Technology and Clock Rate Belong in the Assertions

Why requirements must be translated into SVA with the technology and the clock in view: parameterized latency, delay estimates from operation types, and when AI may add a ready signal

Ben Cohen · Kais Chibani · systemverilog.us · Ben_Cohen@SystemVerilog.us · November 2026

Abstract

A set of English requirements calls for a 2 ns clock. AI generates the RTL, and one path needs about 4 ns. Can the AI fix this by adding a ready (`rdy`) signal so the slow path gets two clock periods, and how is that reflected in the SVA? This paper answers that question and draws a broader lesson from it: **the technology and the clock rate must be considered when requirements are translated into assertions.** A cycle count such as ##1 or ##2 is not a property of the function; it is the delay of the function in a given technology divided by a given clock period. Written as a literal, it is right for one technology-and-clock pair only. We therefore write latency as a parameter, computed in one package from the technology delay, the register overhead and the clock period, and checked against the requirement's budget at elaboration. We also answer a question to the AI: can technology information be combined with the types of operations to estimate the needed delays? Yes, to first order: probe syntheses of nine common operations on the open ASAP7 7 nm library, and a simple logic-level model calibrated on one of them, estimate every delay within 0 to +49%, which is enough to choose the cycle count before the RTL exists. The paper then works through the two fixes (a pipeline, and a multicycle path with a `rdy/ack` handshake), with parameterized RTL and SVA simulated at four clock periods.

1. The questions

- Can I add ready signals in an FSM? I have a set of English requirements with a need for a 2 ns clock rate. AI creates the RTL, but one path is 4 ns. Can AI change the RTL by adding a `rdy` signal to allow for the 4 ns path, and how is that reflected in the SVA?
- More fundamentally: the technology and the clock rate must be considered when the requirements are translated into SVA, so latency assertions must be parameterized. Can AI combine technology information with the types of operations to arrive at an estimate of the needed delays?

2. The example

Take a requirement of the kind that produces the problem: *when req is asserted, y = a·b + c is presented with ack; the clock is 500 MHz (2 ns).* With no technology in view, the first SVA is `req |-> ##1 ack`. In a slow technology, the multiply-add needs about 3.6 ns of logic and wire delay, plus about 0.2 ns of register overhead: 3.8 ns against a 2 ns period, about −1.8 ns of slack.

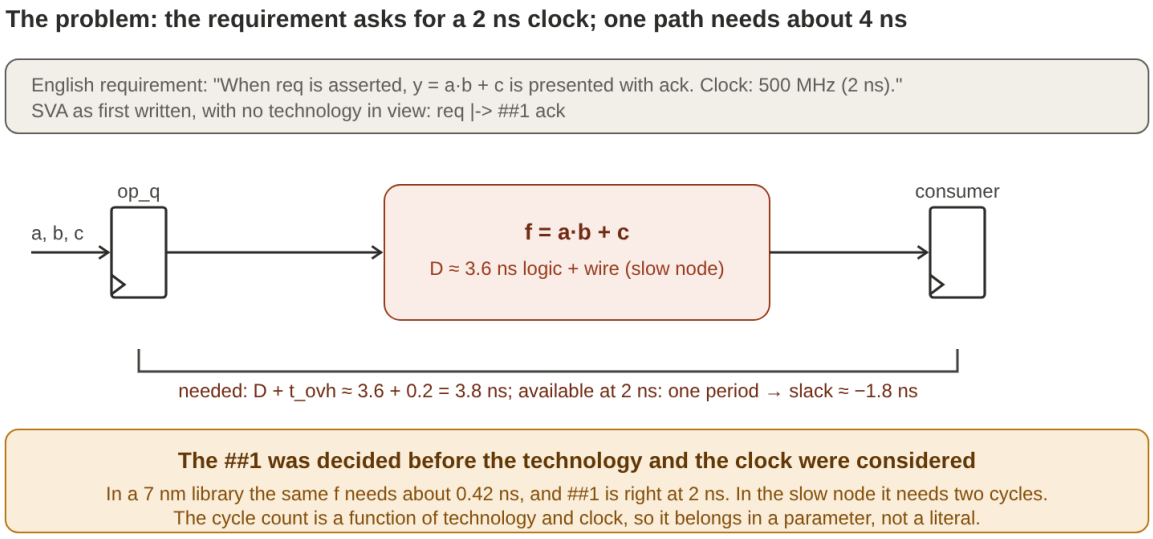


Figure 1. The problem. The ##1 was decided before the technology and the clock were considered.

The same function in the ASAP7 7 nm library needs about 0.42 ns (Section 4), so at 2 ns the original ##1 is correct. The requirement did not change; the technology did. That is the core point of this paper.

3. Technology and clock rate belong in the translation

When a requirement is translated into SVA, every cycle count answers the question: *how many clock periods does this behavior take?* That answer has three inputs, and only one of them is in the English text:

- **The requirement:** the function and its budget. A flexible requirement states a range (ack within 1 to 4 cycles) or a time (ack within 8 ns); a fixed one states an exact cycle.
- **The technology:** the delay D of the logic between registers, which depends on the operation types and widths, and the register overhead t_ovh (clock-to-Q, setup and margins).
- **The clock period** T_clk, which the requirement or the system sets.

