Which Assertion Limits Your Clock?

Classifying Every SVA Requirement in Clock Cycles, and the Recommended Flow from Throwaway AI-Generated RTL and STA to the Fastest Verified Clock

---

Ben Cohen, Dr. Kais Chibani

Cohen Publishing

systemverilog.us

September 2026

Abstract

An assertion such as `req |-> ##1 ack` records a decision; it does not explain it. Was the single cycle demanded by a bus protocol, by a feedback loop, or chosen arbitrarily? The answer decides whether pipelining can ever help that path, and therefore how fast the design can run.

This paper gives a practical way to decide. Every requirement is classified, before any design work, as a **loop** (the next result depends on the previous one), **fixed** (something outside the block depends on the exact cycle), or **flexible** (the consumer waits for a valid, ready, or tag). Three questions settle the class; examples cover counters, accumulators, state machines, memories, bus protocols, and the easily missed case of a flag, such as a FIFO's full, that forms a loop through another block. We then show how to find loops mechanically in the SVA, in the RTL, and across interfaces.

The classification feeds the flow we recommend, built from our recent papers: write the SVA in cycles; generate throwaway probe RTL with AI; run STA once; let a parameter package (Appendix C) compute the maximum frequency and the limiting assertion; examine the real pipeline cut of any path near the limit; decide at the specification level; then build the real design, verify every legal pipeline depth, and let final STA choose one. Measured results from an open 7 nm library, scaled to 5 nm-class and 3 nm-class by stated assumptions, show why the classification matters: a one-cycle counter loop limits the clock in every technology, while a multiply-add almost three times slower does not.

Disclosure. Drafted with Claude (Anthropic) under the authors' direction. The timing numbers come from an open-source flow on the ASAP7 predictive 7 nm library; the 5 nm-class and 3 nm-class values, register margins, and wire allowance are assumptions. The example properties were parsed and elaborated, not formally proven.

Contents

1. Why classification comes first

Three assertions can look alike and mean very different things:

```
inc    |-> ##1 (count == $past(count) + 1);   // a counter
rd_en  |-> ##2 rvalid;                        // a memory with fixed read timing
in_valid |-> strong(##[1:4] out_valid);       // a pipelined arithmetic unit
```

The first cannot be pipelined without changing what it computes. The second can be pipelined only if the external protocol changes. The third can be pipelined freely within its four-cycle budget. When the question is how fast the design can run, these three behave completely differently, yet the operators alone do not say why. The class comes from the **requirement behind the assertion**, so it has to be established when the requirement is written, not guessed later from the code.

2. The three classes

Class	Meaning	Typical SVA	Can pipelining help?	Minimum clock period [i]
Loop	the next result depends on the previous result	`##1` with `$past` of its own output	no; budget set by throughput	t_ovh + ceil(D / II)
Fixed	something outside depends on the exact cycle	`##N`	only by changing the protocol	t_ovh + ceil(D / N)
Flexible	the consumer waits for valid, ready, or a tag	`##[1:B]`	yes, up to B stages	t_ovh + ceil(D / B)

Here D is the path's logic delay in the chosen technology, t_ovh the per-stage register overhead (clock-to-Q, setup, and margins), N the fixed cycle count, B the flexible budget, and II the loop's initiation interval: how many cycles it may take before accepting the next input. "ceil" is short for **ceiling**; it rounds a number **up** to the next whole number.

The design's maximum frequency is set by the path with the largest minimum period. That path names the assertion to work on.

3. Three questions that decide the class

3.1 Question 1: does the next result depend on the previous result?

If yes, the path is a **loop**. Adding a register inside the loop changes the function: the next computation would use a stale value. Examples:

- **Event counter:** `count + 1` needs the current count.
- **Accumulator:** `sum + x` needs the current sum.
- **State machine:** the next state depends on the current state.
- **CRC over a stream, one word per cycle:** the next CRC depends on the current CRC.
- **Round-robin arbiter:** the next priority depends on the last grant.

