



GIOLIT | FORMAL VERIFICATION OF SYSTEMVERILOG DESIGNS

Certificate of Formal Verification

Document GIO-SVA-2CAD57A4AC96 | issued 2026-10-14 17:49:38 UTC (UTC+0000)

PROFESSIONAL LICENSE No. GIOLIT-PRO-2026-000001 licensed to **Giolit Labs**

Every assertion of socbig_top holds.

For every input sequence, from every state after the reset `rst`, in every clock cycle:

- ✓ 25 assertions proved: `p_range`, `p_wrap`, `p_hold`, `p_range`, `p_wrap`, `p_hold`, `a_bound`, `a_not_both`, `a_ptr_sync`, `a_no_ovf` and 15 more

Covered (a run reaches it): `c_wrap`, `c_wrap`, `c_full`, `c_full`, `c_refill`, `c_sys_full`. This is a **mathematical proof** covering all runs of the design, not a simulation of some of them. The proofs are part of this certificate (page 3).

Design	top module socbig_top , 31 properties
File	01_counter.sv (20 lines) SHA-256 7c2e8ade380aff5a0eafd950491ad42d0c5996c8c87eba6b5e63f9b3c29c5739
File	03_fifo.sv (46 lines) SHA-256 38f70a42b64257dbea099b3cfff6f856ebd22301346705bf6ebeb6b6e385662
File	07_hierarchy.sv (21 lines) SHA-256 22f0c68c58c41ecf45b915282dcdaae3719fe3245b8413c41b0a6facad07580b
File	08_assume.sv (36 lines) SHA-256 be5ae73dc667b20df03bf3e86d66fab5acadc74d62355e305048c46a53fe0eba
File	11_margins.sv (40 lines) SHA-256 6965d03abffe468bcb29947e9a9efd71e21e7f159f7f6f2af357f7e9ef5446df
File	socbig_top.sv (59 lines) SHA-256 5c5e0eebe27972c25743ed81404445874b7738ad3b577ce62c41a28cadaa0468
Reset	<code>rst</code> active in the first clock cycle
Issued to	Giolit Labs
Machine ID	5186e2bbb5474f2bb16093c28c3e092a
Machine	95bb1e6c01c1 — Ubuntu 24.04.5 LTS (kernel 7.2.9-orbstack-00391-g2b298d05fbd4); VirtualApple @ 2.50GHz
Verified by	95bb1e6c01c1
Engine	Giolit SystemVerilog Assertion Verifier 1.0.0, build 29f9c97
Date and time	2026-10-14 17:49:38 UTC (UTC+0000) (analysis 174.2 s)
Technique	Exhaustive state exploration, Directed random simulation
Resilience	not rated: no numeric bound in the proved assertions
License	Professional license No. GIOLIT-PRO-2026-000001 , licensed to Giolit Labs

How your client can check this document

- 1. Same design.** sha256sum of each design file must print the SHA-256 listed above.
- 2. Licensed.** Issued under Giolit professional license No. GIOLIT-PRO-2026-000001, licensed to **Giolit Labs**.
- 3. Same result.** `reproduce.sh` re-runs the verification; every counterexample replays in a simulator (evidence/).
- 4. Ask us.** Write to business.giolitlabs@gmail.com quoting document GIO-SVA-2CAD57A4AC96.



GIOLIT — Formal Verification Services

Mathematical proof that your RTL design meets its SystemVerilog assertions.

We offer formal verification as a service: proofs of the SystemVerilog assertions of RTL designs (protocols, data integrity, arbitration, liveness), formal sign-off evidence for safety standards (ISO 26262, DO-254, IEC 61508), assertion writing, and review of counterexamples with your engineers.

www.giolit.combusiness.giolitlabs@gmail.comwww.giolit.com

Properties and results

#	Property	Kind	Line	Result	Method or reason
1	socbig_top.u_cnt0.p_range	assert	16	✓ proved	Exhaustive state exploration
2	socbig_top.u_cnt0.p_wrap	assert	17	✓ proved	Exhaustive state exploration
3	socbig_top.u_cnt0.p_hold	assert	18	✓ proved	Exhaustive state exploration
4	socbig_top.u_cnt0.c_wrap	cover	19	✓ covered	Exhaustive state exploration
5	socbig_top.u_cnt1.p_range	assert	16	✓ proved	Exhaustive state exploration
6	socbig_top.u_cnt1.p_wrap	assert	17	✓ proved	Exhaustive state exploration
7	socbig_top.u_cnt1.p_hold	assert	18	✓ proved	Exhaustive state exploration
8	socbig_top.u_cnt1.c_wrap	cover	19	✓ covered	Exhaustive state exploration
9	socbig_top.u_fifo0.a_bound	assert	40	✓ proved	Exhaustive state exploration
10	socbig_top.u_fifo0.a_not_both	assert	41	✓ proved	Exhaustive state exploration
11	socbig_top.u_fifo0.a_ptr_sync	assert	42	✓ proved	Exhaustive state exploration
12	socbig_top.u_fifo0.a_no_ovf	assert	43	✓ proved	Exhaustive state exploration
13	socbig_top.u_fifo0.a_no_unf	assert	44	✓ proved	Exhaustive state exploration
14	socbig_top.u_fifo0.c_full	cover	45	✓ covered	Directed random simulation
15	socbig_top.u_fifo1.a_bound	assert	40	✓ proved	Exhaustive state exploration
16	socbig_top.u_fifo1.a_not_both	assert	41	✓ proved	Exhaustive state exploration
17	socbig_top.u_fifo1.a_ptr_sync	assert	42	✓ proved	Exhaustive state exploration
18	socbig_top.u_fifo1.a_no_ovf	assert	43	✓ proved	Exhaustive state exploration
19	socbig_top.u_fifo1.a_no_unf	assert	44	✓ proved	Exhaustive state exploration
20	socbig_top.u_fifo1.c_full	cover	45	✓ covered	Exhaustive state exploration
21	socbig_top.u_pipe.a_latency	assert	20	✓ proved	Exhaustive state exploration
22	socbig_top.u_slot.a_nothing_lost	assert	33	✓ proved	Exhaustive state exploration
23	socbig_top.u_slot.a_hold	assert	34	✓ proved	Exhaustive state exploration
24	socbig_top.u_slot.c_refill	cover	35	✓ covered	Exhaustive state exploration
25	socbig_top.u_m.a_level	assert	38	✓ proved	Exhaustive state exploration
26	socbig_top.u_m.a_latency	assert	39	✓ proved	Exhaustive state exploration
27	socbig_top.a_sys_flags0	assert	53	✓ proved	Exhaustive state exploration
28	socbig_top.a_sys_flags1	assert	54	✓ proved	Exhaustive state exploration
29	socbig_top.a_sys_range	assert	55	✓ proved	Exhaustive state exploration
30	socbig_top.a_sys_sync	assert	56	✓ proved	Exhaustive state exploration
31	socbig_top.c_sys_full	cover	57	✓ covered	Directed random simulation

The proofs

For each proved property: the property as written, the method that proved it, and the proof in terms of the design's signals, simplified (relations implied by the signal widths are omitted). The complete machine-readable proofs are in `proof.json`.