The cycle count is then a calculation, not a choice:

```
LAT = ceil( D / (T_clk - t_ovh) )        cycles the path needs at this clock
LAT <= budget                              or the requirement cannot be met as written
```

This leads to three rules for the translation step:

1. **No literal cycle counts for timing-dependent behavior.** Write ##LAT, not ##2. A literal is right for one technology-and-clock pair only, and it silently becomes wrong when either changes, as it does at a node migration or a speed-grade change (see *Your Chip Moved from 5 nm to 3 nm. Did Your SVA Notice?*).
2. **One package holds every number.** The technology delay, the overhead, the clock and the budget live in one package (`lat_pkg` here, `fmax_lat_pkg` in *Which Assertion Limits Your Clock?*). The RTL, the SVA and the STA constraint all read LAT from it, so they change together and never disagree.
3. **The budget is checked at elaboration.** If the technology and the clock push LAT past what the requirement allows, compilation stops with a message. A budget violation is a specification decision (RQM0), never a silent change of the assertion.

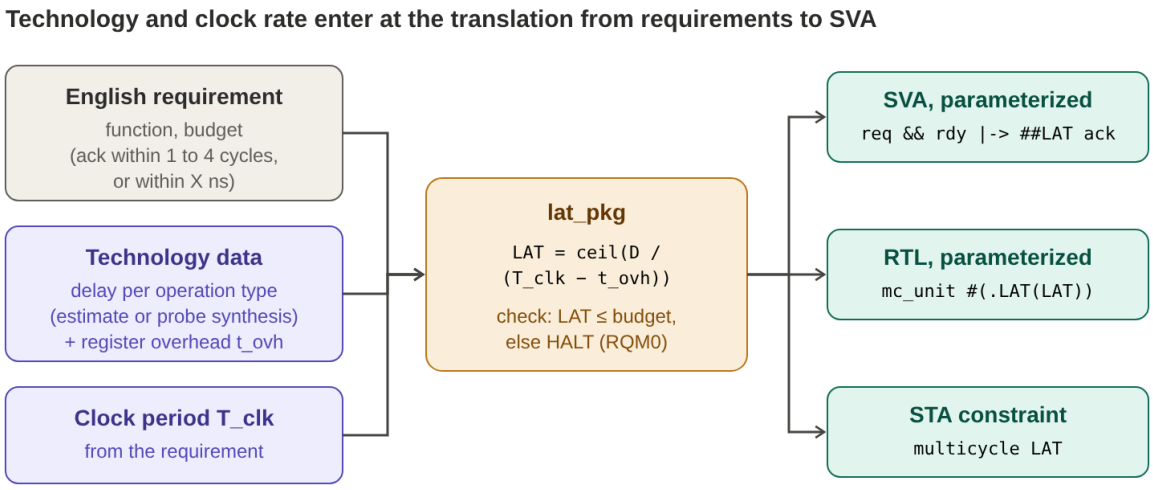


Figure 2. Technology and clock enter at the translation. One package computes LAT and feeds the SVA, the RTL and the STA constraint.

A requirement stated in time converts the same way: *ack within 8 ns* at a 2 ns clock is a budget of 4 cycles; at 1.5 ns it is 5. The SVA still uses cycles, because SVA counts clock ticks, but the number comes from the package.

4. Can AI estimate delays from technology and operation types?

Yes, to first order, and early enough to matter: before the real RTL exists, at the moment the requirement is translated. The method has four steps.

1. **Decompose the requirement into operation types and widths.** `y = a·b + c` with 8-bit `a`, `b` and 16-bit `c` is one 8×8 multiplier and one 16-bit adder, which a synthesizer merges into one multiply-add tree.
2. **Look up or estimate each operation's delay in the target technology.** Two sources: a per-technology table built once by probe synthesis (measured), or a logic-level model (estimated). Adders, incrementers and comparators grow with $\log_2$ of the width; a multiplier is a reduction tree of about $2 \cdot \log_{1.5}(n)$ gate levels plus a final adder; a k-input mux grows with $\log_2 k$.
3. **Add the wire allowance and the register overhead.** Before layout, wires are an assumed fraction of the logic delay (20% in our flow); t_ovh comes from the library's flip-flop (about 164 ps in ASAP7, including assumed margins).
4. **Compute LAT, then confirm with one probe run.** Throwaway AI-generated RTL, one synthesis and STA run, and the measured D replaces the estimate in the package. The flow is described in *Which Assertion Limits Your Clock?*.