A loop's budget comes from **throughput**, not latency. The question to ask is: how often must it accept new data?

- One input per cycle: the loop must close in one cycle (II = 1).
- One input every two cycles: the loop has two cycles (II = 2), either as a two-cycle path or by splitting the computation, because no new input arrives in between.

In SVA, the throughput is an environment assumption, and the budget follows from it:

```
// One input every two cycles (environment) ...
acc_valid |-> ##1 !acc_valid;                          // assumed

// ... so the accumulator may take two cycles
(acc_valid, v_sum = sum + x) |-> ##2 (sum == v_sum);   // asserted
```

Changing a loop's throughput requirement is a specification decision, and it is one of the few ways to relax a loop.

3.2 Question 2: does anyone outside the block depend on the exact cycle?

If yes, the path is **fixed**. Typical signs:

- **A protocol with fixed timing:** a memory interface where read data appears exactly two cycles after the address.
- **A neighbour that samples without a valid bit:** it reads the result on a specific cycle and trusts it.
- **Software or firmware** that counts on a cycle-exact response.
- **A standard** that defines the timing.

```
rd_en |-> ##2 rvalid;     // the interface protocol fixes the second cycle
```

A fixed path can gain a pipeline stage only if the external agreement changes, which means changing the other side too.

3.3 Question 3: for a flexible path, where does B come from?

If the consumer waits for a valid, ready, or tag, the exact cycle does not matter, and the path is **flexible**:

```
in_valid |-> strong(##[1:B_MAD] out_valid);   // B_MAD from the system, not from the logic
```

B must come from the system:

- **An end-to-end latency budget:** for example, a processor pipeline that allows four cycles for its multiply unit.
- **Buffer or credit sizes:** the number of operations in flight equals throughput times latency. With 8 return-buffer entries and one request per cycle, a latency above 8 cycles stalls the requester, so B ≤ 8 preserves full throughput.
- **A performance target:** the maximum latency the application tolerates.

If nobody can say where B comes from, the requirement is ambiguous. That is exactly what the RAG-SVA ambiguity gate (RQM0) is for: halt and ask the architect, rather than pick a number.

4. Loops in disguise: flags that close through another block

Some loops cross a block boundary and are easy to miss. A FIFO's `full` flag looks like a simple output, but the writer uses it to decide whether to push, and the push changes the flag. Together they form a loop:

```
full |-> !push;                  // environment: the writer reacts in the same cycle
(fifo_count == DEPTH) |-> full;  // design: full must be exact, now
```

If STA reports that `full` is too slow and the designer registers it one cycle later, the writer may push once more after the FIFO is already full, and data is lost. Registering the flag is not a timing fix; it needs a specification change: an `almost_full` flag raised one entry earlier, so that the one late cycle still has room.

```
// With a registered warning flag, warn one entry early
(fifo_count >= DEPTH - 1) |-> ##1 (almost_full || fifo_count < DEPTH - 1);
```

The same pattern appears in any flag that its receiver uses to decide its next action: `ready`, `busy`, `grant`, credit counters. Each is a loop in disguise, and each one's timing is part of the interface contract.

5. Finding loops mechanically

Loops can be found in each artifact:

- **In the SVA:** a property that refers to the previous value of its own output (`$past(count)`), or to a state variable, describes a loop.
- **In the probe or real RTL:** an output that feeds back to its own module's input, such as `count` → `count + 1` → `count`, or a state register whose next value is computed from itself.
- **Across blocks:** a handshake flag that the other side uses to decide its next action, as in Section 4.

