Assertions for interface rules

This page answers: *the scoreboard says a beat came out wrong. How do you get a check that names the rule the design broke, and the cycle it broke it in?*

You'll need the answer as soon as you start Lab 05. Your Lab 04 testbench already catches two of that lab's three bugs, B10 and B12, but on B10 the first thing it prints is `[OUT] output 0: got 04 in cycle 56, expected 03`. The scoreboard sees a beat only when a sink takes it, so that line is about the first wrong beat that output 0's sink took. While a beat waits on a stalled output, the scoreboard has no way to notice it going wrong. Part 1 of the lab has you work out how many cycles before that line the router broke its rule. A check that watches the pins themselves fires in the cycle in which the rule is broken, and its message can say which rule it was. A check of that kind is a SystemVerilog **concurrent assertion**. In Lab 05, Part 2 you'll write three of them for the router's outputs, in a checker that the lab's package already binds into the router.

So that this page doesn't write those three for you, its examples use a smaller design of our own, a one-beat valid/ready stage. The stage takes a beat on its input and holds it on its output until the sink takes it. Its source, its checker and the test that drives it are in `guide-examples/assertions-for-interface-rules/`, and every output quoted below comes from a run of it in the course image.

Three rules

The stage has three rules, written in the same style as the router specification's Valid/ready rules:

- **S1**. Once `out_valid` is 1, it stays 1 until the handshake.
- **S2**. While `out_valid` is 1 and `out_ready` is 0, `out_data` doesn't change.
- **S3**. While the stage is empty (`out_valid` is 0), `in_ready` is 1.

S1 and S2 are the router's O1 and O2, written for a design with one output and no `out_last`. We planted two bugs in the stage, and `+BUG=` switches one of them on at run time. With `drop`, the stage throws away a beat that has waited four cycles, which breaks S1. With `bubble`, it holds `in_ready` at 0 for a cycle after each beat leaves, which breaks S3 whenever the stage is empty in that cycle.

A rule as a property

Here is S1 as the checker writes it:

```
a_S1: assert property (@(posedge clk) disable iff (!rst_n)
                       out_valid && !out_ready |=> out_valid)
  else begin n_S1++; $error("S1: out_valid fell without a handshake"); end
```

Read it from the left. `@(posedge clk)` says the property is evaluated at every rising edge, and the values it uses are the ones the signals had just before that edge. That's the same value that a clocking block's `#1step` input gives you, and it's what the specification means by the value of a signal "in a cycle", so the property sees each cycle the way your monitor does.

The expression left of `|=>`, `out_valid && !out_ready`, is the **antecedent**. It picks the cycles that the rule talks about, here the cycles in which a beat is on offer and isn't taken. `|=>` means "then, in the next cycle". The expression after it is the **consequent**, which must hold in that next cycle. In a cycle where the antecedent is false, the property checks nothing and passes. The action block after `else` runs only on a failure. Ours counts the failure and calls `$error` with the rule's name, so the message tells you which rule broke without your having to look up a line number.

S2 has the same antecedent and a different consequent, `$stable(out_data)`, which is 1 when `out_data` has the same value as it had in the previous cycle. S3 is about a single cycle, so it uses the other implication:

```
a_S3: assert property (@(posedge clk) disable iff (!rst_n)
                       !out_valid |-> in_ready)
```

`|->` checks the consequent in the same cycle as the antecedent. A rule of the form "whenever this is true, that is true too" uses `|->`; a rule about what happens next uses `|=>`. Lab 05's O6 is of the first kind, because it's about the beat in the cycle of its own handshake.

`disable iff (!rst_n)` switches the property off while `rst_n` is 0, and abandons any attempt that was waiting for its next cycle when the reset came. The router specification explains why every rule about normal operation needs it. A reset ends whatever was in progress, and the router lowers `out_valid` at once, with no handshake. O1 allows that, but without `disable iff` the assertion for O1 (or for S1) would count it as a failure. The rules about what holds *during* a reset are checked without `disable iff`, and you'll write some of them in Lab 07.

