Agentic AI Coding Meets RTL Static Analysis:

Using Blue Pearl Visual Verification Suite to Raise the Quality of AI-Generated RTL

Adam Taylor | 15 March 2026 | *FPGA Design, AI Tools, Verification*

Introduction

Those of you who follow my work will know I have been looking seriously at AI coding tools for RTL development for a while now. The question I keep coming back to is not whether AI can generate RTL (it clearly can) but whether that RTL meets the quality bar you need for a production FPGA design. Syntactically correct and functionally plausible is not the same as production-ready, and that distinction matters enormously.

To explore this properly I set up a structured experiment. I asked Claude, Anthropic's AI, to generate a VHDL I2C Master Controller. A real peripheral with features you would actually use: Standard and Fast Mode (100 KHz and 400 KHz), repeated start, 7-bit addressing, configurable system clock frequency. The kind of block an experienced RTL engineer would write over an afternoon.

Claude produced the code. I then took that output straight into Blue Pearl Visual Verification Suite (BPS VVS) to see what it looked like under the microscope. BPS VVS is a static analysis tool: it inspects RTL structurally, checking it against an extensive library of rule sets covering correctness, methodology, and coding standards, without simulation or mathematical proof. That is a different discipline from formal verification, which involves model checking and property proving. Static analysis is faster, requires no testbench, and catches a broad class of structural and methodology issues that neither synthesis nor simulation will surface. The results were instructive, but what I found equally valuable was the process of interpreting the BPS output with engineering judgement, understanding which findings demand action, which point to system-level solutions, and which are simply less meaningful for an FPGA target.

> *The question is not whether AI can write RTL. It clearly can. The question is whether the RTL it writes is production-quality, and that requires structured verification, not just synthesis.*

What Claude Generated

The brief was straightforward. I asked Claude to generate an I2C Master Controller in VHDL with the following requirements:

- 100 KHz Standard Mode and 400 KHz Fast Mode operation
- Repeated start (restart) conditions
- 7-bit slave addressing
- Configurable system clock frequency via generics
- Active-low open-drain outputs suitable for direct connection to an I2C bus

Claude produced a well-structured entity, `i2c_master`, with sensible generics (`G_SYS_CLK_FREQ` and `G_I2C_CLK_FREQ`) and a clean state machine covering IDLE, START, ADDR, ADDR_ACK, WRITE, WRITE_ACK, READ, READ_ACK, RESTART, and STOP states. The open-drain interface was correctly implemented with tri-state assignments. At first glance this is solid code. It compiles, the architecture is logical, and an engineer reviewing it cold would consider it a reasonable starting point.

However, 'reasonable starting point' is not the same as 'ready to ship'. Running it through BPS VVS quickly surfaced a set of issues that fall into distinct categories, each requiring a different engineering response.

BPS VVS Check Configuration

Before looking at the findings, it is worth being explicit about what checks were actually enabled. BPS VVS covers an extensive range of check categories, and the configuration you run with determines what you find. For this analysis the following check packages were enabled:

Check Package	Status	Coverage Focus
Bad Logic	**Enabled**	Logic constructs likely to cause incorrect behaviour
Basic Checks	**Enabled**	Fundamental RTL correctness
Design initializes/resets	**Enabled**	Reset and initialisation methodology
No Races	**Enabled**	Race conditions and non-deterministic behaviour
No redundant logic	**Enabled**	Optimisation and synthesis efficiency
No drive conflicts	**Enabled**	Multiple drivers, tri-state issues
No X-source problems	**Enabled**	Undefined value propagation
Coding style	**Enabled**	RTL style and readability conventions
Coding conventions	**Enabled**	Structural and naming conventions
Assignment Checks	**Enabled**	Signal assignment correctness
Simulation/Synthesis	**Enabled**	Sim vs synthesis mismatches
No implied latches	**Enabled**	Unintentional latch inference
Case statements	**Enabled**	Case statement completeness and style
No size conflicts	**Enabled**	Vector width mismatches
Signal Identification	**Enabled**	Clock, reset, and signal classification
FSM Checks	**Enabled**	State machine structure and completeness
Miscellaneous checks	**Enabled**	General catch-all methodology checks
Cycle Based Simulation	**Enabled**	Cycle-accurate simulation considerations
Input IP	**Enabled**	IP reuse and integration checks
New RTL / Golden RTL	**Enabled**	RTL quality methodology
DFT	**Enabled**	Design for test considerations