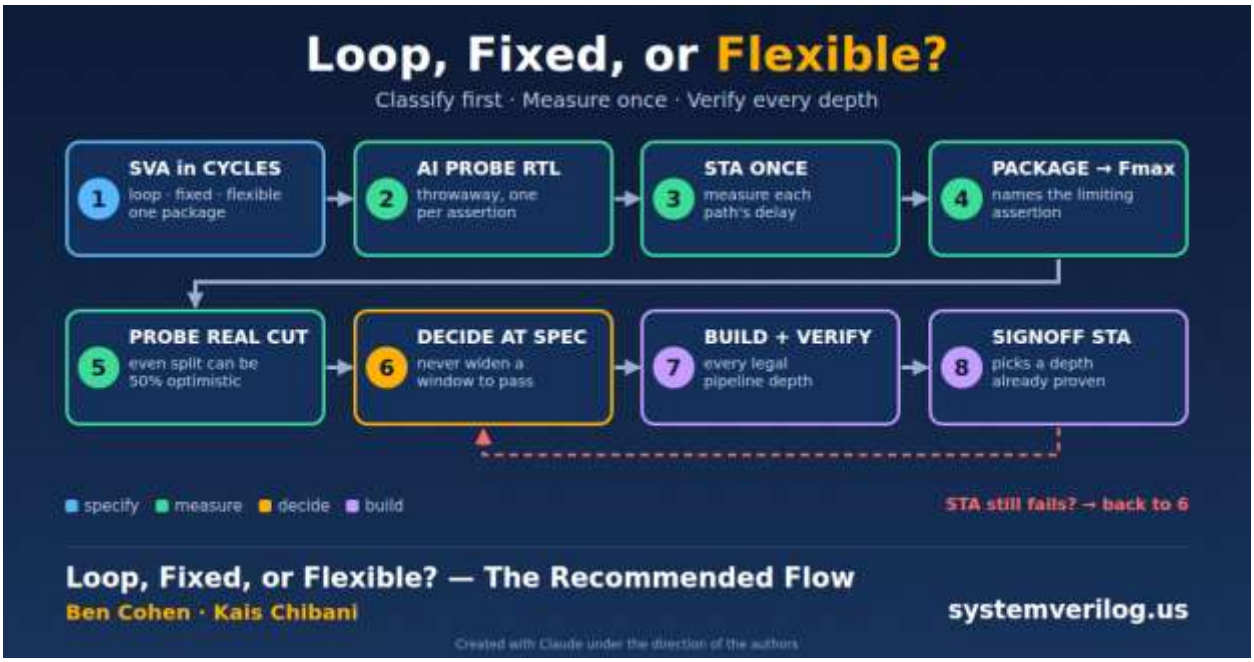
Once found, each loop's throughput requirement should be written as an explicit assumption, so that its budget is visible.

6. Classification examples

Requirement	Class	Why	Budget
Event counter, one event per cycle	loop	next count needs this count	1 cycle (II = 1)
Accumulator, one sample every 2 cycles	loop	next sum needs this sum	2 cycles (II = 2)
State machine next state	loop	next state needs this state	1 cycle
CRC, one word per cycle	loop	next CRC needs this CRC	1 cycle
SRAM read with fixed 2-cycle latency	fixed	the protocol samples that cycle	exactly 2
Register read on a fixed-timing bus	fixed	the bus samples the next cycle	exactly 1
Multiply-add with `out_valid`	flexible	the consumer waits for valid	from latency budget, e.g. 4
Address decode consumed with its valid	flexible	no one samples a fixed cycle	from system, e.g. 2
Memory reads with 8 return credits	flexible	tag or valid identifies data	up to 8 at full throughput
FIFO `full` flag	loop through the writer	the writer acts on it immediately	1 cycle, unless the spec changes to `almost_full`

7. The recommended flow

This is the flow we recommend, assembled from the demonstrations in our recent papers.



