TCL FOR PHYSICAL DESIGN & CONNECTIVITY ENGINEERS

Tcl Netlist Drills

Fifteen exercises that teach Tcl the way you'll actually use it on the job: reading a structural Verilog netlist, walking a DEF's `COMPONENTS` / `PINS` / `NETS` sections, and answering the questions an EDA tool's Tcl shell gets asked all day — "how many buffers are there," "what does this net connect to," "is anything undriven." Every exercise below points at this project's own `gate_netlist.v` and `design_place.def` — the same two files sitting in the repo, not invented stand-ins — so what you see here is what you'll see when you open them for real.

BASIC · 5 MEDIUM · 5 DIFFICULT · 5

JUMP TO A QUESTION

Which Tcl console to run these in

Every exercise below is plain Tcl — variables, procs, `regexp`, `dict`, file I/O — and each one opens its data file directly with `open` / `gets` / `read`, not through any GUI command. That means all fifteen run correctly in a bare `tclsh`, in the **Block Netlist Checker** tab's console, or in the **DEF Feedthrough Checker** tab's console — whichever has the working directory with `gate_netlist.v` / `design_place.def` in it. Each question is tagged below with the console it's most natural to run in, based on which real file it touches:

`DEF console` reads `design_place.def` `Gate console` reads `gate_netlist.v`

`either / tclsh` no file, or toy data only

One shadowing gotcha: D3 has you define a top-level `proc all_fanout` — a name that's *already* registered as a real domain command in both consoles. Type it directly into a real console (rather than a plain `tclsh`) and your toy version silently overrides the real one for the rest of that console's session, until you restart the app or reload the tab. D4's `get_cells` is safe from this — it's defined inside its own `::edaq::` namespace, so it can't collide with the real top-level command.

Reference files used throughout

These aren't invented sample files — they're excerpts copied verbatim from this project's own `gate_netlist.v` and `design_place.def`, the same files that sit in the repo next to `soc_checker_gui.py`. What's printed below is trimmed to fit the page: the real `gate_netlist.v` has all five feedthrough chains (26 instances total, this excerpt shows only chain `ft0`), and the real `design_place.def` has 10,000 components, 4,000 pins, and 4,426 nets. Every exercise that names one of these two files means the real, full copy in your repo — run your script against that, not the excerpt on this page.

gate_netlist.v — *excerpt, chain ft0 of 5*

```
module gate_top (
    clk, a, b, c, d, sel, core_out,
    feedthrough_in_ft0, feedthrough_out_ft0
    /* , ...ft1..ft4 ports, omitted here */
);
    input clk, a, b, c, d, sel;
    output core_out;
    input feedthrough_in_ft0;
    output feedthrough_out_ft0;

    wire n_core_and, n_core_or, n_core_xor,
n_core_mux,
         n_core_nand, n_core_nor, n_core_d;
    wire n_ft0_0, n_ft0_1;

    AND2X1  u_core_and (.A(a), .B(b),
.Y(n_core_and));
    OR2X1   u_core_or  (.A(c), .B(d),
.Y(n_core_or));
    XOR2X1  u_core_xor (.A(n_core_and),
.B(n_core_or), .Y(n_core_xor));
    MUX2X1  u_core_mux (.A(n_core_and),
.B(n_core_or), .S(sel), .Y(n_core_mux));
    NAND2X1 u_core_nand(.A(n_core_xor),
.B(n_core_mux), .Y(n_core_nand));
    NOR2X1  u_core_nor (.A(n_core_mux),
.B(n_core_nand), .Y(n_core_nor));
    AOI21X1 u_core_aoi (.A(n_core_nand),
.B(n_core_nor), .C(a), .Y(n_core_d));
    DFFX2   u_core_reg (.D(n_core_d),
.CK(clk), .Q(core_out));

    // feedthrough chain 'ft0': BUFX2 -> INVX1
-> BUFX1 (0 repeater flops)
    BUFX2 u_ft0_0_bufx2
(.A(feedthrough_in_ft0), .Y(n_ft0_0));
    INVX1 u_ft0_1_invx1 (.A(n_ft0_0),
.Y(n_ft0_1));
    BUFX1 u_ft0_2_bufx1 (.A(n_ft0_1),
.Y(feedthrough_out_ft0));

    // ft1..ft4 continue below in the real
file with 1-2
    // DFFX1 repeater flops inserted per chain
-- 18 more instances
endmodule
```