socbig_top.u_cnt0.p_range (line 16, proved)

```
p_range: assert property (cnt <= 4'd9);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = one of these cases:
case 1 \$initstate = 1
case 2 \$initstate = 0 $\wedge$ 0 $\leq$ u_cnt0.cnt $\leq$ 9

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_cnt0.p_wrap (line 17, proved)

```
p_wrap: assert property (en && cnt == 4'd9 | => cnt == 4'd0);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = the relations that hold in every case, and one of the cases:
in every case
q_103_s0 = 0
case 1 \$initstate = 1 $\wedge$ q_103_s1 = 0
case 2 \$initstate = 0 $\wedge$ 0 $\leq$ u_cnt0.cnt $\leq$ 9 $\wedge$ q_103_s1 = 0
case 3 \$initstate = 0 $\wedge$ q_103_s1 = 1 $\wedge$ u_cnt0.cnt = 0

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_cnt0.p_hold (line 18, proved)

```
p_hold: assert property (!en | => $stable(cnt));
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = the relations that hold in every case, and one of the cases:
in every case
q_103_s0 = 0
case 1 \$initstate = 1 $\wedge$ past_104_1 = 0 $\wedge$ q_103_s1 = 0
case 2 \$initstate = 0 $\wedge$ 0 $\leq$ past_104_1 $\leq$ 15 $\wedge$ q_103_s1 = 0 $\wedge$ u_cnt0.cnt = 0
case 3 \$initstate = 0 $\wedge$ past_104_1 = 0 $\wedge$ q_103_s1 = 0 $\wedge$ u_cnt0.cnt = 1
case 4 \$initstate = 0 $\wedge$ past_104_1 = 0 $\wedge$ q_103_s1 = 1 $\wedge$ u_cnt0.cnt = 0
case 5 \$initstate = 0 $\wedge$ past_104_1 = 1 $\wedge$ q_103_s1 = 0 $\wedge$ u_cnt0.cnt = 2

```

case 6 $initstate = 0 ∧ past_104_1 = 1 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 1
case 7 $initstate = 0 ∧ past_104_1 = 2 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 3
case 8 $initstate = 0 ∧ past_104_1 = 2 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 2
case 9 $initstate = 0 ∧ past_104_1 = 3 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 4
case 10 $initstate = 0 ∧ past_104_1 = 3 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 3
case 11 $initstate = 0 ∧ past_104_1 = 4 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 5
case 12 $initstate = 0 ∧ past_104_1 = 4 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 4
case 13 $initstate = 0 ∧ past_104_1 = 5 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 6
case 14 $initstate = 0 ∧ past_104_1 = 5 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 5
case 15 $initstate = 0 ∧ past_104_1 = 6 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 7
case 16 $initstate = 0 ∧ past_104_1 = 6 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 6
case 17 $initstate = 0 ∧ past_104_1 = 7 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 8
case 18 $initstate = 0 ∧ past_104_1 = 7 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 7
case 19 $initstate = 0 ∧ past_104_1 = 8 ∧ q_103_s1 = 0 ∧ u_cnt0.cnt = 9
case 20 $initstate = 0 ∧ past_104_1 = 8 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 8
case 21 $initstate = 0 ∧ past_104_1 = 9 ∧ q_103_s1 = 1 ∧ u_cnt0.cnt = 9

```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_cnt1.p_range (line 16, proved)

```
p_range: assert property (cnt <= 4'd9);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

```

Inv = one of these cases:
case 1 $initstate = 1
case 2 $initstate = 0 ∧ 0 ≤ u_cnt1.cnt ≤ 9

```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_cnt1.p_wrap (line 17, proved)

```
p_wrap: assert property (en && cnt == 4'd9 | => cnt == 4'd0);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

```

Inv = the relations that hold in every case, and one of the cases:
in every case
q_103_s0 = 0
case 1 $initstate = 1 ∧ q_103_s1 = 0
case 2 $initstate = 0 ∧ 0 ≤ u_cnt1.cnt ≤ 9 ∧ q_103_s1 = 0
case 3 $initstate = 0 ∧ q_103_s1 = 1 ∧ u_cnt1.cnt = 0

```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_cnt1.p_hold (line 18, proved)

```
p_hold: assert property (!en | => $stable(cnt));
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = the relations that hold in every case, and one of the cases:

in every case

$q_103_s0 = 0$

case 1 $\$initstate = 1 \wedge past_104_1 = 0 \wedge q_103_s1 = 0$

case 2 $\$initstate = 0 \wedge 0 \leq past_104_1 \leq 15 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 0$

case 3 $\$initstate = 0 \wedge past_104_1 = 0 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 1$

case 4 $\$initstate = 0 \wedge past_104_1 = 0 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 0$

case 5 $\$initstate = 0 \wedge past_104_1 = 1 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 2$

case 6 $\$initstate = 0 \wedge past_104_1 = 1 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 1$

case 7 $\$initstate = 0 \wedge past_104_1 = 2 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 3$

case 8 $\$initstate = 0 \wedge past_104_1 = 2 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 2$

case 9 $\$initstate = 0 \wedge past_104_1 = 3 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 4$

case 10 $\$initstate = 0 \wedge past_104_1 = 3 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 3$

case 11 $\$initstate = 0 \wedge past_104_1 = 4 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 5$

case 12 $\$initstate = 0 \wedge past_104_1 = 4 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 4$

case 13 $\$initstate = 0 \wedge past_104_1 = 5 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 6$

case 14 $\$initstate = 0 \wedge past_104_1 = 5 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 5$

case 15 $\$initstate = 0 \wedge past_104_1 = 6 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 7$

case 16 $\$initstate = 0 \wedge past_104_1 = 6 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 6$

case 17 $\$initstate = 0 \wedge past_104_1 = 7 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 8$

case 18 $\$initstate = 0 \wedge past_104_1 = 7 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 7$

case 19 $\$initstate = 0 \wedge past_104_1 = 8 \wedge q_103_s1 = 0 \wedge u_cnt1.cnt = 9$

case 20 $\$initstate = 0 \wedge past_104_1 = 8 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 8$

case 21 $\$initstate = 0 \wedge past_104_1 = 9 \wedge q_103_s1 = 1 \wedge u_cnt1.cnt = 9$

where $\$initstate$ is 1 in the first clock cycle only. Every other name is a signal of the design, $x[k]$ bit k of x .

socbig_top.u_fifo0.a_bound (line 40, proved)

a_bound: assert property (count <= DEPTH);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo0.a_not_both (line 41, proved)

a_not_both: assert property (!(full && empty));

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. The relation Inv below holds at every clock edge of every run (it is inductive). A backward analysis showed that no state satisfying Inv can lead to a violation.