Measured and estimated delays in ASAP7

We ran probe syntheses of nine common operations with Yosys 0.33 and ABC on the ASAP7 predictive 7 nm library (RVT, typical corner, combinational cells only, delay-driven mapping). For comparison, a logic-level model with one constant τ, calibrated on the 32-bit adder (209 ps over 7 levels, τ = 29.8 ps per level), estimated the others:

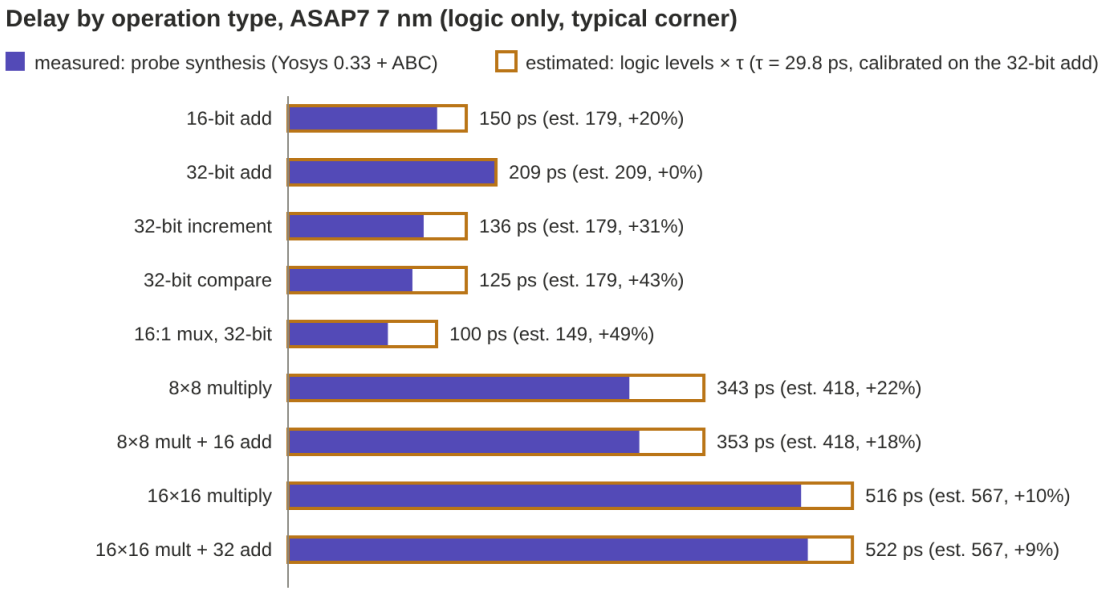


Figure 3. Delay by operation type in ASAP7: measured by probe synthesis, and estimated from logic levels.

Operation	Levels (model)	Measured (ps)	Estimated (ps)	Estimate error
16-bit add	6	150	179	+20%
32-bit add (calibration)	7	209	209	0%
32-bit increment	6	136	179	+31%
32-bit compare	6	125	179	+43%
16:1 mux, 32-bit	5	100	149	+49%
8×8 multiply	14	343	418	+22%
8×8 multiply + 16-bit add	14	353	418	+18%
16×16 multiply	19	516	567	+10%
16×16 multiply + 32-bit add	19	522	567	+9%

Three observations. First, the estimate is consistently conservative (0 to +49%), which is the right direction for choosing LAT: it may ask for a cycle too many, rarely one too few. Second, the multipliers dominate, and adding + c to a multiply costs almost nothing (343 to 353 ps), because the synthesizer folds the addend into the reduction tree; a model that summed the two delays would overestimate badly. Third, the error matters only near a cycle boundary, because LAT rounds up: for the 8×8 multiply-add at a 2 ns clock, both the measured 353 ps and the estimated 418 ps give LAT = 1.

Applied to the example: D = 353 ps of logic plus 20% wire is about 424 ps, and with t_ovh = 164 ps the multiply-add needs 1 cycle at 2 ns, 1 at 0.6 ns, 2 at 0.4 ns and 4 at 0.3 ns. In the slow technology of Figure 1 (D = 3.6 ns, t_ovh = 0.2 ns), it needs 1 cycle at 4 ns, 2 at 2 ns, 3 at 1.5 ns and 5 at 1 ns.

Same requirement, same function: the cycle count depends on technology and clock

Slow node (illustrative): D = 3.6 ns, t_ovh = 0.2 ns

T_clk	4 ns	3 ns	2 ns	1.5 ns	1.25 ns	1 ns
LAT = ##n	**##1**	**##2**	**##2**	**##3**	**##4**	**##5**
budget 4	ok	ok	ok	ok	ok	HALT

ASAP7 7 nm: D = 0.42 ns (0.35 probe + 20% wire), t_ovh = 0.16 ns

T_clk	2 ns	1 ns	0.6 ns	0.4 ns	0.3 ns	0.25 ns
LAT = ##n	**##1**	**##1**	**##1**	**##2**	**##4**	**##5**
budget 4	ok	ok	ok	ok	ok	HALT

A literal ##1 or ##2 is right for one technology-and-clock pair only. LAT from the package is right for all of them, and the budget check turns "the clock is too fast for this requirement" into an elaboration-time stop instead of a silent change of the specification.

Figure 4. Same requirement, same function: the cycle count depends on technology and clock. HALT marks a LAT above the requirement's budget of 4.

So the answer to the question is yes, with a stated limit: AI can combine technology data with operation types to estimate delays well enough to set the cycle counts in the SVA at translation time, and to flag the paths that will not fit. It cannot replace the probe synthesis and STA that confirm the numbers, and it should never present an estimate as a measurement. The numbers above are logic delay before placement; real paths add placement, wire and corner effects, which is why the package carries an explicit wire allowance and margins and why one probe run follows.