1. **Write the SVA in cycles.** Classify every path as loop, fixed, or flexible, and give it a cycle budget. Write two tiers: contract assertions that use only the budget, and implementation assertions that use the pipeline depth. Keep every number in one package.
2. **Generate throwaway probe RTL with AI.** One unpipelined module per SVA path, named after its assertion. It need not be good design, but it must be honest: complete, with exact widths, every output used. Check it before timing it.
3. **Synthesize and run STA once, at the slow setup corner.** Record each path's delay and the register overhead in the package, marked as probe measurements.
4. **Let the package do the frequency sweep.** Each path's minimum period follows from its class (Section 2). The largest sets Fmax and names the limiting assertion. The package also reports each path's cycle margin, which reveals paths that cannot take one more register even when they have slack.
5. **Examine the limiting paths, and those near the limit.** Probe their real pipeline cut, not just the even split, and try a better structure for the probe.
6. **Decide at the specification level.** For each limiting path: restructure the logic, give it more cycles or a lower throughput if the system allows, change a fixed interface to a flexible one, or accept the lower frequency. Never widen an assertion window just to make timing pass.
7. **Build the real design.** With RAG-RTL, pipeline depth as a parameter from the package. Verify every legal depth up front: the contract assertions are the same file for all of them; only the exact-latency assertion is generated per depth.
8. **Run final synthesis and signoff STA** at the chosen frequency. STA picks the depth, and its proof already exists. Confirm the retimed netlist with sequential equivalence checking. If STA still fails: restructure, retime within the existing stages, add a stage if the budget allows, or return to step 6.

Three points distinguish this from the simpler "write SVA, generate RTL, sweep STA, fix the failures":

- **One STA run is enough for the sweep.** The package computes Fmax and the limiting path at every frequency from the measured delays. A final STA run confirms the chosen frequency, because synthesis optimizes differently under different constraints.
- **The even split can mislead.** Probe the actual cut of any near-critical path.
- **Assertions are not tuned to pass timing.** A budget changes only as a deliberate specification decision; implementation assertions are regenerated from the package, never hand-edited.

In one sentence: classify every requirement and state it in cycles, measure the logic early with throwaway AI-generated RTL, let the package give the frequency and the limiting assertion, fix the limit at the specification or architecture level, and verify every legal pipeline depth so that final STA picks one that is already proven.

8. Worked example

The demonstration in *How Fast Can This Design Run?* used four paths. Classified with the three questions:

SVA property	Function	Q1 loop?	Q2 exact cycle required?	Class and budget
`p_cnt`	32-bit event counter, one event per cycle	yes	—	loop, II = 1
`p_sel`	select one of 16 words, sampled next cycle	no	yes	fixed, 1 cycle
`p_mad`	y = a × b + c, with `out_valid`	no	no	flexible, 1 to 4 cycles
`p_hit`	address-region decode, with valid	no	no	flexible, 1 to 2 cycles

Probe delays were measured with Yosys and ABC on the ASAP7 predictive 7 nm library (typical corner) and scaled to 5 nm-class and 3 nm-class by assumed ratios. Minimum clock period per path, in ps:

Path	7 nm	5 nm-class	3 nm-class
`p_cnt` (loop)	506	445	377
`p_mad` (flexible, 4)	410	362	309
`p_sel` (fixed, 1)	284	253	219
`p_hit` (flexible, 2)	272	242	210
Fmax	1.98 GHz	2.25 GHz	2.65 GHz

The classification explains the result. The multiply-add has almost three times the logic delay of the counter, but it is flexible with four cycles, so pipelining divides its delay. The counter is a loop that must accept an event every cycle; nothing divides its delay. The least flexible path, not the slowest, sets the clock.

The ranking is still an estimate. A measured two-stage cut of the multiply-add had its worst stage 50% longer than an even split predicts; if a four-stage cut were equally unbalanced, the multiply-add would limit the clock in all three technologies. That is why step 5 of the flow probes the real cut.

The decisions that follow are specification decisions. For the counter: restructure it (look-ahead, or a fast low part and a slow high part), or accept events only every other cycle if the system allows, which would give it II = 2. For the select: move its consumer to a valid-based interface, making it flexible. For the multiply-add: probe its real cut before committing to four stages.

9. Design advice

- **Prefer flexible interfaces wherever you control both sides.** Valid/ready or tagged interfaces make pipelining free later. Keep fixed timing only where an external protocol truly demands it. Every fixed `##N` avoided is one less path that can limit the clock.
- **State throughput explicitly for every loop.** An II of 2 instead of 1 halves the loop's logic pressure; it is a system decision worth making consciously.
- **Treat handshake flags as loops.** Their timing is part of the interface contract, not an implementation detail.
- **Write the class into the specification.** A short comment on each assertion, such as "loop, II = 1" or "flexible, B from 8 credits", saves the next engineer from reverse-engineering it.

