The `ASSERT_STATIC_LINT_ERROR Macro

A Deliberate Width-Mismatch Trick for Elaboration-Time Checks, and Its Risks

Ben Cohen, Kais Chibani

Cohen Publishing · systemverilog.us

1. Origin and Purpose

`ASSERT_STATIC_LINT_ERROR is a macro in OpenTitan's prim_assert.sv (lowRISC / Google). Its commit message states the intent directly: it is suitable to warn about security parameters taking on non-default values — values that may be required during pre-silicon or FPGA evaluation, but that must never reach tapeout. The problem it solves is that a normal SystemVerilog assert property only fires during simulation. A parameter that nobody happens to simulate in its dangerous state — for example, a debug-backdoor enable left on by mistake — would never trip a simulation assertion at all. `ASSERT_STATIC_LINT_ERROR instead converts the check into something a lint or synthesis tool evaluates at elaboration time, independent of whether any testbench ever simulates the condition.

2. The Macro

The macro expands to:

```
`define ASSERT_STATIC_LINT_ERROR(__name, __prop) \
localparam int __name = (__prop) ? 1 : 2; \
always_comb begin \
  logic unused_assert_static_lint_error; \
  unused_assert_static_lint_error = __name'(1'b1); \
end
```

3. How It Works

- localparam int __name = (__prop) ? 1 : 2; evaluates the condition at elaboration time and stores the result — 1 or 2 — in a parameter named after the macro's __name argument.
- unused_assert_static_lint_error = __name'(1'b1); is the trick. __name'(...) is a size cast, using the parameter __name itself as the cast width. If __prop was true, __name is 1, so this casts 1'b1 to a 1-bit value — completely unremarkable, since the target signal is declared as plain logic (1 bit).
- If __prop was false, __name is 2, so the same literal is cast to a 2-bit value and assigned into a 1-bit signal — a genuine width mismatch, of exactly the kind lint and synthesis tools are built to catch.

In short: a passing condition produces ordinary, unremarkable code. A failing condition produces a real width-mismatch bug, manufactured on purpose, sitting in the RTL for lint or synthesis to find.

4. Worked Example

Consider a module parameter that must never be left enabling a debug backdoor in a production build:

```
module secure_block #(
  parameter bit EnableDebugBackdoor = 0
) (
  ...
);
  `ASSERT_STATIC_LINT_ERROR(NoDebugBackdoor, EnableDebugBackdoor == 0)
endmodule
```

4.1 Safe Case: EnableDebugBackdoor = 0

__prop evaluates true, so the macro expands to:

```
localparam int NoDebugBackdoor = 1;
always_comb begin
  logic unused_assert_static_lint_error;
  unused_assert_static_lint_error = 1'(1'b1);   // 1-bit into 1-bit: fine
end
```

Widths match. No lint tool has anything to report. Elaboration proceeds silently.

4.2 Dangerous Case: EnableDebugBackdoor = 1

__prop evaluates false, so the macro expands to:

```
localparam int NoDebugBackdoor = 2;
always_comb begin
  logic unused_assert_static_lint_error;
  unused_assert_static_lint_error = 2'(1'b1);   // 2-bit into 1-bit: mismatch
end
```

A 2-bit value is now being assigned into a 1-bit variable — an honest width-mismatch bug that a lint or synthesis tool is expected to flag, converting "did this parameter check pass?" into "does this file have a lint violation?", catchable during sign-off regardless of simulation coverage.

5. Danger and Risk

This mechanism is clever, but it is a hack built on the side effects of a cast, not a first-class check. Several risks follow directly from that:

- Not a simulation check. Nothing here is clocked or evaluated at runtime; a regression suite that only runs simulation, without a lint pass, will never see this check at all — there is no equivalent uvm_error or $error to catch in a simulation log.
- Entirely dependent on lint-tool configuration. The check only has an effect if the lint or synthesis tool is configured to treat that specific class of width mismatch as an error rather than an informational message or a suppressed-by-default warning. A tool with looser default settings may implicitly truncate the value and say nothing.
- Silent defeat by waivers. OpenTitan's own lint infrastructure includes per-file, per-rule waiver files (for example, prim.waiver, AscentLint waiver files, Verilator .vlt configuration) that suppress named width-mismatch rules at specific files or locations, with justifying comments such as "Bitwidths are fine here." A waiver written to silence a legitimate, unrelated width warning in the same file — if scoped broadly by file or rule rather than narrowly by signal or line — can inadvertently suppress this deliberately manufactured error as well, with no distinguishing signal between the two in the waiver file itself.
- A pass and a skipped check look identical. Because the safe case produces no output at all, silence from the lint tool cannot be distinguished from "the check passed" versus "the check was never run" (wrong tool invoked, lint step skipped in a fast local flow, rule not enabled). There is no positive confirmation that the constraint was ever evaluated.
- Poor traceability in lint reports. When the check does fire, the lint tool reports a generic width-mismatch complaint against an internal signal literally named unused_assert_static_lint_error — not against the human-readable intent ("NoDebugBackdoor") or the parameter that caused it. Someone triaging a long lint report has to already know to connect that specific warning to the security check; it does not self-explain the way a named simulation assertion failure does.

- Tool portability is not guaranteed. The trick relies on a specific cast-based width-mismatch detection behavior. Not all lint or synthesis tools implement identical semantics for a sized cast assigned to a differently-sized signal, so the same macro instance may be caught by one tool's default rule set and missed entirely by another's.
- Sign-off scope matters. If the specific rule name this pattern trips (which varies by tool — comparable width-mismatch rules appear in OpenTitan's waiver files under names such as ARITH_BITLEN and ONE_BIT_MEM_WIDTH) is not part of a project's mandatory, blocking lint rule set at sign-off, the violation can be downgraded to informational and passed over.

6. Recommendation

Given these risks, `ASSERT_STATIC_LINT_ERROR should not be treated as a self-sufficient guarantee for a security-critical parameter. Practical mitigations:

- Audit lint waiver files whenever this macro is added, to confirm no existing file- or rule-scoped waiver could blanket-suppress the width mismatch it depends on.
- Confirm, per lint tool in the sign-off flow, that the specific width-mismatch rule this pattern trips is enabled and configured as blocking, not informational.
- Where the design is also simulated, pair this elaboration-time check with a genuine simulation-time construct (for example, `ASSERT_KNOWN or an ordinary assert on the parameter) so the constraint is caught by two independent mechanisms rather than one.
- For security-relevant instances specifically, consider grouping them under a distinguishing name prefix (in the spirit of OpenTitan's own FpvSecCm convention for security-critical assertions) so they can be located and confirmed present as a batch, rather than relying on lint-report triage alone to notice one is missing.

References

lowRISC / OpenTitan. prim_assert.sv, github.com/lowRISC/opentitan/blob/master/hw/ip/prim/rtl/prim_assert.sv.

Vogel, P. “[prim] Add new assertion macro for generating static lint errors,” lowRISC/opentitan commit, referencing lowRISC/OpenTitan#10721.

lowRISC / OpenTitan. Lint waiver files (prim.waiver, rv_timer.waiver, and associated .vlt configurations), github.com/lowRISC/opentitan, hw/ip/*/lint/.