

Reverse engineering an ASIC from its layout

Alejandro J. Soto Franco

7 August 2026

1 Summary

The puzzle ships a placed and routed GDS layout, an annotated picture of it, and a value change dump of one unsuccessful attempt. The task is to recover what the chip computes and to read off the string it emits when it is satisfied.

Answer

(* TWO STARS *)

The chip is a checker for Star Battle, also called Two Not Touch, on an 11×11 grid divided into eleven regions. The grid is fed in serially, one cell per clock, row by row, 121 bits in total. `success` rises when every row, every column and every region holds exactly two stars and no two stars touch, diagonals included. Exactly one input satisfies it, and on that input the output generator spells `(* TWO STARS *)`, one character per clock, in OCaml comment brackets.

Results

- A gate-level netlist of 728 standard cells on 725 nets, with 92 state elements and 13 ports, taken from the layout geometry.
- The rule the circuit enforces, including the eleven region shapes, read out of the hardware rather than guessed from the puzzle's name.
- The unique accepted input, established by exhausting the solver rather than by finding one answer and stopping.
- Five easter eggs, and one defect in the shipped design.

Standard of evidence

Every claim below was tested against the circuit, and the two simulators built along the way are required to agree on every verdict. The extraction path was calibrated on the warm-up design, where the correct netlist is published, and it reproduces that netlist exactly: same cells, same nets, and a refinement that separates every cell and every net, so the correspondence is unique rather than merely consistent.

Three intermediate hypotheses were refuted by those checks rather than by inspection. [Section 7](#) records them, since each characterises the design.

2 Netlist extraction

2.1 Information retained in the layout

What the GDS still carries determines the method: geometric pattern recognition if the cell identities were stripped, connectivity recovery if they survived.

Cell names and pin labels survive

Both the warm-up layout and the puzzle layout instantiate cells by their library names, `sky130_fd_sc_hd__nand2_2` and the rest, and every cell definition still carries its pin labels on the li1 label layer. The puzzle's top cell is labelled too: `I`, `clk`, `enable`, `rst_n`, `0[0]` to `0[7]`, and `success`.

The repository states that internal names were removed; the removal covers net and instance naming, leaving cell types intact. Recognising cells by geometry is therefore unnecessary, and the task reduces to recovering the routed connectivity and tying each placed cell's pins to it.

The library is Sky130, which is open, so the cell definitions, their pin directions and their boolean functions are all available as data.

2.2 Extraction procedure

`tools/extract.py` does four things:

1. collects every standard cell placement under the top cell, with its transform, skipping the via cells;
2. merges li1 through met5 into nets across mcon and the via layers, using KLayout's connectivity engine;
3. maps each cell definition's pin labels into top-level coordinates and probes them against those nets;
4. names the nets that the top-level port labels land on.

The device layers are deliberately left out of the connectivity. Including licon1 and diff would short a cell's own pins together through its transistors. Left out, each cell's internal li1 forms an island that the router touches at the pins alone, which is the condition the probe relies on.

Extraction pitfalls

A label only means something on its own layer. Probing a li1 pin label against met1 finds whatever passes overhead, which is usually a power rail, and ties the pin to it silently. Extraction must also be run flat. Left hierarchical, a net that stays inside a cell belongs to that cell's circuit, and every instance of that cell shares it, so cluster identifiers are unique only within a definition. Fifteen clock buffers then appear to drive a single net.

2.3 Extraction results

	warm-up	puzzle
standard cells (excluding taps, decaps and fill)	79	728
nets	84	725
ports	6	13
distinct cell types	18	66
state elements	16	92
unconnected pins	0	0

The state elements are 84 `dfrtp` with active-low reset, four `dfstp` which reset to one, and four `dfxtp` with no reset at all. The undefined initial state of those four is taken up in [Section 5](#).

The clock is a two-level tree: one `clkbuf_16` from the `clk` port into sixteen `clkbuf_8`, each driving five or six flops. Fifteen `clkbuf_4` cells hang off those outputs as loads whose own outputs go nowhere.

3 Validation

A netlist recovered almost correctly still simulates and still produces an answer, which makes silent error the principal risk. Four checks constrain it.

3.1 Calibration against the warm-up design

The warm-up ships its synthesised netlist alongside its layout, so the extraction can be run against a known result. Comparing them by name is meaningless, since the extractor invents its own. What must hold is that the two cell and net graphs are isomorphic with cell types preserved, pin names preserved on every edge, and the ports pinned to themselves.

`tools/compare.py` colours both graphs by iterated refinement, seeded from the ports. Both partitions come out discrete, 79 cells into 79 classes and 84 nets into 84 classes, and the colour multisets agree. A discrete refinement is a unique correspondence, so this is an isomorphism proof rather than a coincidence of counts.

Simulated against the published source, the recovered warm-up netlist reproduces $A + B = 496$ over 335 input pairs.