Inv = one of these cases:

case 1 true

case 2 true

where $\$initstate$ is 1 in the first clock cycle only. Every other name is a signal of the design, $x[k]$ bit k of x .

socbig_top.u_fifo0.a_ptr_sync (line 42, proved)

a_ptr_sync: assert property (AW'(wp - rp) == AW'(count));

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo0.a_no_ovf (line 43, proved)

a_no_ovf: assert property (full && push && !pop | => count == DEPTH);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo0.a_no_unf (line 44, proved)

a_no_unf: assert property (empty && pop && !push | => count == 0);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo1.a_bound (line 40, proved)

a_bound: assert property (count <= DEPTH);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo1.a_not_both (line 41, proved)

a_not_both: assert property (!(full && empty));

Method: Exhaustive state exploration. **The proof** is the same invariant Inv as for socbig_top.u_fifo0.a_not_both above: no state in it violates this property either.

socbig_top.u_fifo1.a_ptr_sync (line 42, proved)

a_ptr_sync: assert property (AW'(wp - rp) == AW'(count));

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo1.a_no_ovf (line 43, proved)

a_no_ovf: assert property (full && push && !pop | => count == DEPTH);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_fifo1.a_no_unf (line 44, proved)

a_no_unf: assert property (empty && pop && !push | => count == 0);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_pipe.a_latency (line 20, proved)

a_latency: assert property (@(posedge clk) ##4 dout == \$past(din, 4));

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

socbig_top.u_slot.a_nothing_lost (line 33, proved)

```
a_nothing_lost: assert property (!lost);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

```
Inv = one of these cases:
case 1 $initstate = 1
case 2 $initstate = 0 ∧ u_slot.lost = 0
```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_slot.a_hold (line 34, proved)

```
a_hold: assert property (full && !take && !put | => full && $stable(data_out));
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

```
Inv = the relations that hold in every case, and one of the cases:
in every case
q_112_s0 = 0
case 1 $initstate = 1 ∧ past_113_1 = 0 ∧ q_112_s1 = 0
case 2 $initstate = 0 ∧ past_113_1 = 3 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 3
case 3 $initstate = 0 ∧ past_113_1 = 1 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 1
case 4 $initstate = 0 ∧ past_113_1 = 2 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 2
case 5 $initstate = 0 ∧ past_113_1 = 0 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 0
case 6 $initstate = 0 ∧ 1 ≤ u_slot.data_out ≤ 2 ∧ past_113_1 = 3 ∧ q_112_s1 = 0 ∧ u_slot.full = 1
case 7 $initstate = 0 ∧ 1 ≤ past_113_1 ≤ 3 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 0 ∧ u_slot.full = 1
case 8 $initstate = 0 ∧ 0 ≤ past_113_1 ≤ 1 ∧ 2 ≤ u_slot.data_out ≤ 3 ∧ q_112_s1 = 0 ∧ u_slot.full = 1
case 9 $initstate = 0 ∧ past_113_1 = 2 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 3 ∧ u_slot.full = 1
case 10 $initstate = 0 ∧ past_113_1 = 2 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 1 ∧ u_slot.full = 1
case 11 $initstate = 0 ∧ past_113_1 = 0 ∧ q_112_s1 = 0 ∧ u_slot.data_out = 1 ∧ u_slot.full = 1
case 12 $initstate = 0 ∧ past_113_1 = 3 ∧ q_112_s1 = 1 ∧ u_slot.data_out = 3 ∧ u_slot.full = 1
case 13 $initstate = 0 ∧ past_113_1 = 1 ∧ q_112_s1 = 1 ∧ u_slot.data_out = 1 ∧ u_slot.full = 1
case 14 $initstate = 0 ∧ past_113_1 = 2 ∧ q_112_s1 = 1 ∧ u_slot.data_out = 2 ∧ u_slot.full = 1
case 15 $initstate = 0 ∧ past_113_1 = 0 ∧ q_112_s1 = 1 ∧ u_slot.data_out = 0 ∧ u_slot.full = 1
```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_m.a_level (line 38, proved)

```
a_level: assert property (level <= 4'd12);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

```
Inv = one of these cases:
case 1 $initstate = 1
case 2 $initstate = 0 ∧ 0 ≤ u_m.level ≤ 9
```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.u_m.a_latency (line 39, proved)

```
a_latency: assert property (!busy && req |-> ##[1:6] ack);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = the relations that hold in every case, and one of the cases:

in every case

q_101_age10 = 0 ∧

q_101_age11 = 0 ∧

q_101_age12 = 0 ∧

q_101_s0 = 0

case 1 \$initstate = 1 ∧ q_101_age7 = 0 ∧ q_101_age8 = 0 ∧ q_101_age9 = 0

case 2 \$initstate = 0 ∧ 0 ≤ u_m.age ≤ 1 ∧ q_101_age7 = 0 ∧ q_101_age8 = 0 ∧ q_101_age9 = 0 ∧ u_m.ack = 0 ∧ u_m.busy = 0

case 3 \$initstate = 0 ∧ q_101_age7 = 1 ∧ q_101_age8 = 0 ∧ q_101_age9 = 0 ∧ u_m.ack = 0 ∧ u_m.age = 0 ∧ u_m.busy = 1

case 4 \$initstate = 0 ∧ q_101_age7 = 0 ∧ q_101_age8 = 1 ∧ q_101_age9 = 0 ∧ u_m.ack = 0 ∧ u_m.age = 1 ∧ u_m.busy = 1

case 5 \$initstate = 0 ∧ q_101_age7 = 0 ∧ q_101_age8 = 0 ∧ q_101_age9 = 1 ∧ u_m.ack = 1 ∧ u_m.age = 1 ∧ u_m.busy = 0

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.a_sys_flags0 (line 53, proved)

```
a_sys_flags0: assert property (!(full0 && empty0));
```

Method: Exhaustive state exploration. **The proof** is the same invariant Inv as for socbig_top.u_fifo0.a_not_both above: no state in it violates this property either.

socbig_top.a_sys_flags1 (line 54, proved)

```
a_sys_flags1: assert property (!(full1 && empty1));
```

Method: Exhaustive state exploration. **The proof** is the same invariant Inv as for socbig_top.u_fifo0.a_not_both above: no state in it violates this property either.

socbig_top.a_sys_range (line 55, proved)

```
a_sys_range: assert property (cnt0 <= 4'd9 && cnt1 <= 4'd9);
```

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = one of these cases:

case 1 \$initstate = 1 ∧ 10 ≤ u_cnt1.cnt ≤ 15

case 2 \$initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 1 ∧ u_cnt1.cnt = 2

case 3 \$initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 2 ∧ u_cnt1.cnt = 3

case 4 \$initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 3 ∧ u_cnt1.cnt = 4

case 5 \$initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 4 ∧ u_cnt1.cnt = 5

```

case 6 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 5 ∧ u_cnt1.cnt = 6
case 7 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 6 ∧ u_cnt1.cnt = 7
case 8 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 7 ∧ u_cnt1.cnt = 8
case 9 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 8 ∧ u_cnt1.cnt = 9
case 10 $initstate = 1 ∧ 1 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 0
case 11 $initstate = 1 ∧ 10 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 9
case 12 $initstate = 1 ∧ 2 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 1
case 13 $initstate = 1 ∧ 3 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 2
case 14 $initstate = 1 ∧ 4 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 3
case 15 $initstate = 1 ∧ 5 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 4
case 16 $initstate = 1 ∧ 6 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 5
case 17 $initstate = 1 ∧ 7 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 6
case 18 $initstate = 1 ∧ 8 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 7
case 19 $initstate = 1 ∧ 9 ≤ u_cnt0.cnt ≤ 15 ∧ u_cnt1.cnt = 8
case 20 $initstate = 1 ∧ u_cnt0.cnt = 0 ∧ u_cnt1.cnt = 1
case 21 u_cnt0.cnt = 0 ∧ u_cnt1.cnt = 0
case 22 u_cnt0.cnt = 1 ∧ u_cnt1.cnt = 1
case 23 u_cnt0.cnt = 2 ∧ u_cnt1.cnt = 2
case 24 u_cnt0.cnt = 3 ∧ u_cnt1.cnt = 3
case 25 u_cnt0.cnt = 4 ∧ u_cnt1.cnt = 4
case 26 u_cnt0.cnt = 5 ∧ u_cnt1.cnt = 5
case 27 u_cnt0.cnt = 6 ∧ u_cnt1.cnt = 6
case 28 u_cnt0.cnt = 7 ∧ u_cnt1.cnt = 7
case 29 u_cnt0.cnt = 8 ∧ u_cnt1.cnt = 8
case 30 u_cnt0.cnt = 9 ∧ u_cnt1.cnt = 9