Binding the checker into the design

The assertions are in a module of their own, `stage_sva`, whose inputs are the stage's pins. A `bind` statement at the end of the same file, `stage_sva.sv`, puts a copy of that module inside the stage:

```
bind stage stage_sva u_sva (
  .clk(clk), .rst_n(rst_n),
  .in_valid(in_valid), .in_ready(in_ready),
  .out_valid(out_valid), .out_ready(out_ready), .out_data(out_data)
);
```

`bind` instantiates `stage_sva` as `u_sva` in every instance of `stage`. The names on the right of each connection are signals inside `stage`, so the checker sees the design's own ports, and the stage's RTL doesn't change. The top module reads the failure count back out through the design instance, as `dut.u_sva.n_fail`. Lab 05 does the same for the router, with the `bind` in a file of its own, `tb/router_sva_bind.sv`, and Part 2 asks you why the testbench attaches its checker with `bind` instead of instantiating it in the top module.

Failure messages on Verilator

The checker is plain SystemVerilog with no UVM in it, so the simulator reports its failures itself, and they don't look like the `UVM_ERROR` lines you're used to. Here is the `drop` bug with Verilator's default error limit. The command is `make run BUG=drop PLUSARGS=`, in which the empty `PLUSARGS=` replaces the `+verilator+error+limit+1000` that our Makefile passes:

```
UVM_INFO @ 0: reporter [RNTST] Running test stage_test...
[245000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
%Error: guide-examples/assertions-for-interface-rules/stage_sva.sv:37: Verilog $stop
Aborting...
```

The number in brackets is the simulation time in picoseconds, so the failure was at 245 ns. Our clock has a 10 ns period with its first rising edge at 5 ns, so the edge that ends cycle n is at $10n - 5$ ns. The failure was therefore found at the edge that ends cycle 25. A `|=>` failure is reported in the cycle of the consequent, so the beat was still waiting in cycle 24, and `out_valid` was 0 in cycle 25. The waveform of the same run shows both cycles:

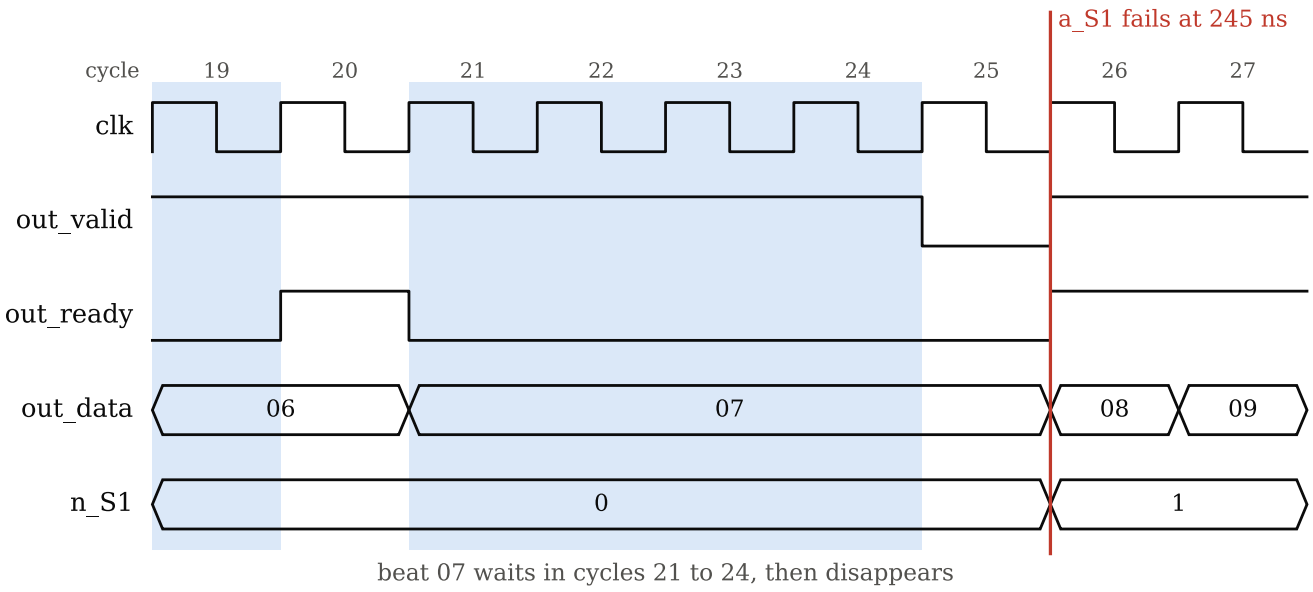


Figure 1: The stage with the drop bug, cycles 19 to 27

The shaded cycles are the ones in which a beat is on offer and isn't taken, so the antecedent of S1 holds in them. Beat `07` waits in cycles 21 to 24, and in cycle 25 `out_valid` is 0 although the sink never took `07`. At the edge that ends cycle 25 the assertion fails and `n_S1` counts it. (The stage takes `08` in cycle 25, so the output is busy again from cycle 26.)