3.2 Replay of the supplied trace

`example_inputs.vcd` records 0 and `success` as well as the inputs, so it is a reference trace for the puzzle itself. Replayed against the recovered netlist it matches on all 623 samples of 0 and all 623 of `success`.

Coverage limits of a single trace

That check passed while the Python model still contained a real defect. The constant cells were classified apart from the combinational ones and never evaluated, so every tie-high in the design read as zero. The supplied trace exercises no path through a tie cell, so it passed throughout. Running two independent simulators against each other on random inputs caught it, which is now `tools/crosscheck.py`, by which time it had already contaminated a solver run.

3.3 Cell model equivalence

Simulation needs models for the 66 cell types. Sky130 publishes each cell's boolean function twice, as `equation` in `definition.json` and as `function` in the timing data, and the two disagree.

The `equation` field names inverted inputs by their uninverted form, so `nor3b` is written in terms of `C` when the port is `C_N`, and several entries do not balance their brackets. The timing data uses the real pin names and is what the generators read.

Both generated model sets are then checked rather than trusted. Each of the 62 combinational cells is compared against the library's own model over its complete truth table, at most six inputs, so the check is exhaustive.

3.4 Cross-simulator agreement

The work uses two independent paths: generated Verilog under iverilog with the library models, and a Python cycle simulator over expression trees, which is also what the solver unrolls. Across 216 inputs the two agree on `success` every time.

They differ on the reply text in 15 of those cases, for a reason that is part of the design and is taken up in [Section 7](#).

4 Circuit function

4.1 Input protocol

The supplied dump gives the shape of the exchange. `rst_n` is pulsed low, `enable` goes high for 121 rising edges with one bit presented on `I` at each, and when `enable` falls the output generator spells a message on `O`, one character per clock. The recorded attempt is answered `TRY AGAIN` and `success` never rises.

Feeding other inputs draws out other answers. An empty grid is answered `EMPTY SKY`, a full one `BIG BANG`. Each reply diagnoses the input it was given, which makes the reply vocabulary itself a source of information about the predicate.

4.2 Constraint recovery by single-cell excitation

Inference from the reply vocabulary alone proved unreliable, and two hypotheses formed that way were later refuted. The constraint structure is instead measured directly from the circuit.

Single-cell excitation

Feed a grid holding a single star and compare the final state against the empty grid. The state bits that moved are the accumulators that cell contributes to. Repeat for all 121 cells and the constraint groups fall out of the hardware.

Doing that, `tools/probe_state.py` finds eleven accumulators fed by the eleven columns, and eleven more fed by irregular groups of cells of sizes 4, 5, 6, 7, 8, 8, 9, 11, 14, 21 and 28. Those sum to 121, they are disjoint, and each one is orthogonally connected. They are the regions of a Star Battle puzzle, recovered without assuming the puzzle was Star Battle.

4.3 Recovered predicate

Written out, `success` rises exactly when the 121 bits, taken row by row as an 11×11 grid, satisfy

1. exactly two stars in each row,
2. exactly two stars in each column,
3. exactly two stars in each of the eleven regions of [Figure 1](#),
4. no two stars orthogonally or diagonally adjacent.

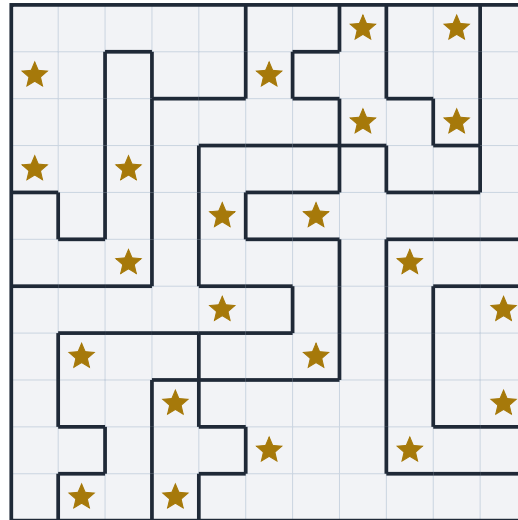


Figure 1: The eleven regions, recovered from the accumulators each grid cell feeds, with the circuit's only accepted input marked. Two stars sit in every row, every column and every region, and no two touch.

This is Star Battle with two stars, the variant also sold as Two Not Touch. The circuit names it: a grid satisfying the first three rules while allowing two stars to touch is answered **TWO NOT TOUCH**.

4.4 Output vocabulary

reply	condition
EMPTY SKY	no stars at all
BIG BANG	every cell a star
TWO NOT TOUCH	every count correct, with two stars touching
TRY AGAIN	anything else
(* TWO STARS *)	the puzzle solved

The output generator is the block marked in the supplied picture. It accounts for 228 of the 728 cells, and the **success** cone lies wholly inside the **0** cone, so it can be left alone while the rule is worked out and simulated afterwards to read the answer, which is how the puzzle text suggests approaching it.

5 Solution

5.1 Exhaustive solution search

`tools/solve.py` unrolls the netlist one cycle per input bit over the same expression trees the simulator uses, leaves the 121 input bits free, and asserts **success**. Adding the negation of each answer found and asking again enumerates the accepted set until the solver reports unsatisfiable.