Check Package	Status	Coverage Focus
Principles of Verifiable RTL	**Enabled**	Verifiability guidelines
Reuse Methodology Manual	**Enabled**	ARM/Synopsys RMM compliance
Semiconductor Reuse Standard	**Enabled**	Industry reuse standard compliance
DO-254	**Enabled**	Airborne electronic hardware guideline
STARC	**Enabled**	Semiconductor Technology Academic Research Centre guidelines
UltraFast Design Methodology for Vivado	**Enabled**	Xilinx FPGA-specific best practices
Quartus II Best Practices	**Enabled**	Intel/Altera FPGA-specific best practices
Microsemi RTG4 Best Practices	**Enabled**	Microsemi radiation-tolerant FPGA guidelines
Low Power	**Not enabled**	Power optimisation checks
Naming	**Not enabled**	Signal and port naming conventions
Comments	**Not enabled**	Documentation and comment coverage
VHDL Only	**Not enabled**	VHDL-specific language checks

The breadth of that enabled check set is significant. We are not just running a quick lint pass; we are checking against established industry methodologies including Reuse Methodology Manual, STARC, DO-254, and both major FPGA vendor best practice guides. This is a comprehensive quality assessment, and it means the 69 findings in the report carry real weight.

What BPS VVS Found and What To Do About It

The report returned 69 individual messages. Rather than list them all, I want to focus on the key categories and, critically, what the right engineering response is to each. Not all findings are equal, and part of working effectively with a verification tool is knowing how to interpret and prioritise its output.

1. Asynchronous Reset Without Synchronous De-assertion

The most pervasive finding: every register in the design uses an asynchronous reset (`i_rst_n`) with no synchronous de-assertion. BPS flagged this across 11 registers with W-BPS-0453.

BPS is correct to flag this. Asynchronous assertion is fine; you want reset to take effect immediately regardless of the clock. But asynchronous de-assertion is a metastability risk. If the reset release is not synchronised to the clock domain, different flip-flops can come out of reset on different clock edges, leaving the state machine in an undefined intermediate state. On an FPGA design like an I2C master, where the reset state matters, this is a real concern.

The correct response here is not to modify the I2C master block at all. The right solution is a reset synchroniser at the system level, external to the IP. A dedicated reset controller or synchroniser stage at the top level handles reset release cleanly for all blocks in the design, and the I2C master simply receives a pre-synchronised reset signal. The block itself is correct as written, given that assumption.

Which means the correct action in BPS VVS is to raise a waiver against these findings, with a documented rationale: the synchronous de-assertion requirement is met at the system level by the reset controller; this IP block is designed to receive a pre-synchronised reset and the findings are dispositioned accordingly. The waiver is not bypassing the check. It is formal evidence that the finding was reviewed by someone with the knowledge to understand it and disposition it correctly.

> *Raising a waiver with a documented rationale is not the same as ignoring a warning. It is the professional response: evidence that the finding was understood, that the system-level architecture addresses it, and that a qualified engineer has taken responsibility for that decision.*

This is where the finding becomes a clear demonstration of why experienced FPGA engineers remain essential in an AI-assisted design flow.

An inexperienced engineer handed this BPS report would likely do one of two things: ask the AI to fix it directly, or not know how to disposition it at all. If they ask the AI to fix it, the AI will add a synchroniser inside the block, the warning disappears, and the design is subtly worse. The synchroniser now lives inside the IP, coupled to assumptions about clock domain topology that the block has no business making. The check is cleared; the engineering decision was wrong.

An experienced FPGA engineer reads the same finding differently. They understand that reset architecture is a system-level concern. They know that IP blocks should receive clean, pre-synchronised resets from a controller that has visibility of the full clock domain picture. They know how to write a technically defensible waiver that documents the rationale and records the decision. That is a skill that requires genuine system-level FPGA design knowledge, the kind that comes from years of building real designs and understanding why the methodology conventions exist.

No AI tool reliably carries that depth of contextual knowledge. BPS surfaces the finding correctly. The experienced engineer interprets it correctly, dispositions it correctly, and leaves a documented audit trail. That is the combination that produces a trustworthy design.

2. Unregistered Input Ports

Every control input: `i_start`, `i_stop`, `i_restart`, `i_read`, `i_ack_type`, `i_slave_addr`, `i_data_wr` was flagged as unregistered (W-BPS-0655).

This is a legitimate finding with a legitimate fix inside the block. Input registration at the entity boundary provides metastability protection for signals crossing into the I2C master's clock domain, and gives the timing closure tools a clean register-to-register path at the input. For a production peripheral, you want this.