```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.

socbig_top.a_sys_sync (line 56, proved)

a_sys_sync: assert property (en | => cnt0 == cnt1);

Method: Exhaustive state exploration. Every reachable state of the design together with its property checkers was enumerated, for every input sequence; no reachable state violates the property.

The proof. Inv below is the exact set of states of the design (with its property checkers) reachable at a clock edge, as enumerated: every case is a reachable state or a range of them. It holds at the first clock edge, every cycle from a state in it leads to a state in it, and no cycle from a state in it violates the property.

Inv = the relations that hold in every case, and one of the cases:

in every case

q_107_s0 = 0

```

case 1 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 0 ∧ u_cnt1.cnt = 0
case 2 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 1 ∧ u_cnt1.cnt = 1
case 3 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 2 ∧ u_cnt1.cnt = 2
case 4 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 3 ∧ u_cnt1.cnt = 3
case 5 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 4 ∧ u_cnt1.cnt = 4
case 6 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 5 ∧ u_cnt1.cnt = 5
case 7 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 6 ∧ u_cnt1.cnt = 6
case 8 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 7 ∧ u_cnt1.cnt = 7
case 9 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 8 ∧ u_cnt1.cnt = 8
case 10 $initstate = 0 ∧ q_107_s1 = 1 ∧ u_cnt0.cnt = 9 ∧ u_cnt1.cnt = 9
case 11 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 1 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 2
case 12 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 2 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 3
case 13 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 3 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 4
case 14 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 4 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 5
case 15 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 5 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 6
case 16 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 6 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 7
case 17 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 7 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 8
case 18 $initstate = 1 ∧ 0 ≤ u_cnt0.cnt ≤ 8 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 9
case 19 $initstate = 1 ∧ 1 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 0
case 20 $initstate = 1 ∧ 10 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 9
case 21 $initstate = 1 ∧ 2 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 1

```

```
case 22 $initstate = 1 ∧ 3 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 2
case 23 $initstate = 1 ∧ 4 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 3
case 24 $initstate = 1 ∧ 5 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 4
case 25 $initstate = 1 ∧ 6 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 5
case 26 $initstate = 1 ∧ 7 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 6
case 27 $initstate = 1 ∧ 8 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 7
case 28 $initstate = 1 ∧ 9 ≤ u_cnt0.cnt ≤ 15 ∧ q_107_s1 = 0 ∧ u_cnt1.cnt = 8
case 29 $initstate = 1 ∧ 10 ≤ u_cnt1.cnt ≤ 15 ∧ q_107_s1 = 0
case 30 $initstate = 1 ∧ q_107_s1 = 0 ∧ u_cnt0.cnt = 0 ∧ u_cnt1.cnt = 1
case 31 q_107_s1 = 0 ∧ u_cnt0.cnt = 0 ∧ u_cnt1.cnt = 0
case 32 q_107_s1 = 0 ∧ u_cnt0.cnt = 1 ∧ u_cnt1.cnt = 1
case 33 q_107_s1 = 0 ∧ u_cnt0.cnt = 2 ∧ u_cnt1.cnt = 2
case 34 q_107_s1 = 0 ∧ u_cnt0.cnt = 3 ∧ u_cnt1.cnt = 3
case 35 q_107_s1 = 0 ∧ u_cnt0.cnt = 4 ∧ u_cnt1.cnt = 4
case 36 q_107_s1 = 0 ∧ u_cnt0.cnt = 5 ∧ u_cnt1.cnt = 5
case 37 q_107_s1 = 0 ∧ u_cnt0.cnt = 6 ∧ u_cnt1.cnt = 6
case 38 q_107_s1 = 0 ∧ u_cnt0.cnt = 7 ∧ u_cnt1.cnt = 7
case 39 q_107_s1 = 0 ∧ u_cnt0.cnt = 8 ∧ u_cnt1.cnt = 8
case 40 q_107_s1 = 0 ∧ u_cnt0.cnt = 9 ∧ u_cnt1.cnt = 9
```

where \$initstate is 1 in the first clock cycle only. Every other name is a signal of the design, x[k] bit k of x.



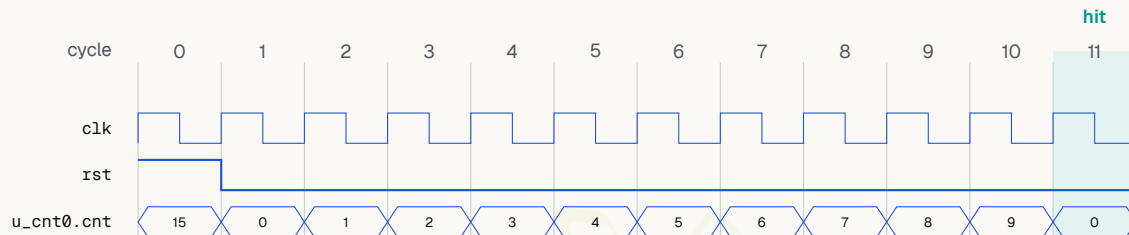
Counterexamples and witnesses

Each run starts at clock cycle 0 (the first clock edge) and is shown as a timing diagram of the signals that change, then cycle by cycle with the values that change. Registers without a reset start with the values shown in cycle 0. The run is also in the bundle as a waveform (VCD, for GTKWave or any waveform viewer) and as a replay testbench that drives the same inputs into the design in your simulator.

socbig_top.u_cnt0.c_wrap (line 19, ✔ covered)

c_wrap: cover property (cnt == 4'd9 ##1 cnt == 4'd0);

Found by: Exhaustive state exploration. 12 clock cycles; the property is reached in cycle 11.