It reports unsatisfiable after one. The circuit accepts exactly one input out of 2^{121} , and it carries 22 stars.

Solving the four rules of Section 4 directly on the recovered regions independently yields exactly one grid, and the two agree. The predicate read out of the hardware and the input the hardware accepts therefore determine each other, so neither rests on the other having been guessed correctly.

Reading**PER ARENAM AD ASTRA**

Through the sand to the stars, on *per aspera ad astra*, with *arenam* standing in for the silicon. It also names the subject matter before any of the logic has been read.

6.2 A message inside the example input

The supplied dump feeds two attempts of 121 bits, neither accepted. Laid out as the grid the circuit reads, both leave columns 7 to 10 entirely empty, which no arbitrary guess would do. The occupied width is seven columns, and reading each row as seven bits with the first bit fed as the least significant gives printable ASCII on all 22 rows of the two attempts:

Reading**The night sky awaits**

The example is therefore a wrong answer chosen to carry text, which is what the version string quoted below points at.

6.3 A leap second in the supplied dump

`example_inputs.vcd` is dated `Sat Dec 31 23:59:60 2016`. The sixtieth second is the leap second inserted at the end of that year, so the timestamp is a real instant that most date libraries reject.

6.4 An instruction in the version string

The same file carries a `$version` string reading “Leave no stone unturned!”, which is also a fair description of the search that finds the first two.

6.5 The answer in OCaml brackets

The success message is delivered as `(* TWO STARS *)`. The surrounding `(*` and `*)` are OCaml comment syntax, a signature of the authors’ house language, and they are emitted by the hardware one character per clock alongside the message itself.

7 Design observations**7.1 Undriven net in the output generator**

One net in the puzzle layout has no driver. It runs from the `A1` pin of an `a31oi_2` through `met1` and a `met2` jog to the `A1` pin of the `a31io_2` beside it, and nothing else touches it. This is not an extraction miss: the net’s full geometry was traced and it is a closed island of two gate inputs.

Its forward cone reaches `O[1]` and `O[4]` and no further, so it cannot affect `success`, and the answer is untouched. It does perturb the reply text: with the net held low the `TWO NOT TOUCH` message comes out as `TWO"NOT TOUCH`, and held high other characters shift instead. Under iverilog the net is undriven, the affected bits are unknown, and the message prints cleanly. A floating input on a real die would settle wherever leakage and coupling put it.

7.2 Corrected hypotheses

Three hypotheses were adopted at intermediate stages and subsequently eliminated by the checks of [Section 3](#). Each is stated with the evidence that eliminated it, since the eliminations are what constrain the final result.

1. *The predicate is two stars per row and column.* Grids satisfying these counts alone are rejected by the circuit. `tools/rule.py` established this by generating grids that satisfy the hypothesis and querying the circuit directly, which returned a rejection for every one.
2. *Adjacency is unconstrained.* A solver run returned sixteen accepted inputs, fifteen carrying touching stars. All sixteen proved to be artefacts of the constant-cell defect of [Section 3](#): with every tie-high evaluating as zero, the adjacency term could never assert. The independent Verilog path disagreed on those inputs, which located the defect. With it corrected, the accepted set is a single grid and adjacency is enforced.
3. *The undefined initial state accounts for the disagreement.* The four unreset flops were the natural candidate, and enumerating all sixteen power-up states exonerated them while the defect remained undetected. Assigning those flops definite values in Verilog removed the unknown values that were masking the comparison, after which the defect appeared immediately as a single flop diverging at cycle 31.

8 Reproduction

Everything runs from the puzzle repository and the open Sky130 cell library. No commercial tools are involved.

stage	command
extract the warm-up	<code>tools/extract.py warmup/04_final.gds</code>
prove it against the published netlist	<code>tools/compare.py</code>
extract the puzzle	<code>tools/extract.py puzzle.gds</code>
audit drivers and loads	<code>tools/check.py</code>
generate cell models	<code>tools/genmodels.py</code>
prove the models exhaustively	<code>tools/checkmodels.py, checktrees.py</code>
replay the supplied dump	<code>tools/replay.py</code>
agree between simulators	<code>tools/crosscheck.py</code>
read the constraint groups	<code>tools/probe_state.py</code>
solve and exhaust	<code>tools/enumerate.py</code>
check every claim	<code>tools/verify.py</code>

`make all` runs the chain and ends on `tools/verify.py`, which re-derives the regions, re-solves the puzzle, re-runs both simulators over every power-up state and prints the answer. It exits non-zero if any claim fails.

Tool chain

KLayout's Python module for the layout, Icarus Verilog for simulation, z3 for the solving, and the Sky130 high-density cell library for the models and the published cell functions. Yosys, Magic, ngspice and Verilator were installed during the work and turned out not to be needed: once the cell names were found intact, the problem was connectivity and search rather than device extraction.