5. Classify the requirement before changing the RTL

When the package reports LAT > 1 for a path the requirement wrote as ##1, the RTL can be changed in several ways, but each changes something the requirement may have fixed. Classify the requirement on that path first (*Which Assertion Limits Your Clock?*):

- **Fixed:** something outside depends on the exact cycle. Neither a pipeline nor a `rdy` is allowed without changing the contract with the other side.
- **Flexible:** the consumer waits for a `valid`, a `ready` or a tag. A pipeline or a ready handshake is legal, up to the budget.
- **Loop:** the next result depends on the previous one. A register inside the loop changes the function; a `rdy` helps only by lowering throughput (initiation interval II = LAT).

Unless the requirement is already flexible, the change is a specification decision, and under the RAG-SVA clarification rule (RQM0) the AI stops and asks with options: pipeline, add a handshake, relax the clock, or restructure the logic. The engineer's answer becomes a written requirement.

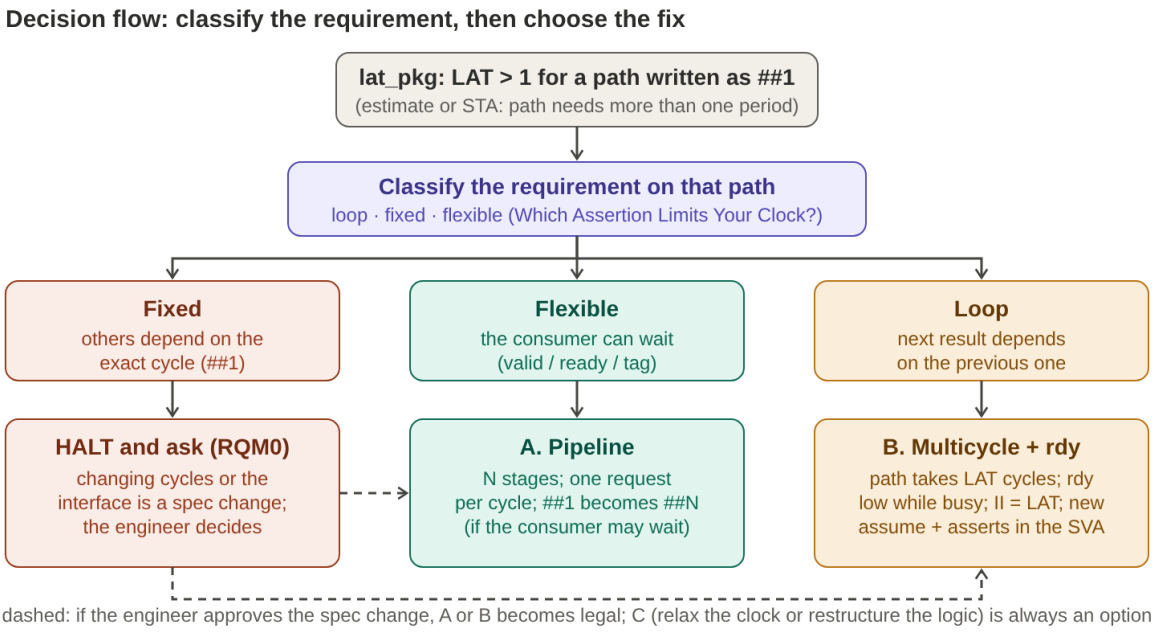


Figure 5. Decision flow: classify the requirement on the slow path, then choose the fix.

6. Option A: pipeline

If the path is not a loop, split it into N stages, with N from the package and a valid bit traveling with the data. Throughput stays at one request per cycle; latency becomes N cycles; no `rdy` is needed. The SVA becomes `req |-> ##N ack`, legal while N is within the budget.

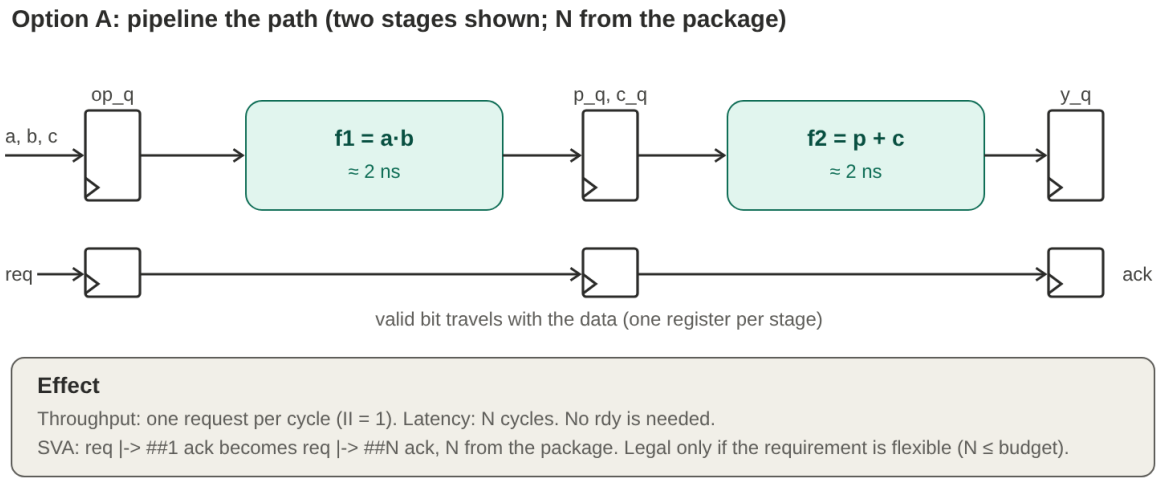


Figure 6. Option A: pipeline stages and a valid bit.