design_place.def — *excerpt, full file has 10,000 / 4,000 / 4,426*

```
VERSION 5.8 ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN block_top ;
UNITS DISTANCE MICRONS 1000 ;
DIEAREA ( 0 0 ) ( 500000 500000 ) ;

ROW ROW_0 CORE_SITE 0 0 N DO 2500 BY 1 STEP
200 0 ;
ROW ROW_1 CORE_SITE 0 2000 N DO 2500 BY 1 STEP
200 0 ;
... 248 more rows ...

COMPONENTS 10000 ;
- u_000000_invx1 INVX1 + PLACED ( 287000
104000 ) N ;
- u_000001_invx2 INVX2 + PLACED ( 357800
208000 ) N ;
- u_000002_dffx1 DFFX1 + PLACED ( 428200
312000 ) N ;
- u_000003_dffx2 DFFX2 + PLACED ( 103600
334000 ) N ;
- u_000004_bufx2 BUFX2 + PLACED ( 186400
166000 ) N ;
... 9,995 more components ...
END COMPONENTS

PINS 4000 ;
- ft_0000_in  + NET ft_0000_h0 + DIRECTION
INPUT + USE SIGNAL
  + LAYER M1 ( -50 -50 ) ( 50 50 ) + PLACED (
500000 225400 ) N ;
- ft_0000_out + NET ft_0000_h0 + DIRECTION
OUTPUT + USE SIGNAL
  + LAYER M1 ( -50 -50 ) ( 50 50 ) + PLACED (
0 249400 ) N ;
... 3,998 more pins ...
END PINS

NETS 4426 ;
- ft_0000_h0 ( PIN ft_0000_in ) ( PIN
ft_0000_out ) ;
- ft_0002_h0 ( PIN ft_0002_in ) (
u_000000_invx1 A ) ;
- ft_0002_h1 ( u_000000_invx1 Y ) (
u_000001_invx2 A ) ;
- ft_0002_h2 ( u_000001_invx2 Y ) (
```

```
u_000002_dffx1 D ) ;
- ft_0002_h3 ( u_000002_dffx1 Q ) ( PIN
ft_0002_out ) ;
... 4,421 more nets ...
END NETS

END DESIGN
```

Basic — core Tcl

No netlist parsing yet — just the handful of Tcl mechanics everything else is built from: substitution, lists, control flow, file I/O, and procs.

B1 variables & substitution **either / tclsh**

Print an instance's identity

Store an instance name (`u_ft0_0_bufx2`) and a cell type (`BUFX2`) in two variables, then print **"Cell u_ft0_0_bufx2 is of type BUFX2."** on one line. Then try the same thing with the string in `{braces}` instead of `"quotes"` and explain out loud why the output changes.

B2 lists & foreach **either / tclsh**

Number the instances

Given `set instances {u_core_and u_core_or u_ft0_0_bufx2}` , print how many instances there are, then loop over the list printing each one with a zero-based index, e.g. `(0) u_core_and` .

B3 conditionals & string match **either / tclsh**

`is_buffer_cell`

Write a proc `is_buffer_cell {cell_type}` that returns `1` if `cell_type` starts with `BUFX` or `INVX` (the naming convention this project's DEF generator uses for pass-through cells), and `0` otherwise. Test it against `BUFX1` , `INVX2` , and `DFFX2` .

B4 file I/O DEF console

Count a keyword in a file

Open `design_place.def` and count how many lines contain the substring `COMPONENTS` (there should be two: the section header and the `END` line — that's true whether the section holds 3 components or the real file's 10,000). Read the file one line at a time rather than slurping it whole — you'll want that habit for D5.

B5 procs & default args either / tclsh

`report_line`

Write a proc `report_line {name value {unit ""}}` that prints a left-aligned, colon-terminated label followed by the value and, if given, a unit — e.g. `report_line "Die width" 500 "um"` → `Die width: 500 um`. Use `format` for the alignment rather than hand-padding with spaces.

Medium — parsing real design data

Now the netlist and DEF text itself becomes the input: regular expressions, arrays, dicts, and the standard shape of a small "extract → summarize" script.

M1 regexp capture groups DEF console

Parse one COMPONENTS line

Given the line `- u_000000_invx1 INVX1 + PLACED ( 287000 104000 ) N ;` (a real line from `design_place.def`), write a proc `parse_component_line {line}` that uses a single `regexp` call with capture groups to pull out the instance name, cell type, x, and y, returning them as a 4-element list (or an empty string if the line doesn't match).

M2 arrays & section tracking DEF console

Cell-type census

Using `parse_component_line` from M1, write a script that reads `design_place.def`, tracks whether you're currently inside the `COMPONENTS ... END COMPONENTS` block, and tallies instance counts per cell type into a Tcl *array*. Print the census sorted alphabetically by cell type — against the real file this is a one-shot census of a genuine 10,000-cell design, not a toy.