The message names the assertion by its full path, `stage_tb_top.dut.u_sva.a_S1`, so it also tells you which instance of the checker fired. In Lab 05 the path tells you the output as well, because the router's checker has one instance of `router_sva_out` for each output, `u_out0` and `u_out1`.

The run stopped at that failure. Verilator's default error limit is 1, so the first `$error` ends the simulation with exit status 1, before the test's report phase and before UVM's summary. The course's grader reads the log as well as the exit status. It found the assertion's `%Error` line in this run's log and called the run `caught: 1 assertion`.

The run-time option `+verilator+error+limit+<value>` raises the limit (`verilator --help` lists it as "Set error limit"). Every lab from Lab 05 on passes `+verilator+error+limit+100000`, and our example's Makefile passes 1000. With that limit, `make run BUG=drop` prints every failure of the same bug:

```
[245000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
-Info: guide-examples/assertions-for-interface-rules/stage_sva.sv:37: Verilog $stop, ignored
due to +verilator+error+limit
[375000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
[605000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
[1165000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
UVM_ERROR guide-examples/assertions-for-interface-rules/stage_tb.sv(79) @ 2025000:
uvm_test_top [SVA] the bound assertions counted 4 failures
```

Verilator prints the `-Info` line once, after the first failure. A raised limit still stops the run at the error that reaches it, again before UVM's summary, so the labs use a number that no run will reach.

Our test prints the `UVM_ERROR` on the last line itself. In its check phase it reads the checker's `n_fail` and reports an `[SVA]` error if the count isn't 0. Without that check the test would pass, because `dd105_base_test` passes a test when UVM's report server has counted no `UVM_ERROR` and no `UVM_FATAL`, and the server counts only UVM's own reports, not an assertion's `$error`. With `+IGNORE_SVA`, which skips that check, the same run printed the same four `%Error` lines and then `DD105: TEST PASSED`, and its exit status was 0. So any testbench that relies on assertions needs something that turns their failures into a `UVM_ERROR` before the test ends. In the labs, the top module passes `dut.u_sva.n_fail` to the scoreboard through an interface, `router_sva_if`, and the scoreboard reports an `[SVA]` error at the end of the test if the count isn't 0.

When the antecedent never holds

Here is the same buggy stage again, with the sink ready in every cycle (`+READY_PCT=100`):

```
UVM_INFO    guide-examples/assertions-for-interface-rules/stage_tb.sv(85)    @    2025000:
uvm_test_top [STAGE] 132 beats in, 131 out; a beat waited in 0 cycles (cover property: 0); 0
assertion failures
```

and the grader's verdict on it was `MISSED`. The bug is still in the stage, but it acts only on a beat that has waited four cycles, and with the sink always ready no beat waited at all. The antecedent of S1 was false in every cycle, so the assertion passed in every cycle without checking anything. A passing assertion tells you only that its rule held in the cycles where its antecedent was true, and not whether there were any such cycles. Lab 05's B11 hides from your O6 assertion for a related reason. O6's antecedent holds on the last beat of every 8-beat packet, but B11 acts only on a packet longer than 8 beats, so O6 can fail only on the 8th beat of a packet that has a 9th. None of Lab 04's tests sends such a packet, and your virtual sequence in Part 4 is the first stimulus that does.

The checker therefore also counts the cycles in which the antecedent of S1 and S2 holds, in two ways. `n_tried` is a counter in an `always` block beside the assertions, and `c_S1` is a cover property that counts the same cycles:

```
c_S1: cover property (@(posedge clk) disable iff (!rst_n) out_valid && !out_ready)
  n_covered++;
```

On the golden stage, with the sink ready half the time, both counted 90. In the `drop` run above, with the sink always ready, both counted 0. Verilator also writes the cover property's count into `coverage.dat`, as a `user` point named `stage_tb_top.dut.u_sva.c_S1` with the count 90, and `verilator_coverage` reports that point as `user : 100.0% (1/1)`. The course's `make cov` lists only covergroup bins, though, so in the labs the evidence that a rule was exercised comes from covergroups. Lab 05's `cg_wait` is one of them, and it records, for each packet an output delivers, the longest that any of its beats waited.