7. Option B: a multicycle path with rdy and ack

When the path should not be split, give it LAT clock periods instead. The design holds its operands for LAT cycles; while busy it lowers `rdy` so the environment does not send a new request; `ack` comes LAT cycles after acceptance. This is the "add a ready signal" idea, now parameterized.

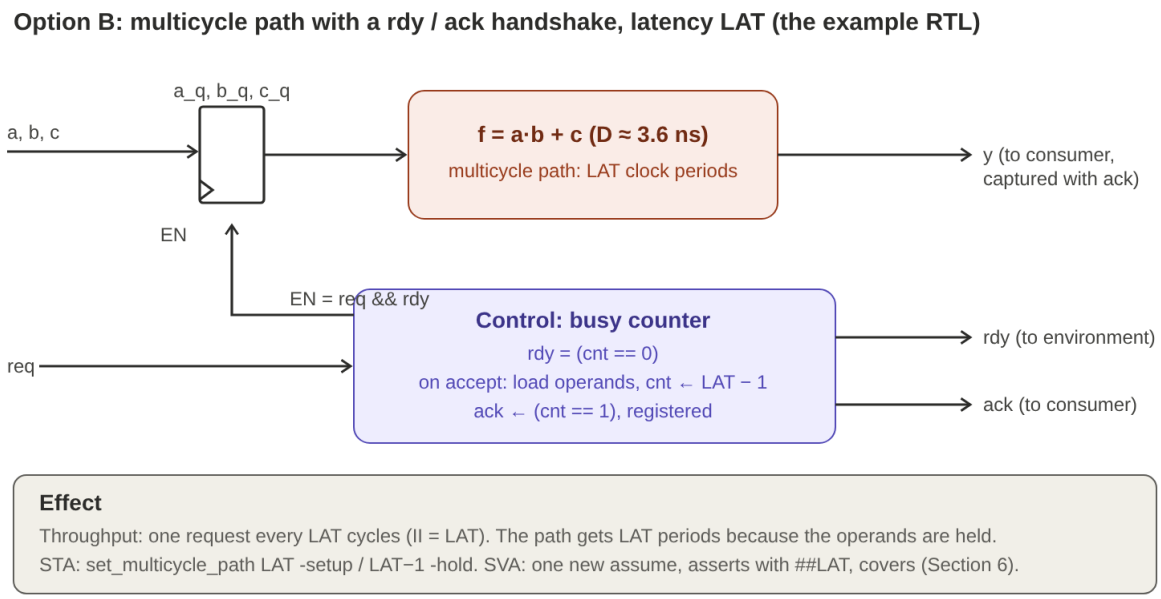


Figure 7. Option B: operand registers loaded only on req && rdy, a busy counter, and a LAT-cycle path to the consumer.

The latency package

```
// lat_pkg.sv : latency from technology and clock -- one place for every number
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// Ben Cohen, Kais Chibani -- systemverilog.us
//   LAT = ceil( D_F_PS / (T_CLK_PS - T_OVH_PS) )     cycles the path f needs at this clock
//   D_F_PS   : logic + wire delay of f = a*b + c in the target technology (estimate or probe)
//   T_OVH_PS : register overhead per stage (clock-to-Q + setup + margins)
// Override from the command line, e.g. +define+T_CLK_PS=2000 +define+D_F_PS=3600
+define+T_OVH_PS=200
`ifndef T_CLK_PS
  `define T_CLK_PS 2000
`endif
`ifndef D_F_PS
  `define D_F_PS 3600
`endif
`ifndef T_OVH_PS
  `define T_OVH_PS 200
`endif
`ifndef LAT_BUDGET
  `define LAT_BUDGET 4          // requirement: flexible, ack within 1 to 4 cycles
`endif
package lat_pkg;
  localparam int T_CLK_PS   = `T_CLK_PS;
  localparam int D_F_PS     = `D_F_PS;
  localparam int T_OVH_PS   = `T_OVH_PS;
  localparam int LAT_BUDGET = `LAT_BUDGET;
  localparam int USABLE_PS  = T_CLK_PS - T_OVH_PS;
  localparam int LAT        = (D_F_PS + USABLE_PS - 1) / USABLE_PS;   // ceil
  localparam int II         = LAT;                                    // multicycle: one request
per LAT cycles
endpackage
```