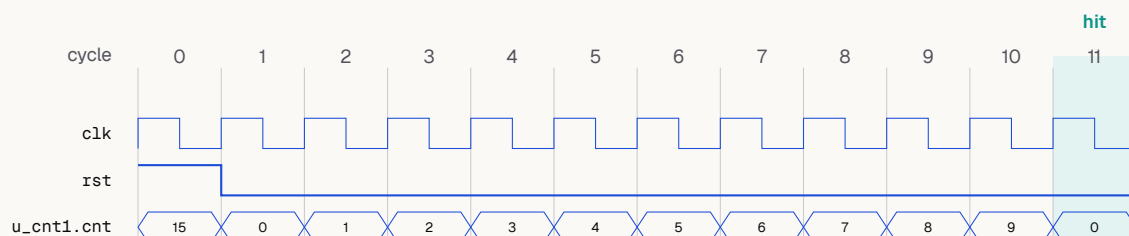
Cycle	Signal values at the clock edge (changed values only)
0	en=1, rst=1, u_cnt0.cnt=15
1	rst=0, u_cnt0.cnt=0
2	u_cnt0.cnt=1
3	u_cnt0.cnt=2
4	u_cnt0.cnt=3
5	u_cnt0.cnt=4
6	u_cnt0.cnt=5
7	u_cnt0.cnt=6
8	u_cnt0.cnt=7
9	u_cnt0.cnt=8
10	u_cnt0.cnt=9
11	u_cnt0.cnt=0 ←

Evidence: evidence/socbig_top_u_cnt0_c_wrap.vcd, evidence/socbig_top_u_cnt0_c_wrap_replay.sv.

socbig_top.u_cnt1.c_wrap (line 19, ✔ covered)

c_wrap: cover property (cnt == 4'd9 ##1 cnt == 4'd0);

Found by: Exhaustive state exploration. 12 clock cycles; the property is reached in cycle 11.



Cycle	Signal values at the clock edge (changed values only)
0	en=1, rst=1, u_cnt1.cnt=15
1	rst=0, u_cnt1.cnt=0
2	u_cnt1.cnt=1
3	u_cnt1.cnt=2
4	u_cnt1.cnt=3
5	u_cnt1.cnt=4

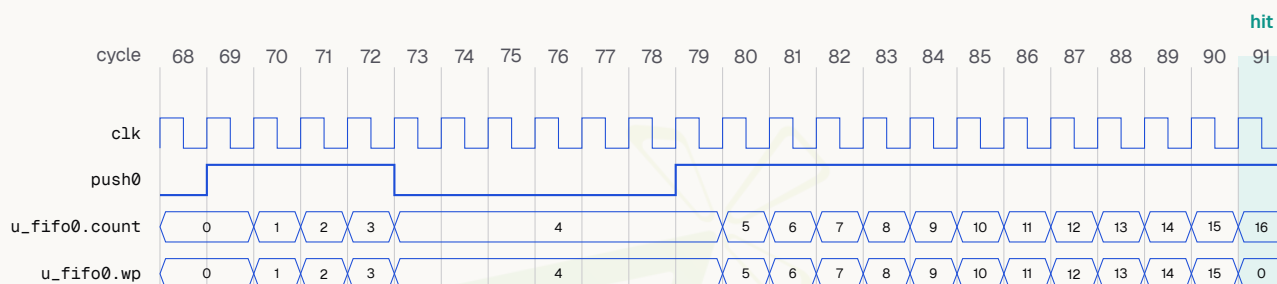
Cycle	Signal values at the clock edge (changed values only)
6	u_cnt1.cnt=5
7	u_cnt1.cnt=6
8	u_cnt1.cnt=7
9	u_cnt1.cnt=8
10	u_cnt1.cnt=9
11	u_cnt1.cnt=0 ←

Evidence: evidence/socbig_top_u_cnt1_c_wrap.vcd, evidence/socbig_top_u_cnt1_c_wrap_replay.sv.

socbig_top.u_fifo0.c_full (line 45, ✔ covered)

c_full: cover property (full);

Found by: Directed random simulation. 92 clock cycles; the property is reached in cycle 91.



The first 68 cycles are not drawn.

Cycle	Signal values at the clock edge (changed values only)
0	pop0=0, push0=0, rst=1, u_fifo0.count=17, u_fifo0.rp=15, u_fifo0.wp=15
1	u_fifo0.count=0, u_fifo0.rp=0, u_fifo0.wp=0
2	no change
3	no change
4	no change
5	no change
6	no change
7	pop0=1
8	no change
9	no change
10	no change
11	no change
12	no change
13	no change
14	push0=1
15	no change
16	no change
17	push0=0
18	no change
19	no change
20	no change
21	no change
22	no change
23	no change
24	no change
25	no change
26	no change
27	no change
28	no change
29	no change
... 32 cycles omitted (complete in the VCD file and results.json) ...	
62	rst=0
63	no change
64	no change
65	no change
66	no change
67	no change

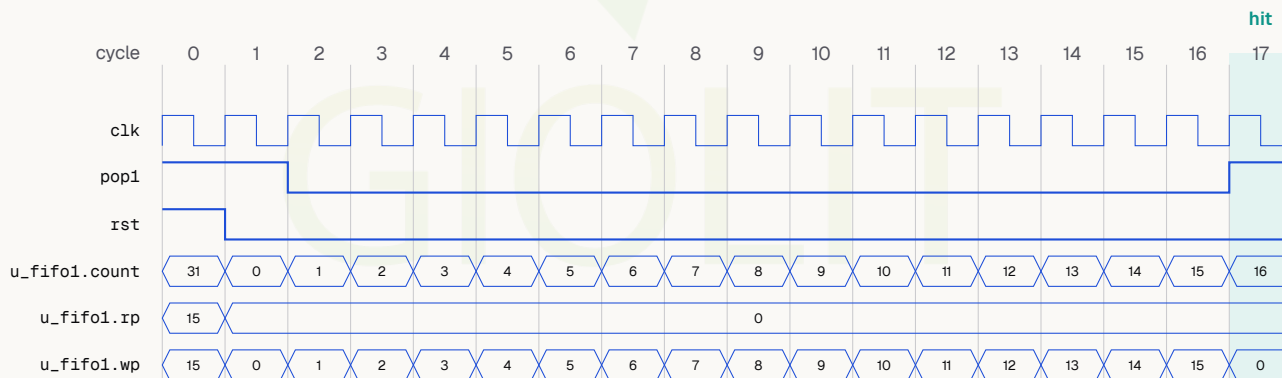
Cycle	Signal values at the clock edge (changed values only)
68	no change
69	push0=1
70	u_fifo0.count=1, u_fifo0.wp=1
71	u_fifo0.count=2, u_fifo0.wp=2
72	u_fifo0.count=3, u_fifo0.wp=3
73	push0=0, u_fifo0.count=4, u_fifo0.wp=4
74	no change
75	no change
76	no change
77	no change
78	no change
79	push0=1
80	u_fifo0.count=5, u_fifo0.wp=5
81	u_fifo0.count=6, u_fifo0.wp=6
82	u_fifo0.count=7, u_fifo0.wp=7
83	u_fifo0.count=8, u_fifo0.wp=8
84	u_fifo0.count=9, u_fifo0.wp=9
85	u_fifo0.count=10, u_fifo0.wp=10
86	u_fifo0.count=11, u_fifo0.wp=11
87	u_fifo0.count=12, u_fifo0.wp=12
88	u_fifo0.count=13, u_fifo0.wp=13
89	u_fifo0.count=14, u_fifo0.wp=14
90	u_fifo0.count=15, u_fifo0.wp=15
91	u_fifo0.count=16, u_fifo0.wp=0 ←

Evidence: evidence/socbig_top_u_fifo0_c_full.vcd, evidence/socbig_top_u_fifo0_c_full_replay.sv.

socbig_top.u_fifo1.c_full (line 45,  covered)

c_full: cover property (full);

Found by: Exhaustive state exploration. 18 clock cycles; the property is reached in cycle 17.