10. What not to do

- Do not widen a window, raise a budget, or drop an assertion to make STA pass.
- Do not register a handshake flag to fix its timing without changing the specification.
- Do not add a register inside a loop.
- Do not trust even-split estimates for paths near the limit.
- Do not hand-edit a `##N`; change the package and regenerate.
- Do not treat probe numbers as signoff.

11. Conclusion

An assertion's delay operator says what the cycle count is; the requirement behind it says whether that count can ever change. Classifying each requirement as a loop, fixed, or flexible, with three questions, answers that before design starts, and it largely predicts which paths will limit the clock. Combined with throwaway AI-generated RTL, one STA run, and a parameter package, it turns "how fast can this design run?" into a short, traceable calculation, and it points each timing problem to the specification decision that can solve it.

Appendix A. Example properties for each class

class_examples_sva.sv

```
// class_examples_sva.sv : example properties for each requirement class (loop / fixed / flexible)
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// Illustrative checker; signal names are generic. Parsed with slang; not bound to a design.
module class_examples_sva #(parameter int unsigned DEPTH = 8, parameter int B_MAD = 4) (
  input logic        clk, reset_n,
  // loop: event counter, one event per cycle
  input logic        inc,
  input logic [31:0] count,
  // loop: accumulator
  input logic        acc_valid,
  input logic [31:0] x, sum,
  // fixed: SRAM with a 2-cycle read latency on a fixed-timing interface
  input logic        rd_en, rvalid,
  // flexible: multiply-add with out_valid
  input logic        in_valid, out_valid,
  input logic [15:0] a, b,
  input logic [31:0] c,
  input logic [32:0] y,
  // loop through another block: FIFO flags
  input logic        push, pop, full, almost_full,
  input logic [$clog2(DEPTH+1)-1:0] fifo_count
);
  // ---- LOOP: the next value depends on the previous value ($past of its own output) -------
  property p_cnt_next;
    inc |-> ##1 (count == $past(count) + 32'd1);
  endproperty
  ap_cnt_next: assert property (@(posedge clk) disable iff (!reset_n) p_cnt_next);

  property p_acc_next;
    acc_valid |-> ##1 (sum == $past(sum) + $past(x));
  endproperty
  ap_acc_next: assert property (@(posedge clk) disable iff (!reset_n) p_acc_next);

  // A loop whose throughput requirement allows one input every 2 cycles gets a 2-cycle budget
  property p_acc_ii2_env;                  // environment: no back-to-back inputs
    acc_valid |-> ##1 !acc_valid;
  endproperty
  mp_acc_ii2_env: assume property (@(posedge clk) disable iff (!reset_n) p_acc_ii2_env);

  property p_acc_ii2;
    logic [31:0] v_sum;
    (acc_valid, v_sum = sum + x) |-> ##2 (sum == v_sum);
  endproperty
  ap_acc_ii2: assert property (@(posedge clk) disable iff (!reset_n) p_acc_ii2);

  // ---- FIXED: an external protocol samples an exact cycle ---------------------------------
  property p_sram_read_fixed;
    rd_en |-> ##2 rvalid;
  endproperty
  ap_sram_read_fixed: assert property (@(posedge clk) disable iff (!reset_n) p_sram_read_fixed);

  // ---- FLEXIBLE: the consumer waits for out_valid; B comes from the system ----------------
  logic [7:0] in_cnt, out_cnt;             // checker-only transaction counters
  always_ff @(posedge clk or negedge reset_n)
    if (!reset_n) begin in_cnt <= '0; out_cnt <= '0; end
    else begin
      if (in_valid)  in_cnt  <= in_cnt  + 8'd1;
      if (out_valid) out_cnt <= out_cnt + 8'd1;
    end

  property p_mad_contract;
```

```
    logic [7:0]  v_id;
    logic [32:0] v_exp;
    (in_valid, v_id = in_cnt, v_exp = 33'(a) * 33'(b) + 33'(c)) |->
      strong(##[1:B_MAD] (out_valid && out_cnt == v_id && y == v_exp));
  endproperty
  ap_mad_contract: assert property (@(posedge clk) disable iff (!reset_n) p_mad_contract);

  // ---- LOOP THROUGH ANOTHER BLOCK: FIFO full -----------------------------------------
  // The writer reacts to 'full' in the same cycle (environment), so 'full' must be exact:
  property p_writer_obeys_full;
    full |-> !push;
  endproperty
  mp_writer_obeys_full: assume property (@(posedge clk) disable iff (!reset_n) p_writer_obeys_full);


property p_full_exact;
    (fifo_count == DEPTH) |-> full;
  endproperty
  ap_full_exact: assert property (@(posedge clk) disable iff (!reset_n) p_full_exact);

  // To register the flag one cycle later, the SPEC must change: warn one entry earlier
  property p_almost_full_early;
    (fifo_count >= DEPTH - 1) |-> ##1 (almost_full || fifo_count < DEPTH - 1);
  endproperty
  ap_almost_full_early: assert property (@(posedge clk) disable iff (!reset_n) p_almost_full_early);

endmodule : class_examples_sva
```