The RTL

```
// mc_unit.sv : multicycle unit with rdy/ack, latency LAT from lat_pkg
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// Ben Cohen, Kais Chibani -- systemverilog.us
// f = a*b + c runs from the operand registers to the consumer; it is given LAT clock
// periods (STA: set_multicycle_path LAT -setup, LAT-1 -hold). LAT = 1 is a plain register path.
module mc_unit #(parameter int LAT = lat_pkg::LAT) (
  input  logic        clk, rstn,
```

```
  input  logic        req,          // environment: request (only when rdy)
  input  logic [7:0]  a, b,
  input  logic [15:0] c,
  output logic        rdy,          // design: can accept a request this cycle
  output logic        ack,          // design: y is valid this cycle
  output logic [15:0] y
);
  localparam int CW = (LAT > 1) ? $clog2(LAT) : 1;
  logic [CW-1:0] cnt;               // busy cycles remaining
  logic [7:0]  a_q, b_q;            // operand registers, held while busy
  logic [15:0] c_q;
  logic        accept;

  assign rdy    = (cnt == '0);
  assign accept = req && rdy;
  assign y      = 16'(a_q) * 16'(b_q) + c_q;   // the long path: LAT clock periods

  always_ff @(posedge clk or negedge rstn)
    if (!rstn) begin
      cnt <= '0; ack <= 1'b0; a_q <= '0; b_q <= '0; c_q <= '0;
    end else begin
      if (accept) begin
        a_q <= a; b_q <= b; c_q <= c;
        cnt <= CW'(unsigned'(LAT - 1));
      end else if (cnt != '0) begin
        cnt <= cnt - 1'b1;
      end
      ack <= (LAT == 1) ? accept : (cnt == CW'(unsigned'(1)));
    end
endmodule
```

The parameterized SVA

```
// mc_unit_sva.sv : parameterized checker for mc_unit (latency and throughput = LAT)
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// Ben Cohen, Kais Chibani -- systemverilog.us
// PORT-INTENT: req, a, b, c = environment inputs (assume)
//              rdy, ack, y  = design outputs (assert)
// All cycle counts come from lat_pkg: technology delay and clock period decide LAT.
module mc_unit_sva #(parameter int LAT = lat_pkg::LAT, parameter int LAT_BUDGET =
lat_pkg::LAT_BUDGET) (
  input logic clk, rstn, req, rdy, ack,
  input logic [7:0] a, b, input logic [15:0] c, y,
  input logic [7:0] a_q, b_q, input logic [15:0] c_q
);
  // Requirement budget: the clock and technology must not push LAT past what the spec allows
  if (LAT > LAT_BUDGET) begin : g_budget
    $error("LAT=%0d exceeds the requirement budget %0d: spec decision needed (RQM0)", LAT,
LAT_BUDGET);
  end

  // REQ-07a (environment): a request is issued only when the design is ready
  // DOC9: interface rule introduced by the rdy handshake (spec change, approved)
  mp_req_when_rdy: assume property (@(posedge clk) disable iff (!rstn)
    req |-> rdy);
  // REQ-07b (design): an accepted request is answered exactly LAT cycles later
  ap_ack_lat: assert property (@(posedge clk) disable iff (!rstn)
    req && rdy |-> ##LAT ack);
  // REQ-07d (design): ack only for an accepted request
  ap_no_spurious_ack: assert property (@(posedge clk) disable iff (!rstn)
    ack |-> $past(req && rdy, LAT));
  // REQ-07e (design): the result is the function of the accepted operands
  ap_result: assert property (@(posedge clk) disable iff (!rstn)
    req && rdy |-> ##LAT (y == 16'($past(a, LAT)) * 16'($past(b, LAT)) + $past(c, LAT)));

  if (LAT > 1) begin : g_multi
    // REQ-07c (design): busy for LAT-1 cycles, then ready (II = LAT)
    ap_busy: assert property (@(posedge clk) disable iff (!rstn)
      req && rdy |-> ##1 (!rdy)[*LAT-1] ##1 rdy);
```

```
    // Timing contract: operands held through the capture edge (guards the multicycle
constraint).
    // Loaded ON the accepting edge, so the hold is checked from ##2 to ##LAT.
    ap_op_stable: assert property (@(posedge clk) disable iff (!rstn)
      req && rdy |-> ##2 ($stable(a_q) && $stable(b_q) && $stable(c_q))[*LAT-1]);
  end else begin : g_single
    ap_always_ready: assert property (@(posedge clk) disable iff (!rstn)
      req && rdy |-> ##1 rdy);
  end

  // Covers
  cp_req_when_rdy_hit: cover property (@(posedge clk) disable iff (!rstn) req && rdy);
  cp_back_to_back:     cover property (@(posedge clk) disable iff (!rstn) req && rdy ##LAT req &&
rdy);
  cp_idle_gap:         cover property (@(posedge clk) disable iff (!rstn) rdy && !req ##1 rdy &&
req);
endmodule
```

What changed from a fixed-latency checker:

1. **Every cycle count is `LAT`.** ##LAT, `$past(..., LAT)` and `[*LAT-1]` replace the literals. The checker is the same file at every clock.
2. **The budget check is part of the checker.** The generate-time `$error` fires when LAT exceeds the requirement's budget, so a clock or technology that breaks the requirement stops the build instead of quietly changing the contract.
3. **The trigger is `req && rdy`,** and a new assumption (`mp_req_when_rdy`) states the environment's side of the handshake, with a DOC9 header and a cover.
4. **The structure adapts to LAT.** For LAT > 1, `ap_busy` states II = LAT and `ap_op_stable` checks the operands from ##2 through ##LAT (they are loaded on the accepting edge, so the hold starts one cycle later). For LAT = 1 there is no busy time, and `ap_always_ready` states that instead.
5. **`ap_op_stable` guards the timing exception.** The multicycle constraint in STA is safe only if the operands are held; STA cannot check that, the assertion does.