Cycle	Signal values at the clock edge (changed values only)
0	pop1=1, push1=1, rst=1, u_fifo1.count=31, u_fifo1.rp=15, u_fifo1.wp=15
1	rst=0, u_fifo1.count=0, u_fifo1.rp=0, u_fifo1.wp=0
2	pop1=0, u_fifo1.count=1, u_fifo1.wp=1
3	u_fifo1.count=2, u_fifo1.wp=2
4	u_fifo1.count=3, u_fifo1.wp=3
5	u_fifo1.count=4, u_fifo1.wp=4
6	u_fifo1.count=5, u_fifo1.wp=5
7	u_fifo1.count=6, u_fifo1.wp=6
8	u_fifo1.count=7, u_fifo1.wp=7
9	u_fifo1.count=8, u_fifo1.wp=8
10	u_fifo1.count=9, u_fifo1.wp=9
11	u_fifo1.count=10, u_fifo1.wp=10
12	u_fifo1.count=11, u_fifo1.wp=11
13	u_fifo1.count=12, u_fifo1.wp=12
14	u_fifo1.count=13, u_fifo1.wp=13
15	u_fifo1.count=14, u_fifo1.wp=14
16	u_fifo1.count=15, u_fifo1.wp=15

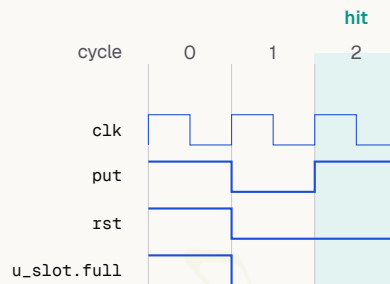
Cycle	Signal values at the clock edge (changed values only)
17	pop1=1, u_fifo1.count=16, u_fifo1.wp=0 ←

Evidence: evidence/socbig_top_u_fifo1_c_full.vcd,evidence/socbig_top_u_fifo1_c_full_replay.sv.

socbig_top.u_slot.c_refill (line 35, ✔ covered)

c_refill: cover property (take ##1 put);

Found by: Exhaustive state exploration. 3 clock cycles; the property is reached in cycle 2.



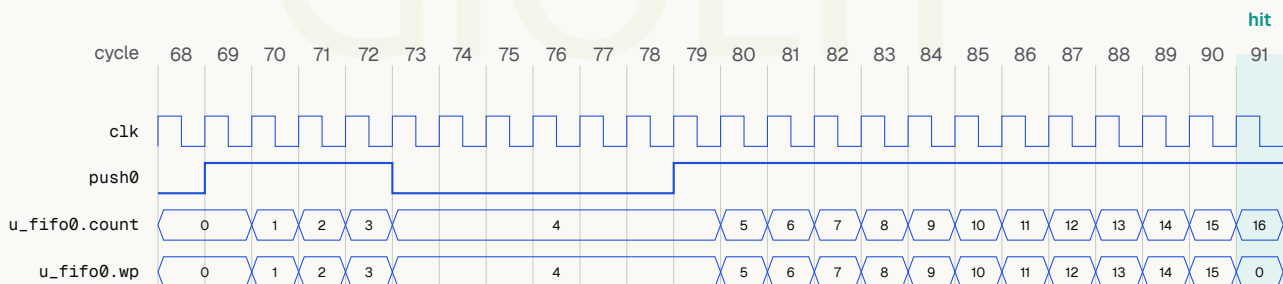
Cycle	Signal values at the clock edge (changed values only)
0	put=1, rst=1, sdata=3, take=1, u_slot.data_out=0, u_slot.full=1, u_slot.lost=0
1	put=0, rst=0, u_slot.full=0
2	put=1 ←

Evidence: evidence/socbig_top_u_slot_c_refill.vcd,evidence/socbig_top_u_slot_c_refill_replay.sv.

socbig_top.c_sys_full (line 57, ✔ covered)

c_sys_full: cover property (full0);

Found by: Directed random simulation. 92 clock cycles; the property is reached in cycle 91.



The first 68 cycles are not drawn.

Cycle	Signal values at the clock edge (changed values only)
0	pop0=0, push0=0, rst=1, u_fifo0.count=17, u_fifo0.rp=15, u_fifo0.wp=15
1	u_fifo0.count=0, u_fifo0.rp=0, u_fifo0.wp=0
2	no change
3	no change
4	no change
5	no change
6	no change
7	pop0=1
8	no change
9	no change
10	no change

Cycle	Signal values at the clock edge (changed values only)
11	no change
12	no change
13	no change
14	push0=1
15	no change
16	no change
17	push0=0
18	no change
19	no change
20	no change
21	no change
22	no change
23	no change
24	no change
25	no change
26	no change
27	no change
28	no change
29	no change
... 32 cycles omitted (complete in the VCD file and results.json) ...	
62	rst=0
63	no change
64	no change
65	no change
66	no change
67	no change
68	no change
69	push0=1
70	u_fifo0.count=1, u_fifo0.wp=1
71	u_fifo0.count=2, u_fifo0.wp=2
72	u_fifo0.count=3, u_fifo0.wp=3
73	push0=0, u_fifo0.count=4, u_fifo0.wp=4
74	no change
75	no change
76	no change
77	no change
78	no change
79	push0=1
80	u_fifo0.count=5, u_fifo0.wp=5
81	u_fifo0.count=6, u_fifo0.wp=6
82	u_fifo0.count=7, u_fifo0.wp=7
83	u_fifo0.count=8, u_fifo0.wp=8
84	u_fifo0.count=9, u_fifo0.wp=9
85	u_fifo0.count=10, u_fifo0.wp=10
86	u_fifo0.count=11, u_fifo0.wp=11
87	u_fifo0.count=12, u_fifo0.wp=12
88	u_fifo0.count=13, u_fifo0.wp=13
89	u_fifo0.count=14, u_fifo0.wp=14
90	u_fifo0.count=15, u_fifo0.wp=15
91	u_fifo0.count=16, u_fifo0.wp=0 ←

Evidence: evidence/socbig_top_c_sys_full.vcd, evidence/socbig_top_c_sys_full_replay.sv.

Safety margins and the Giolit Safety Resilience Score

A proof says the assertion holds; the **safety margin** says by how much. For every numeric bound of a proved assertion — a comparison such as `count <= DEPTH`, or the upper end of a delay window such as `##[1:3]` — the verifier tightened the bound step by step and proved the tightened assertion again. The margin is the largest tightening that still holds: the distance between the values (or latencies) the design actually reaches, over all runs, and the closest value the assertion forbids. It is exact when tightening one step further fails with a counterexample; otherwise it is a lower bound ($\geq$). A margin of 0 means the design reaches its bound — often by design, such as a FIFO that fills up — and that any change that moves the design toward the bound turns the verdict.