This is a straightforward remediation: add a registered pipeline stage at the inputs. Unlike the reset synchroniser, this change lives naturally inside the I2C master and does not create unwanted coupling to the system architecture. This is the kind of finding you do want to feed back to Claude for a second-iteration fix.

3. Fan-out: Knowing When to Push Back

BPS reported significant fan-out on several signals: `state` at 231, `clk_tick` at 168, `ack_received` at 64, `clk_phase` at 102. These generated warnings under W-BPS-0662.

Here is where engineering judgement matters, and where I would push back on taking the BPS output at face value. On an FPGA, fan-out is primarily a placer and router concern, not an RTL concern. Both Vivado and Quartus have sophisticated automatic replication and dedicated high fan-out routing resources. A state register with 231 fan-out in a block of this size will be handled competently by the toolchain; the tools are specifically designed for exactly this.

Blanket fan-out warnings without awareness of the target technology are less actionable on FPGA than they would be for ASIC. If this were a gate-level netlist destined for a custom process, the fan-out numbers would demand attention. For FPGA RTL, the right response is to note them, check the post-implementation timing report, and act only if the tools genuinely cannot close timing on those nets.

> *Fan-out warnings in BPS are meaningful context, not mandatory action items for FPGA targets. Let Vivado or Quartus tell you if it is actually a timing problem before restructuring your RTL around it.*

That said, the architectural reason for the high fan-out on the state signal is worth understanding. A monolithic single-process state machine will naturally produce a high-fan-out state register because every case branch references it. A two-process (or Moore/Mealy split) architecture can reduce this. But that is an architectural refactoring decision, not a response to a fan-out number in a lint report.

4. The Dangling scl_in Signal: A Real Functional Gap

This one is subtle and easy to miss in review. BPS flagged `scl_in` as driven but never used (W-BPS-0809). Claude correctly connected `scl_in <= io_scl` but never reads it anywhere in the state machine.

This is not a style issue or a methodology preference; it is a functional gap. The I2C specification requires that a master can detect when a slave is holding SCL low for clock stretching. Without reading back SCL, the master will not detect this condition and will continue driving the bus, potentially corrupting a transaction with a slave that uses clock stretching.

The dangling signal is the tell. BPS found it; a synthesis tool would silently optimise it away. This is exactly the category of finding where static analysis earns its keep: a functional incompleteness that no amount of directed simulation would find unless you specifically tested against a clock-stretching slave.

5. If-Then-Else Depth

Two locations in the state machine have ITE nesting 3 to 4 levels deep (W-BPS-0250). BPS recommends case statements, and this is worth understanding properly rather than treating as a simple style preference.

The difference between if-then-else and case is not cosmetic. It reflects two fundamentally different synthesis models: priority encoding versus parallel selection.

When you write a chain of if-then-else conditions, the synthesiser implements them as a priority chain. The first condition is evaluated first; if it is false, the second is evaluated; if that is false, the third, and so on. Each level adds logic in series. The result is a ripple of comparators and

multiplexers in cascade, where the critical path grows with every level of nesting. At 3 or 4 levels deep, you are adding meaningful combinatorial delay on the path through that logic, which directly affects timing closure.

A case statement, by contrast, tells the synthesiser that the conditions are mutually exclusive and can be evaluated in parallel. The synthesiser implements a one-hot or binary decoded multiplexer tree rather than a priority chain. The conditions are checked simultaneously, the result is selected, and the critical path is the depth of a single comparison plus the mux, regardless of how many cases there are. For something like a state machine with 10 states, that is a significant structural difference.

There is a place for if-then-else: when conditions genuinely have priority and you need the first true condition to take precedence. Interrupt masking, fault override logic, and similar constructs are natural fits. But a state machine where only one state can be active at a time has no inherent priority between its branches. Every state is mutually exclusive by definition. Using if-then-else to decode it is applying a priority structure to something that does not need one, and paying a timing penalty for it.

The AI generated if-then-else nesting because it models what it has seen, and deeply nested conditionals appear throughout training data. It did not reason about synthesis implications. BPS caught it. The fix is clean and localised: convert the nested ITE structures to case statements and the synthesiser gets the parallel selection it needs to build efficient logic.