A common mistake is `req |-> ##2 rdy && ack`. It is legal SVA, but it hard-codes the latency and puts `rdy` in the wrong role: `rdy` should gate the request, not be an obligation in the consequent.

8. Same RTL and SVA, two clocks

Figure 8 shows the same design and checker, with nothing edited between the runs except `T_CLK_PS` in the package. At 2 ns, LAT = 2: requests accepted at edges 3, 5 and 7 are answered at 5, 7 and 9, with `rdy` low for one cycle after each acceptance. At 1.5 ns, LAT = 3: acceptances at 3, 6 and 9 are answered at 6, 9 and 12, with `rdy` low for two cycles. The results (17, 101, 56) are the same; only the cycle count moved.

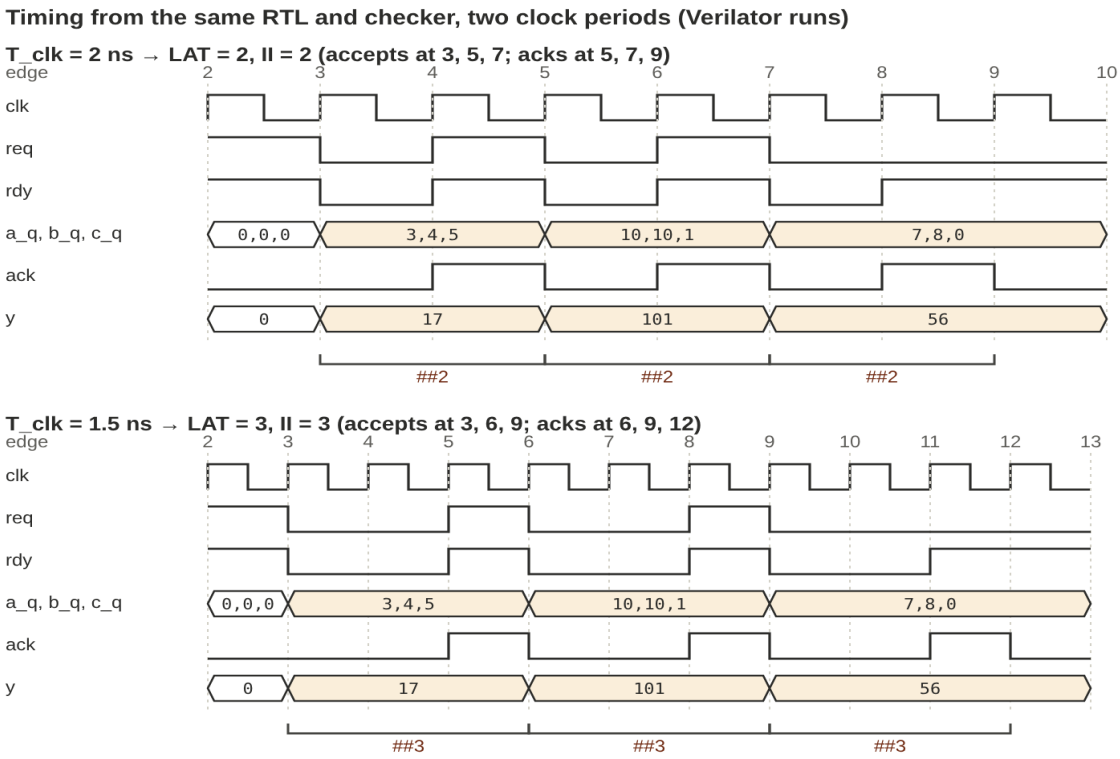


Figure 8. Timing at T_clk = 2 ns (LAT = 2) and 1.5 ns (LAT = 3), from Verilator runs of the same files.

9. The timing constraint follows the same parameter

A LAT-cycle path must be declared to STA, with the same LAT:

```
set LAT [lat_from_package]       ;# same value the RTL and the SVA use
set_multicycle_path $LAT -setup -from [get_cells {a_q_reg* b_q_reg* c_q_reg*}] -to [get_cells
{consumer_y_reg*}]
set_multicycle_path [expr {$LAT-1}] -hold -from [get_cells {a_q_reg* b_q_reg* c_q_reg*}] -to
[get_cells {consumer_y_reg*}]
```

The constraint is a promise that the operands do not change for LAT cycles. STA believes the promise; `ap_busy` and `ap_op_stable` check it. Cell names are illustrative, and generating the LAT line from the package (rather than typing it) keeps the three artifacts in step.

10. The loop case

If the slow path is the FSM's own next-state logic, a register inside the loop changes the function. Option B's shape still applies, but it now means the FSM changes state only every LAT cycles, which is a throughput decision for the specification. The alternative is to shorten the loop (state re-encoding, precomputed conditions, splitting the FSM). The technology matters here too: a loop that is too slow in one library may close in one cycle in another, so its budget check belongs in the package as well.