Property	Bound	Margin	Allowed	Relative	Score
p_range	cnt <= 4'd9	≥ 0	10 values	—	—
p_range	cnt <= 4'd9	≥ 0	10 values	—	—
a_bound	count <= DEPTH	≥ 0	17 values	—	—
a_bound	count <= DEPTH	≥ 0	17 values	—	—
a_level	level <= 4'd12	≥ 0	13 values	—	—
a_latency	##[1:6]	≥ 0 cycles	6 cycles	—	—
a_sys_range	cnt0 <= 4'd9	≥ 0	10 values	—	—
a_sys_range	cnt1 <= 4'd9	≥ 0	10 values	—	—

The score. Each bound is scored by its relative margin (margin / size of the allowed range): ★★★★★ from 50%, ★★★★★☆ from 25%, ★★★★★☆☆ from 10%, ★★★★★☆☆ above 0, ★☆☆☆☆ when the design reaches the bound. The design's **Giolit Safety Resilience Score** is the score of its weakest bound: not rated.

Semantics and scope

- **Clock.** One step of the model is one clock cycle. Every clocked process is clocked by one global clock unless the run used `--multiclock` (exact edge semantics for several or derived clocks).
- **Inputs.** The top module's inputs are free in every cycle: the results cover *every* input sequence, constrained only by the design's assume properties.
- **Start state.** Registers without an initial value start with arbitrary values; the reset `rst` is active in the first cycle and free afterwards.
- **Values.** Two-state semantics: X and Z values are arbitrary 0/1 values; asynchronous resets act at the clock edge. Several drivers of a net that can release it (z) are checked for contention.
- **Assertions.** Concurrent assertions follow IEEE 1800-2017 (sampled values, `disable iff, $past`); liveness properties are proved by k-liveness or refuted by an infinite run (lasso).
- **Note.** asynchronous resets are modelled as synchronous to the clock (`async2sync`)
- **Note.** asynchronous resets are modelled as synchronous to the clock (`async2sync`)

The design

01_counter.sv

SHA-256 7c2e8ade380aff5a0eafd950491ad42d0c5996c8c87eba6b5e63f9b3c29c5739

```

1 // A modulo-10 counter with reset and enable.
2 // Expected (sva-verify --reset=rst): p_range, p_wrap and p_hold PROVED, c_wrap COVERED.
3 module counter (
4     input logic    clk,
5     input logic    rst,
6     input logic    en,
7     output logic [3:0] cnt
8 );
9     always_ff @(posedge clk)
10         if (rst) cnt <= 4'd0;
11         else if (en) cnt <= (cnt == 4'd9) ? 4'd0 : cnt + 4'd1;
12
13     default clocking cb @(posedge clk); endclocking
14     default disable iff (rst);
15
16     p_range: assert property (cnt <= 4'd9);
17     p_wrap:  assert property (en && cnt == 4'd9 | => cnt == 4'd0);
18     p_hold:  assert property (!en | => $stable(cnt));
19     c_wrap:  cover property (cnt == 4'd9 ##1 cnt == 4'd0);
20 endmodule

```

03_fifo.sv

SHA-256 38f70a42b64257dbea099b3cff6f856ebcd22301346705bf6ebeb6bb6e385662

```

1 // Synchronous FIFO, depth 8, with full/empty flags and an occupancy counter.
2 // Expected: all assertions PROVED.
3 module fifo #(parameter int W = 8, parameter int DEPTH = 8, localparam int AW = $clog2(DEPTH)) (
4     input logic      clk,
5     input logic      rst_n,
6     input logic      push,
7     input logic      pop,
8     input logic [W-1:0] din,
9     output logic [W-1:0] dout,
10    output logic      full,
11    output logic      empty
12 );
13 logic [W-1:0] mem [DEPTH];
14 logic [AW-1:0] wp, rp;
15 logic [AW:0] count;
16
17 assign full = (count == DEPTH);
18 assign empty = (count == 0);
19 assign dout = mem[rp];
20
21 wire do_push = push && !full;
22 wire do_pop = pop && !empty;
23
24 always_ff @(posedge clk or negedge rst_n) begin
25     if (!rst_n) begin
26         wp <= '0; rp <= '0; count <= '0;
27     end else begin
28         if (do_push) begin
29             mem[wp] <= din;
30             wp <= wp + 1'b1;
31         end
32         if (do_pop) rp <= rp + 1'b1;
33         count <= count + do_push - do_pop;
34     end
35 end
36
37 default clocking cb @(posedge clk); endclocking
38 default disable iff (!rst_n);
39
40 a_bound:    assert property (count <= DEPTH);
41 a_not_both: assert property (!(full && empty));
42 a_ptr_sync: assert property (AW'(wp - rp) == AW'(count));
43 a_no_ovf:   assert property (full && push && !pop | => count == DEPTH);
44 a_no_unf:   assert property (empty && pop && !push | => count == 0);
45 c_full:     cover property (full);
46 endmodule

```

07_hierarchy.sv

SHA-256 22f0c68c58c41ecf45b915282dcdaae3719fe3245b8413c41b0a6facad07580b

```

1 // Hierarchy and generate: a pipeline of N registers built from a submodule.
2 // Expected: a_latency UNDECIDED at the full width W = 8 (2^64 states, and no linear invariant);
3 // PROVED with the data width reduced: sva-verify -G W=2 07_hierarchy.sv
4 module stage #(parameter int W = 8) (input logic clk, input logic [W-1:0] d, output logic [W-1:0]
5     q);
6     always_ff @(posedge clk) q <= d;
7 endmodule

```

```

8  module pipeline #(parameter int N = 4, parameter int W = 8) (
9      input logic      clk,
10     input logic [W-1:0] din,
11     output logic [W-1:0] dout
12 );
13     logic [W-1:0] tap [N+1];
14     assign tap[0] = din;
15     for (genvar i = 0; i < N; i++) begin : g
16         stage #(.W(W)) u (.clk(clk), .d(tap[i]), .q(tap[i+1]));
17     end
18     assign dout = tap[N];
19
20     a_latency: assert property (@(posedge clk) ##4 dout == $past(din, 4));
21 endmodule

```

08_assume.sv

SHA-256 be5ae73dc667b20df03bf3e86d66fab5acadc74d62355e305048c46a53fe0eba

```

1  // Assumptions: a one-slot buffer that is only correct if the producer respects 'ready'.
2  // Expected (sva-verify --reset=rst -G W=2 08_assume.sv): a_nothing_lost and a_hold PROVED under
   the assumption
3  // m_wait_ready, c_refill COVERED; without the assumption (add -D NO_ASSUME) a_nothing_lost FAILED
   .
4  // At the full width W = 8, a_hold (a data-stability property) is UNDECIDED.
5  module one_slot #(parameter int W = 8) (
6      input logic      clk,
7      input logic      rst,
8      input logic      put,          // the producer offers data_in
9      input logic [W-1:0] data_in,
10     input logic      take,         // the consumer takes data_out
11     output logic      ready,       // the slot is free
12     output logic [W-1:0] data_out
13 );
14     logic full, lost;
15     assign ready = !full;
16
17     always_ff @(posedge clk)
18         if (rst) begin
19             full <= 1'b0; lost <= 1'b0;
20         end else begin
21             if (put && full) lost <= 1'b1;          // a put into a full slot overwrites data
22             if (put) begin full <= 1'b1; data_out <= data_in; end
23             else if (take) full <= 1'b0;
24         end
25
26     default clocking cb @(posedge clk); endclocking
27     default disable iff (rst);
28
29     `ifndef NO_ASSUME
30         // The producer only puts when the slot is free.
31         m_wait_ready: assume property (put |-> ready);
32     `endif
33     a_nothing_lost: assert property (!lost);
34     a_hold:         assert property (full && !take && !put |=> full && $stable(data_out));
35     c_refill:       cover property (take ##1 put);
36 endmodule