Summary of Findings and Recommended Actions

Check	Action	Rationale
`NO_SYNCH_DEASSERT_RST`	Waiver + system level	BPS is correct to flag; fix belongs in the reset controller at top level. Raise a waiver with documented rationale in BPS VVS
`REGI (inputs)`	Fix inside block	Input registration is a legitimate boundary concern; straightforward remediation
`FANOUT`	Monitor only	FPGA toolchains handle high fan-out; act only if post-implementation timing fails
`UNCONNECTED scl_in`	Fix inside block	Real functional gap: clock stretching support is missing and must be addressed
`ITE_DEPTH`	Fix inside block	Clean conversion to case statements improves readability and synthesis quality

The Workflow: AI Generation + Static Analysis

So where does this leave us? I want to be direct: the generated RTL is a genuinely good starting point. The state machine is architecturally sound, the I2C protocol logic is correctly implemented for the common case, and the code is clean and well-structured. The AI did not produce garbage; it produced competent RTL that needed a verification pass to identify the gaps between working on the bench and production quality.

The workflow I am advocating looks like this:

- Use AI to generate the initial RTL: fast, architecturally competent, syntactically correct
- Configure BPS VVS with the appropriate check packages for your target and methodology
- Run BPS immediately, before simulation, before synthesis, and before peer review
- Interpret the BPS output with engineering judgement and categorise findings by action type
- Feed actionable findings back to the AI for a second-iteration fix
- Raise waivers for system-level findings (reset synchronisation) with documented rationale; do not modify the block
- Re-run BPS VVS to verify closure; the loop is complete when the report is clean

The interpretation step is the one I most want to emphasise. BPS gives you 69 findings. Treated naively, you might ask Claude to fix all 69. Treated with engineering judgement, you get three categories: fix in the block, fix at the system level, and monitor. That categorisation is what turns a verification report into a useful engineering action plan.

> *The value of BPS VVS in an AI coding workflow is not just finding issues; it is providing a structured, deterministic quality specification that you can use to direct AI remediation intelligently.*

Why BPS VVS Fits This Workflow

Several things make BPS VVS particularly well-suited to an agentic AI coding workflow.

Speed. Running the I2C master through BPS took seconds. For an iterative AI workflow where you might generate, verify, remediate, and re-verify multiple times in a single session, that turnaround is essential.

Depth. The 29 enabled check packages cover an enormous range of quality dimensions, from basic logic correctness through FSM completeness, reset methodology, simulation/synthesis consistency, and FPGA-vendor-specific best practices. This is not a lint pass; it is a multi-dimensional quality assessment.

Structure. The CSV output, covering check names, message IDs, line numbers, and signal names, is directly actionable and directly machine-readable. You can sort by category, prioritise by engineering impact, and track closure systematically. That structure is exactly what you need when using BPS findings to direct an AI remediation loop.

Industry methodology alignment. When BPS flags something against Reuse Methodology Manual, STARC, DO-254, or UltraFast Design Methodology for Vivado, it is not expressing a style preference. It is pointing to a documented industry practice with a rationale behind it. That provenance matters; it means the findings have engineering authority, not just tool authority.

Conclusions

AI-generated RTL is here and it is useful. But 'useful' and 'production-ready' are still not the same thing, and the gap between them is where static analysis earns its keep.

The I2C master example makes the point clearly. Claude generated structurally sound, syntactically correct VHDL with a coherent state machine and correctly implemented I2C protocol logic. BPS VVS, running 29 check packages covering everything from basic logic

correctness to FPGA-vendor best practices, found 69 issues. Some demand immediate action inside the block. Some are system-level architectural concerns. Some are effectively noise for an FPGA target and need monitoring rather than fixing.

Knowing the difference is engineering judgement. Having a tool that surfaces all of them systematically and quickly is what makes the agentic AI coding workflow viable at production quality.

The async reset finding makes this concrete. BPS correctly flags a reset methodology issue. An inexperienced engineer, or an AI acting without oversight, either fixes it in the wrong place or does not know how to disposition it. An experienced FPGA engineer knows that the fix belongs in the system-level reset controller, raises a waiver in BPS with a documented rationale, and leaves the block correct and reusable. That distinction is invisible to the tool and invisible to the AI. It requires genuine system-level FPGA design knowledge to get right.

The right mental model is this: AI accelerates RTL generation. Static analysis ensures quality. Experienced engineering judgement interprets the results and makes the right architectural decisions. All three together are significantly more powerful than any one alone. Remove the experienced engineer from that chain and you get faster, cleaner-looking RTL that may still be wrong in ways that matter. That is the workflow worth building, and the skill set worth investing in.

Adam Taylor *is an FPGA engineer and blogger with extensive experience in embedded systems, RTL design, and FPGA-based product development. He writes regularly on FPGA design, AI tools, and engineering methodology.*