Appendix B. One-page checklist

```
CLASSIFY (for every SVA path)
  [ ] Q1  Next result depends on previous result?        -> LOOP; write throughput (II) as an
assumption
  [ ] Q2  Anyone outside depends on the exact cycle?     -> FIXED ##N; name the protocol/agent
  [ ] Q3  Consumer waits for valid/ready/tag?            -> FLEXIBLE ##[1:B]; name the source of B
  [ ] Handshake flags (full, ready, busy, grant) checked as loops through the other block
  [ ] Unknown class or unknown B -> RQM0 halt: ask the architect

MEASURE
  [ ] Throwaway probe RTL: one module per path, named after its assertion, complete, exact widths
  [ ] Probe checked (scan, simulation or equivalence) before timing
  [ ] One synthesis + STA run at the slow setup corner; D and t_ovh into the package (PROBE)

DECIDE
  [ ] Package: T_min per path, Fmax, limiting assertion, cycle margins
  [ ] Real pipeline cut probed for every path near the limit
  [ ] Limit addressed at spec/architecture level; no window widened to pass timing

BUILD AND CLOSE
  [ ] RTL depth from the package; contract + generated implementation assertions
  [ ] Every legal depth verified up front
  [ ] Final STA picks the depth; sequential equivalence on the retimed netlist
```