```

11_margins.sv

SHA-256 6965d03abffe468bcb29947e9a9efd71e21e7f159f7f6f2af357f7e9ef5446df

```

1  // Safety margins: a buffer that never holds more than 9 entries, checked against a limit of 12,
   // and a responder that
2  // answers in exactly 3 cycles, checked against a window of 1 to 6 cycles.
3  // Expected (sva-verify --reset=rst --margins 11_margins.sv): both PROVED; a_level has margin 3 of
   // 13 values, a_latency
4  // margin 3 of 6 cycles.
5  module margins (
6      input logic      clk,
7      input logic      rst,
8      input logic      put,
9      input logic      take,
10     input logic      req,
11     output logic [3:0] level,
12     output logic      ack
13 );
14 // the buffer: put adds an entry while fewer than 9 are held, take removes one
15 always_ff @(posedge clk)
16     if (rst) level <= '0;
17     else if (put && !take && level < 4'd9) level <= level + 4'd1;
18     else if (take && !put && level != 4'd0) level <= level - 4'd1;
19
20 // the responder: ack exactly 3 cycles after a request it accepts
21 logic [1:0] age;
22 logic      busy;
23 always_ff @(posedge clk)
24     if (rst) begin
25         busy <= 1'b0; ack <= 1'b0; age <= '0;
26     end else begin
27         ack <= 1'b0;
28         if (!busy && req) begin busy <= 1'b1; age <= '0; end
29         else if (busy) begin
30             if (age == 2'd1) begin ack <= 1'b1; busy <= 1'b0; end
31             else age <= age + 2'd1;
32         end
33     end
34
35 default clocking cb @(posedge clk); endclocking
36 default disable iff (rst);
37
38 a_level:  assert property (level <= 4'd12);
39 a_latency: assert property (!busy && req |-> ##[1:6] ack);
40 endmodule

```

socbig_top.sv

SHA-256 5c5e0eebe27972c25743ed81404445874b7738ad3b577ce62c41a28cadaa0468

```

1  // Large SoC integration: two counters, two depth-16 FIFOs, a 4-stage and an
2  // 8-stage pipeline, a one-slot buffer and a margins block sharing one clock
3  // and reset. Submodule assertions ride along; system checks sit at the top.
4  module socbig_top (
5      input logic      clk,
6      input logic      rst,
7      input logic      en,
8      input logic      push0,
9      input logic      pop0,
10     input logic [7:0] din0,
11     input logic      push1,
12     input logic      pop1,
13     input logic [7:0] din1,
14     input logic      put,
15     input logic [1:0] sdata,

```

```

16  input logic      take,
17  input logic      mput,
18  input logic      mtake,
19  input logic      mreq
20 );
21  logic rst_n;
22  assign rst_n = ~rst;
23
24  logic [3:0] cnt0, cnt1;
25  counter u_cnt0 (.clk(clk), .rst(rst), .en(en), .cnt(cnt0));
26  counter u_cnt1 (.clk(clk), .rst(rst), .en(en), .cnt(cnt1));
27
28  logic [7:0] fdout0, fdout1;
29  logic full0, empty0, full1, empty1;
30  fifo #(.W(8), .DEPTH(16)) u_fifo0 (.clk(clk), .rst_n(rst_n), .push(push0),
31  .pop(pop0), .din(din0), .dout(fdout0), .full(full0), .empty(empty0));
32  fifo #(.W(8), .DEPTH(16)) u_fifo1 (.clk(clk), .rst_n(rst_n), .push(push1),
33  .pop(pop1), .din(din1), .dout(fdout1), .full(full1), .empty(empty1));
34
35  logic [1:0] pdout;
36  pipeline #(.N(4), .W(2)) u_pipe (.clk(clk), .din(sdata), .dout(pdout));
37
38  logic sready;
39  logic [1:0] sdata_out;
40  one_slot #(.W(2)) u_slot (.clk(clk), .rst(rst), .put(put), .data_in(sdata),
41  .take(take), .ready(sready), .data_out(sdata_out));
42
43  logic [3:0] level;
44  logic mack;
45  margins u_m (.clk(clk), .rst(rst), .put(mput), .take(mtake), .req(mreq),
46  .level(level), .ack(mack));
47
48
49  default clocking cb @(posedge clk); endclocking
50  default disable iff (rst);
51
52  // System-level checks over the integrated state.
53  a_sys_flags0: assert property (!(full0 && empty0));
54  a_sys_flags1: assert property (!(full1 && empty1));
55  a_sys_range:  assert property (cnt0 <= 4'd9 && cnt1 <= 4'd9);
56  a_sys_sync:   assert property (en | => cnt0 == cnt1);
57  c_sys_full:   cover property (full0);
58  endmodule

```

Reproducing this result

The bundle delivered with this document contains:

- 01_counter.sv — the verified design file, unchanged
- 03_fifo.sv — the verified design file, unchanged
- 07_hierarchy.sv — the verified design file, unchanged
- 08_assume.sv — the verified design file, unchanged
- 11_margins.sv — the verified design file, unchanged
- socbig_top.sv — the verified design file, unchanged
- certificate.json — the statement of this result, with the SHA-256 of every design and evidence file
- proof.json — the proofs (relations over the design state, and as stated by the engine) and the runs
- results.json — per-property results (machine-readable)

- `evidence/*.vcd` — every counterexample and witness as a waveform
- `evidence/*_replay.sv` — ... and as a replay testbench for a SystemVerilog simulator
- `verifier-output.txt` — the verifier's console report
- `reproduce.sh` — the command that produced this result
- `certificate-password.txt` — the password that opens `certificate.pdf`

Command (run in the bundle folder):

```
sva-verify --reset=rst --timeout=800 --top=socbig_top 01_counter.sv 03_fifo.sv 07_hierarchy.sv  
08_assume.sv 11_margins.sv socbig_top.sv
```

About Giolit

We offer formal verification as a service: proofs of the SystemVerilog assertions of RTL designs (protocols, data integrity, arbitration, liveness), formal sign-off evidence for safety standards (ISO 26262, DO-254, IEC 61508), assertion writing, and review of counterexamples with your engineers. Visit **www.giolit.com** or write to **business.giolitlabs@gmail.com**.



Giolit — Formal Verification Services

Mathematical proof that your RTL design meets its SystemVerilog assertions.

We offer formal verification as a service: proofs of the SystemVerilog assertions of RTL designs (protocols, data integrity, arbitration, liveness), formal sign-off evidence for safety standards (ISO 26262, DO-254, IEC 61508), assertion writing, and review of counterexamples with your engineers.



www.giolit.com



business.giolitlabs@gmail.com



www.giolit.com