M3 regexp -all -inline Gate console

Every instantiation in one pass

Against `gate_netlist.v`, use a single `regexp -all -inline` call to find every module instantiation (pattern: a cell type, then an instance name starting with `u_`, then a parenthesized port list ending in `;`) and print each `type → instance` pair — the real file has 26 (8 core gates plus 18 across the five feedthrough chains). Note that the match list comes back flat — think about how many elements each match contributes before you loop over it.

M4 dict & custom sort DEF console

Top-N cell types

Redo M2's census using a `dict` instead of an array (`dict incr` is your friend). Then write `top_n_types {census n}` that returns the `n` most common cell types as `{count type}` pairs, most common first. You'll need `lsort` with an index and a direction, not the default alphabetical sort.

M5 catch / try either / tclsh

A lookup that never throws

Given a dict mapping pin name → `{x y}`, write `safe_get_pin_location {pin_name pin_dict}` that returns the location, or the string `UNKNOWN` if the pin isn't in the dict — without ever letting `dict get`'s error escape. Write it once with `catch` and once with Tcl 8.6's `try`/`on error`, and note which reads more clearly to you.

Difficult — connectivity, geometry, and scale

These mirror the checks a real connectivity/DEF tool runs: driver analysis, physical distance, graph traversal, a tiny query language, and the point where "just read the whole file" stops working.

D1 connectivity from NETS **DEF console**

Undriven and multiply-driven nets

Assume you've parsed a DEF's `NETS` section into a dict of `net_name → {{inst pin dir} {inst pin dir} ...}`, where `dir` is `IN` or `OUT`. Write `classify_drivers {nets}` that returns two lists: nets with zero output-direction connections (undriven), and nets with more than one (multiply-driven) — the same two defects `checkers.py`'s `undriven_nets` / `multiply_driven_nets` checks flag, now in Tcl.

D2 geometry **DEF console**

Close feedthrough pairs

Write `manhattan_distance {x1 y1 x2 y2}`, then `find_close_feedthrough_pairs {pins threshold_um dbu_per_um}` that, given a dict of `pin_name → {x y}` in database units, finds every `<id>_in` / `<id>_out` pair whose Manhattan distance is under the threshold (converted from microns to DBU first) — this project's own DEF close-pair check, minus the DEF parsing.

D3

graph traversal **either / tclsh**

`all_fanout`

Given a dict mapping `inst.pin → {list of inst.pin it drives}`, write `all_fanout {start fanout_map}` returning every instance/pin reachable downstream of `start`, direct or indirect. Your traversal has to tolerate a cycle in the map without looping forever — recall this project's own `feedthrough_loops` check exists precisely because real feedthrough chains sometimes loop back on themselves.

```
proc all_fanout {start fanout_map} {
    # your traversal here -- BFS or DFS, your choice
}
```

> **Shadowing risk:** `all_fanout` is already a real domain command in both the Block Netlist Checker and DEF Feedthrough Checker consoles. Try this one in a plain `tclsh` — typing it into a real console overrides the genuine command until that tab is reloaded or the app restarted.

D4

namespaces & a mini DSL `either / tclsh`

A tiny `get_cells`

Build a namespace `::edaq::` with `load_cells {cells}` (takes a list of `{name ... type ... x ... y ...}` dicts) and `get_cells {args}` supporting a single `-filter {field == value}` option, so that `::edaq::get_cells -filter {type == BUFX1}` returns just the matching instance names — a miniature of the `get_cells` / `get_ports` commands this project's own DEF-viewer Tcl console registers via `tk::createcommand`.

```
namespace eval ::edaq {
    variable design_cells {}
    proc load_cells {cells} { # ... }
    proc get_cells {args}   { # ... }
}
```

> **Not a shadowing risk here:** since this `get_cells` lives inside the `::edaq::` namespace, it can't collide with the real top-level `get_cells` command in either console — that's exactly the namespacing benefit the solution's note calls out. It's still listed as "either" because it uses only invented, in-memory data, not either real file.

D5

performance `DEF console`

Streaming vs. slurp, and proving it

Write two versions of the M2 census over `design_place.def` — a genuinely large DEF, 26,695 lines and 1.3 MB, not a hypothetical one: one that reads line-by-line with `gets` and a running `dict`, and one that `read`s the whole file into a string and runs a single `regexp -all -inline` over it. Time both with `clock clicks -milliseconds`. Then explain, in your own words, why the second approach's memory use grows with file size in a way the first one's doesn't — even before either one is "slow."

Companion document: the fully worked solutions live in **Tcl Netlist Solutions**. Work each tier before checking it — the difficulty labels are there so you can gauge where to spend the most time, not so you can skip ahead.