Appendix c: package fmax_lat_pkg

```
// fmax_lat_pkg.sv : cycle-budget latency package -- how fast can the design run, per technology
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//
// Requirements are in CYCLES (budgets B, from the SVA). Time is used only to measure logic.
// Integers only: ps and whole percent.
//   T_min(path) = t_ovh + ceil(D / B)        Fmax period = max over paths of T_min
//   N(path)     = ceil(D / (T_clk - t_ovh))  stages needed at the chosen clock (must be <= B)
// Delay provenance: PROBE, Yosys 0.33 + ABC, ASAP7 RVT TT (7 nm predictive), buffered and sized.
// 5 nm-class and 3 nm-class values are DERIVED from 7 nm by ASSUMED ratios.
// Margins (skew, uncertainty, variation) and the 20% wire allowance are ASSUMED.
//
// Select:  +define+FMAX_TECH_5NM  or  +define+FMAX_TECH_3NM   (default 7 nm)
//          +define+FMAX_CLK_PS=<ps>                          (default: this technology's Fmax period)
`default_nettype none

package fmax_lat_pkg;

  // ---- technology step: 0 = 7 nm (measured), 1 = 5 nm-class, 2 = 3 nm-class ----------------
`ifdef FMAX_TECH_3NM
  localparam int TECH_STEP = 2;
`elsif FMAX_TECH_5NM
  localparam int TECH_STEP = 1;
`else
  localparam int TECH_STEP = 0;
`endif

  // ---- ASSUMED scaling ratios (percent) ----------------------------------------------------
  localparam int CELL_PCT_7_TO_5 = 85;  localparam int NET_PCT_7_TO_5 = 95;
  localparam int CELL_PCT_5_TO_3 = 80;  localparam int NET_PCT_5_TO_3 = 95;
  localparam int WIRE_PCT        = 20;  // net delay added to the pre-layout probe delay (7 nm)
  localparam int MARGIN_PS       = 20 + 30 + 20;   // skew + uncertainty + variation, not scaled

  function automatic int ceil_div(input int x, input int y);
    return (x + y - 1) / y;
  endfunction
  function automatic int scale_cell(input int v7);
    int v = v7;
    if (TECH_STEP >= 1) v = v * CELL_PCT_7_TO_5 / 100;
    if (TECH_STEP >= 2) v = v * CELL_PCT_5_TO_3 / 100;
    return v;
  endfunction
  function automatic int scale_net(input int v7);
    int v = v7;
    if (TECH_STEP >= 1) v = v * NET_PCT_7_TO_5 / 100;
    if (TECH_STEP >= 2) v = v * NET_PCT_5_TO_3 / 100;
    return v;
  endfunction
  function automatic int path_delay(input int probe7);    // cell + net, in this technology
    return scale_cell(probe7) + scale_net(ceil_div(probe7 * WIRE_PCT, 100));
  endfunction
  function automatic int max4(input int a, input int b, input int c, input int d);
    int m = a;
    if (b > m) m = b;
    if (c > m) m = c;
    if (d > m) m = d;
    return m;
  endfunction

  // ---- register overhead -------------------------------------------------------------------
  localparam int CLK2Q_SETUP_7 = 83 + 11;          // ASAP7 DFFHQNx1, mid-table (library)
  localparam int T_OVH_PS      = scale_cell(CLK2Q_SETUP_7) + MARGIN_PS;   // 164 / 149 / 133

  // ---- 7 nm probe delays, ps (PROBE) and cycle budgets from the SVA ------------------------
  localparam int PROBE7_MAD = 819;  localparam int B_MAD = 4;   // p_mad : y = a*b + c, ##[1:4]
  localparam int PROBE7_HIT = 179;  localparam int B_HIT = 2;   // p_hit : region decode, ##[1:2]
```

```
  localparam int PROBE7_CNT = 285;  localparam int B_CNT = 1;   // p_cnt : counter LOOP, ##1
  localparam int PROBE7_SEL = 100;  localparam int B_SEL = 1;   // p_sel : 16:1 select, ##1

  localparam int D_MAD = path_delay(PROBE7_MAD);
  localparam int D_HIT = path_delay(PROBE7_HIT);
  localparam int D_CNT = path_delay(PROBE7_CNT);
  localparam int D_SEL = path_delay(PROBE7_SEL);

  // ---- minimum clock period each path allows, and the design's Fmax period ---------------
  localparam int TMIN_MAD = T_OVH_PS + ceil_div(D_MAD, B_MAD);
  localparam int TMIN_HIT = T_OVH_PS + ceil_div(D_HIT, B_HIT);
  localparam int TMIN_CNT = T_OVH_PS + ceil_div(D_CNT, B_CNT);
  localparam int TMIN_SEL = T_OVH_PS + ceil_div(D_SEL, B_SEL);
  localparam int FMAX_PERIOD_PS = max4(TMIN_MAD, TMIN_HIT, TMIN_CNT, TMIN_SEL);

  // ---- the chosen clock, and what each path needs at that clock ---------------------------
`ifdef FMAX_CLK_PS
  localparam int CLK_PS = `FMAX_CLK_PS;
`else
  localparam int CLK_PS = FMAX_PERIOD_PS;
`endif
  localparam int USABLE_PS = CLK_PS - T_OVH_PS;

  localparam int N_MAD = ceil_div(D_MAD, USABLE_PS);   // pipeline depth for RTL and exact SVA
  localparam int N_HIT = ceil_div(D_HIT, USABLE_PS);
  localparam int N_CNT = ceil_div(D_CNT, USABLE_PS);   // must be 1: a loop cannot be pipelined
  localparam int N_SEL = ceil_div(D_SEL, USABLE_PS);

  localparam int MARGIN_MAD = B_MAD - N_MAD;           // cycle margin: extra stages still allowed
  localparam int MARGIN_HIT = B_HIT - N_HIT;
  localparam int MARGIN_CNT = B_CNT - N_CNT;
  localparam int MARGIN_SEL = B_SEL - N_SEL;

  localparam int SLACK_MAD = CLK_PS - (T_OVH_PS + ceil_div(D_MAD, N_MAD));   // per stage, even split
  localparam int SLACK_HIT = CLK_PS - (T_OVH_PS + ceil_div(D_HIT, N_HIT));
  localparam int SLACK_CNT = CLK_PS - (T_OVH_PS + ceil_div(D_CNT, N_CNT));
  localparam int SLACK_SEL = CLK_PS - (T_OVH_PS + ceil_div(D_SEL, N_SEL));

  localparam bit FEASIBLE = (USABLE_PS > 0) && (MARGIN_MAD >= 0) && (MARGIN_HIT >= 0) &&
                            (MARGIN_CNT >= 0) && (MARGIN_SEL >= 0);

endpackage : fmax_lat_pkg

`default_nettype wire
```

