	

The same files at other clocks, for comparison. Only the clock define changes; the stage count follows from the package.

Clock frequency	Period (ps)	Usable per stage (ps)	Stages, N	Product out after rq
1 MHz	1,000,000	999,800	1	1,000 ns
1 GHz	1,000	800	4	4 ns
2 GHz, as required	500	300	9	4.5 ns

A faster clock does not mean a faster answer. From **1 GHz** to **2 GHz** the stage count more than doubles, because the 200 ps overhead takes a larger share of each shorter cycle, and the product arrives **0.5 ns** later.

When the requirement is not feasible

A requirement is not feasible when the clock frequency it asks for is higher than the multiplier delay and the cycle limit allow. The workflow then stops at the architecture plan, before any RTL or assertion is written.

The feasibility condition

$$N = \left\lceil \frac{D_{\text{mul}}}{1/f_{\text{clk}} - T_{\text{ovh}}} \right\rceil \leq n$$

$$\Leftrightarrow f_{\text{clk}} \leq f_{\text{max}}(n) = \frac{1}{D_{\text{mul}}/n + T_{\text{ovh}}}$$

Here **f** is the clock frequency, **D** the multiplier delay (**2,600 ps**, an estimate), **T** the register overhead per stage (**200 ps**, an estimate), and **n** the number of cycles the requirement allows. The requirement is not feasible when the clock frequency is above **f-max** for its **n**. The clock period must also be longer than the register overhead, or no logic fits at all.

The 2 GHz requirement, by cycle limit

Cycle limit, n	Highest clock for that n	Stages needed at 2 GHz	Verdict at 2 GHz
2	667 MHz	9	Not feasible; the build stops
4	1.18 GHz	9	Not feasible; the build stops
8	1.90 GHz	9	Not feasible; the build stops
9	2.04 GHz	9	Feasible; built and simulated

So the requirement as given, with **n** left open, has one workable answer at 2 GHz: **n** of 9 or more. The row for **n = 2** is the case of a second requirement added to the first: **c == a * b** in 2 cycles. Two cycles at 2 GHz is 1 ns, and the multiplier alone is estimated at 2.6 ns.

What the workflow reports, and when

When	Flag	Action
CHECK 0	The clock is given as a frequency	Converted to CLK_PS=500 and shown for confirmation
CHECK 0	TECH=UNKNOWN and the design has a multiplier	Delay estimates are requested; no depth can be computed without them
Phase 1	n has no value, or "within 2 cycles" and "in 2 cycles" are both stated	Halt and ask for the limit, and whether it is an upper limit or an exact latency
Phase 3	Not feasible: N = 9 against the n given	Halt; no RTL and no SVA until the requirement changes
Elaboration	The RTL's own check, if the files are built anyway	The build stops with the message below

\$fatal encountered: mul_pipe: INFEASIBLE: multiplier needs N_MUL=9 stage(s) at CLK_PS=500, spec allows 1..2

The ways out offered at Phase 3

- **Relax the cycle limit** to 9 or more at 2 GHz.
- **Lower the clock frequency** to the highest clock for the n required: 667 MHz for n=2, **1.18 GHz** for n = 4.
- **Supply a smaller multiplier delay.** Two stages at 2 GHz need **600 ps** or less, which means a faster multiplier or a faster technology.

The workflow does not pick one, and it never widens the limit on its own.

The exact form: $c == a * b$ in 2 cycles

"In 2 cycles" fixes the latency; "within 2 cycles" only bounds it. The two agree while the derived depth is 2, which is above 357 MHz and up to 667 MHz. At 357 MHz and below the derived depth is 1: the product arrives a cycle early, which satisfies "within 2" and violates "in 2".

The delivered files would not catch that, because the exact assertion checks N_MUL, whatever it works out to. Supporting an exact latency needs three changes, none of which has been generated: a fixed latency constant in the package, an RTL that pads its pipeline to that constant, and a feasibility check that N_MUL does not exceed it.

What was checked and what was not

The RTL was compiled, elaborated and simulated at three clocks; the assertions were not run in an SVA simulator or a formal tool.

Check	1 MHz, n = 4	1 GHz, n = 4	2 GHz, n = 9
rtl_postcheck.py text scan	Pass	Pass	Pass
slang compile and elaboration, bind included	0 diagnostics	0 diagnostics	0 diagnostics
Verilator lint, -wall, on the RTL	Clean	Clean	Clean
Testbench in Verilator, 3 seeds, 220 requests each	Pass, 0 errors	Pass, 0 errors	Pass, 0 errors
Python re-check of the 3 assertions and 6 covers on the trace	Pass	Pass	Pass
Latency seen in simulation	1 cycle	4 cycles	9 cycles

The first two columns are the delivered files. The 2 GHz column is the same files with the two package constants changed on a scratch copy; the infeasible cases (n = 2, 4 and 8 at 2 GHz) were elaborated only, which is as far as they get.

Two deliberately broken versions of the RTL, run at 1 MHz and 1 GHz, show why the exact-latency assertion is kept beside the one that was asked for. They were not repeated at 2 GHz.

Broken RTL	1 MHz	1 GHz
Wrong product on some operands	Window and exact both fail	Window and exact both fail
Result one stage late	Window passes, since 2 cycles is still inside 4; exact and no-spurious fail	All three fail

Limits to state with any use of this example:

- **The assertions have not been executed.** Verilator does not support `strong()` or range delays in assertions, so the trace re-check is a Python model of the three properties, not the SVA itself.
- **The delays are assumptions.** 2,600 ps and 200 ps have no technology data behind them. Replace them in the package with timing-report values; N, the RTL depth and the assertion bounds follow.

- **The RTL relies on synthesis retiming.** Without it the whole multiplier delay sits before the first register. Nine evenly balanced stages of an 8×8 multiplier is an optimistic assumption, and depth is not signed off until static timing analysis confirms it.
- **The window assertion is loose under back-to-back requests.** In the 2 GHz run, 2 of 218 results were matched early by a neighbour with the same product.
- **Keep `rq` low during reset.** A request sampled in the last reset cycle is flushed, and the product assertions would report it.
- **Formal cost.** A multiplier is expensive to prove even at 8 bits, and a 9-cycle window adds to it; reduce `W` in the package if the proof does not converge.
- **The gates were not re-run for 2 GHz.** The gated session used 1 MHz and `n = 4`; this paper restates its answers for the 2 GHz requirement.
- **RAG-RTL is a version 0.1 draft** and has not been validated on a benchmark.