Which rules suit an assertion

A rule about one or two cycles at a time becomes a short property like S1. I1, I2, O1 and O2 are rules of that kind, and the specification says that most of its other rules are too. Assertions for them name the rule and fire in the cycle in which it's broken. They need no model of the design, so they keep working in any testbench that binds the file. The specification's Valid/ready rules section lists the rules that need more than a short property:

- O5, O6 and O8 need state kept for each output (the order of the beats, how many beats the current packet has had, and the cycles in which each beat went in and was taken). They belong in the scoreboard, or in an assertion with tracking code beside it. Your O6 in Lab 05 uses a small counter, `beats`, which counts the beats of the current packet that have already been taken. It's written for you in `router_sva_out`.
- I4, O3 and O7 say what a signal does *not* depend on, which no single trace can show. A model that predicts the outputs checks them, and an assertion can catch only their consequences.
- I3 is a rule for your own source's stimulus, so it belongs in the driver or the sequence, not in an assertion.

The scoreboard knows what every beat should be, so it catches far more kinds of bug than the assertions do, but its message says only that a beat was wrong. An assertion names the rule and the cycle, which tells you where and when to look, so a testbench needs both kinds of check. In Lab 05 you'll see both fire on the same bugs, often many cycles apart.

Check yourself

1. The `drop` run's first failure printed `[245000]`. In which cycle was `out_valid` 0 without a handshake, and why is the failure reported at the edge that ends that cycle rather than one cycle earlier?

   Answer: A1, in Answers at the end.

2. A test of yours prints three `%Error: ... Assertion failed` lines and then `DD105: TEST PASSED`, with exit status 0. What is it missing?

   Answer: A2, in Answers at the end.

3. With the sink always ready, the `drop` bug passed. Which number in that run's `[STAGE]` line tells you that S1 was never tried?

   Answer: A3, in Answers at the end.

4. Which implication would you use for the router's O6, "a beat taken when the packet has already had 7 beats has `out_last` = 1", and why?

   Answer: A4, in Answers at the end.

Further reading

ChipVerify's assertion pages are a good reference for the syntax. Their example output comes from commercial simulators, so the failure messages look different from Verilator's.

- Property Implication: `|->` against `|=>`, and vacuous success, with the same trace read both ways.
- disable iff: where it goes, and why gating the antecedent instead isn't the same thing.
- $rose, $fell, $stable.
- Handshake Properties, in the formal section: the valid/ready rules, which side's rules are assertions and which are assumptions, and why a sink that is always ready hides a sender bug.
- assert, assume & cover.
- SystemVerilog Assertions, the overview. Correction: it writes an immediate assertion as `$assert(...)`, which isn't SystemVerilog (the keyword is `assert`), and its practice solution asserts a bare sequence with no implication, so it fails in every cycle where `req` is 0.
- Assertion Based Verification. Correction: it says `assume` has no effect in simulation; the standard says a simulator checks an assumed property the same way as an assertion.
- SystemVerilog Assertions in a UVM Testbench: which checks belong in assertions, which in the monitor and which in the scoreboard.

Next: Reactive agents and backpressure.

Answers

A1. Check yourself

245 ns is the edge that ends cycle 25 ($10 \times 25 - 5 = 245$), so `out_valid` was 0 in cycle 25. The antecedent held in cycle 24, when the beat was on offer and not taken, and `|=>` checks the consequent one cycle later, so the failure can't be known until the edge that ends cycle 25.

A2. Check yourself

Something that turns the assertions' failure count into a `UVM_ERROR`. UVM counts only its own reports, so `dd105_base_test` saw no errors and passed the test. The labs pass the count from the bound checker to the scoreboard, which reports it at the end of the test.

A3. Check yourself

`a beat waited in 0 cycles` is `n_tried`, and the cover property's count beside it is 0 as well. Either one tells you. Give every assertion you rely on a count like these, or a coverage bin that shows its antecedent held.

A4. Check yourself

`|->`. The rule is about the beat in the cycle of its own handshake, so the consequent has to be checked in the same cycle as the antecedent. With `|=>` the check would look at `out_last` in the cycle after, when it belongs to a different beat or to no beat at all.