References

1. B. Cohen and K. Chibani, *How Fast Can This Design Run?*, systemverilog.us, 2026.
2. B. Cohen and K. Chibani, *When Timing Closure Breaks the Proof*, systemverilog.us, 2026.
3. B. Cohen and K. Chibani, *RAG-RTL: Rule-Governed AI Generation of Pipelined RTL*, systemverilog.us, 2026.
4. B. Cohen and K. Chibani, Same Design, New Node, systemverilog.us, 2026.
5. B. Cohen and K. Chibani, *Knowledge-Driven AI for SystemVerilog Assertions*, VhdlCohen Publishing, 2026. TOC and sample
6. L. T. Clark et al., "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, 2016.

Further material is available at systemverilog.us.

---

[i] That's two capital letters I, "II", stands for **initiation interval**. Here are the terms in T_min = t_ovh + ceil(D / II):

- **D is the logic delay of the path**, in picoseconds: the time for a signal to pass through the combinational logic from one register to the next, with no pipelining. It comes from STA on the unpipelined probe module.
- **II is the initiation interval**: how many clock cycles pass between one new input and the next. II = 1 means a new input every cycle; II = 2 means every other cycle. It's used for **loop** paths (counter, accumulator, FSM), where the next result needs the previous one. The loop can't be pipelined deeper than the time between inputs, so the loop's logic gets at most II cycles. For a counter that must count every cycle, II = 1.
- **ceil is round up** to the next whole number: ceil(2.1) = 3, ceil(3.0) = 3. D / II is the logic delay per cycle if the logic is split evenly across II cycles. We work in integer picoseconds, so a fraction has to be rounded, and rounding up keeps the result from being optimistic.
- **t_ovh is the per-register overhead**: clock-to-Q, plus setup, plus clock skew, uncertainty, and on-chip variation margins.

Example (illustrative numbers): a loop path with D = 350 ps, II = 1, and t_ovh = 160 ps:
T_min = 160 + ceil(350 / 1) = 510 ps, so the fastest clock for that path is about 1.96 GHz.
With II = 2 (new data every other cycle), T_min = 160 + 175 = 335 ps. The clock can run faster, but the path accepts new data only half as often.
II and B: for non-loop paths the same formula uses the cycle budget B from the latency requirement: t_ovh + ceil(D / B). For loops the budget comes from throughput, so B = II. It's the same formula, with the budget taken from a different requirement.