11. The read-back

After the change, the read-back states the requirement in terms of LAT, and the package states LAT for the chosen technology and clock:

Item	Intention	Requirement	Property
REQ-07a	Assume (environment)	A request is issued only when rdy is high.	mp_req_when_rdy
REQ-07b	Obligation	An accepted request is answered with ack exactly LAT cycles later.	ap_ack_lat
REQ-07c	Obligation	rdy is low for LAT−1 cycles after acceptance, then high (II = LAT).	g_multi.ap_busy
REQ-07d	Obligation	ack occurs only for an accepted request.	ap_no_spurious_ack
REQ-07e	Obligation	y equals a·b + c of the accepted operands when ack is high.	ap_result
Timing	Obligation	Operands are held through the capture edge (guards the multicycle constraint).	g_multi.ap_op_stable
Budget	Elaboration check	LAT ≤ 4 for the chosen technology and clock.	g_budget
LAT	Package	LAT = ceil(D / (T_clk − t_ovh)); 2 at 2 ns in the slow node, 1 at 2 ns in ASAP7.	lat_pkg

12. Tool results

- **Compile:** slang (pyslang 12.0.0): package, RTL, checker and testbench, 0 diagnostics.
- **Simulator:** Verilator 5.055 (development build from source, November 2026), because the installed 5.020 does not support ##N inside properties. The assertions were evaluated by its SVA engine.
- **Slow-node model** (D = 3,600 ps, t_ovh = 200 ps), 3 seeds × 2,000 cycles per clock, random environment that respects `rdy`: T_clk 4 ns (LAT 1): 4,438 requests accepted and answered; 2 ns (LAT 2): 2,555; 1.5 ns (LAT 3): 1,800; 1 ns (LAT 5): 1,124. 0 result mismatches, 0 assertion failures, all three covers hit at every clock.
- **Budget check:** at 1 ns, LAT = 5 exceeds the budget of 4 and the generate-time `$error` was raised at elaboration ("LAT=5 exceeds the requirement budget 4: spec decision needed"). The run above was made with that error demoted to a warning, to show the design still works at LAT = 5; in a normal build it stops.
- **ASAP7 values** (D = 424 ps, t_ovh = 164 ps), seed 1: T_clk 2 ns (LAT 1), 0.4 ns (LAT 2) and 0.3 ns (LAT 4): 1,476, 848 and 461 requests, 0 mismatches, 0 assertion failures.
- **Probe synthesis:** Yosys 0.33 + ABC, ASAP7 RVT TT NLDM libraries (SIMPLE, INVBUF, AO, OA cells), delay-driven mapping; logic delay only, before placement. The 164 ps overhead and 20% wire allowance are taken from *Which Assertion Limits Your Clock?* and include assumed margins.
- **Mutation tests (supplement, not an SVA engine result):** at LAT = 3, `rdy` tied high is caught by `g_multi.ap_busy`; a busy count one cycle short is caught by `ap_no_spurious_ack`.
- **A lesson kept from the first version of this paper:** `ap_op_stable` was first written with ##1, which fails because the operands are loaded on the accepting edge. Running the checker found it; the property now starts at ##2.
- **Not run:** formal verification, and STA of the example RTL. The slow-node delays are illustrative; the ASAP7 delays are measured probe results.

13. Conclusion

The question started as "can AI add a `rdy` to meet the clock?" The answer is yes, as a specification decision that the AI raises rather than takes. The larger lesson is about translation. A cycle count in an assertion is the delay of a function in a technology, divided by a clock period; it is not in the English text, and writing it as a literal bakes one technology and one clock into the verification. Writing latency as a parameter, computed in one package from the technology delay, the overhead and the clock, and checked against the requirement's budget at elaboration, keeps the RTL, the SVA and the timing constraints in step when either changes. And AI can supply the technology side early: combining operation types and widths with per-technology delay data, it estimates path delays within tens of percent before any RTL exists, enough to set the cycle counts and flag the paths that need a decision, with one probe synthesis to confirm.

References

- B. Cohen, K. Chibani, *Which Assertion Limits Your Clock?*, systemverilog.us/Which_Assertion_Limits_Your_Clock.pdf.
- B. Cohen, K. Chibani, *Your Chip Moved from 5 nm to 3 nm. Did Your SVA Notice?*, systemverilog.us/Same_Design_New_Node.pdf.
- B. Cohen, K. Chibani, *Using RAG-RTL with RAG-SVA: A Pipelined Multiplier Example*, systemverilog.us/UsingRagRtlWithRagSVA4Multiplier.pdf.
- B. Cohen, K. Chibani, *Knowledge-Driven AI for SystemVerilog Assertions* (RAG-SVA), Cohen Publishing, 2026, systemverilog.us.
- ASAP7: 7 nm predictive process design kit (Arizona State University and ARM); standard-cell libraries as distributed with OpenROAD-flow-scripts.
- Yosys Open SYnthesis Suite 0.33, with ABC (Berkeley).
- Related papers: systemverilog.us/#papers.

RAG-SVA: personal/academic use per book purchase. Commercial use requires a Commercial License from Cohen Publishing (systemverilog.us).