TCL FOR PHYSICAL DESIGN & CONNECTIVITY ENGINEERS

Tcl Netlist Solutions

Worked solutions for every exercise in **Tcl Netlist Drills**, numbered to match. Each one includes the code, its output against this project's own `gate_netlist.v` and `design_place.def` — real files, real output, not a toy fixture — and a short note on the Tcl idiom the exercise was actually testing. Read that even if your own script already ran correctly.

BASIC · 5 MEDIUM · 5 DIFFICULT · 5

JUMP TO A SOLUTION

Basic

B1 variables & substitution

Print an instance's identity

```
set cell_name "u_ft0_0_bufx2"
set cell_type "BUFX2"
puts "Cell $cell_name is of type $cell_type."
```

```
→ Cell u_ft0_0_bufx2 is of type BUFX2.
```

Why: double-quoted strings let Tcl substitute `$variable` and `[command]` as it builds the string. The same literal in `{braces}` suppresses all substitution — `puts {Cell $cell_name...}` prints the four characters `$cell_name` verbatim. Braces are for text you want Tcl to leave alone (a regexp pattern, a proc body); quotes are for text you want built.

Number the instances

```
set instances {u_core_and u_core_or u_ft0_0_bufx2}
puts "Number of instances: [llength $instances]"

set i 0
foreach inst $instances {
    puts "  ($i) $inst"
    incr i
}
```

```
→ Number of instances: 3
(0) u_core_and
(1) u_core_or
(2) u_ft0_0_bufx2
```

> **Why:** a Tcl list is just a specially-quoted string, so `llength` / `foreach` work on it without any separate "list type." If you only need the index, `foreach {i inst} [lmap i [lrepeat ...` gets awkward fast — plain `incr` is the idiom.

B3

conditionals & string match

`is_buffer_cell`

```
proc is_buffer_cell {cell_type} {
    if {[string match "BUFX*" $cell_type] || [string match "INVX*" $cell_type]} {
        return 1
    }
    return 0
}

puts [is_buffer_cell "BUFX1"] ;# 1
puts [is_buffer_cell "INVX2"] ;# 1
puts [is_buffer_cell "DFFX2"] ;# 0
```

> **Why:** `string match` does shell-style glob matching (`*`, `?`, `[abc]`) and is cheaper to read and write than a regexp when you only need a prefix/suffix check. Reach for `regexp` (M1) once you need capture groups or anchors more specific than glob offers.

B4 `file I/O`

Count a keyword in a file

```
set count 0
set fh [open "design_place.def" r]
while {[gets $fh line] >= 0} {
    if {[string first "COMPONENTS" $line] >= 0} {
        incr count
    }
}
close $fh
puts "Lines mentioning COMPONENTS: $count"
```

```
→ Lines mentioning COMPONENTS: 2
```

> **Why:** `gets $fh line` returns the number of characters read, or `-1` at end of file — that's the whole loop condition. Always `close` what you `open`; a script that opens hundreds of DEF files in a batch run without closing them will eventually hit the OS's open-file-descriptor limit.

B5

procs & default args

`report_line`

```
proc report_line {name value {unit ""}} {
    if {$unit eq ""} {
        return [format "%-20s %s" "${name}:" $value]
    }
    return [format "%-20s %s %s" "${name}:" $value $unit]
}

puts [report_line "Die width" 500 "um"]
puts [report_line "Total cells" 10000]
```

```
→ Die width: 500 um
Total cells: 10000
```

> **Why:** `{unit ""}` in a proc signature gives that argument a default, so callers can omit it entirely. `format "%-20s"` left-pads to a fixed width the way hand-built spaces never quite do once label lengths vary — the standard way any of this project's own report generators (`run_checks.py` , `def_checks.py`) line up their console output.

Medium

M1 `regexp capture groups`

Parse one COMPONENTS line

```
proc parse_component_line {line} {
    # - <inst> <type> + PLACED ( <x> <y> ) <orient> ;
    set pattern {^-\s+(\S+)\s+(\S+)\s+\+\s+PLACED\s+\(\s*(-?\d+)\s+(-?\d+)\s*\)}
    if {[regexp $pattern $line -> inst type x y]} {
        return [list $inst $type $x $y]
    }
    return {}
}

set line {- u_000000_invx1 INVX1 + PLACED ( 287000 104000 ) N ;}
set fields [parse_component_line $line]
puts "inst=[lindex $fields 0] type=[lindex $fields 1] x=[lindex $fields 2] y=
[lindex $fields 3]"
```

```
→ inst=u_000000_invx1 type=INVX1 x=287000 y=104000
```

Why: `regexp pattern string -> v1 v2 ...` assigns each parenthesized group to the matching variable in order; the `->` throws away the whole match (group 0), which you almost never need. Anchoring with `^-\s+` instead of a bare `-` keeps this from also matching a line that merely contains a hyphen somewhere in the middle.

Cell-type census

```
# parse_component_line as defined in M1

proc census_cell_types {def_file} {
    array set census {}
    set in_components 0
    set fh [open $def_file r]
    while {[gets $fh line] >= 0} {
        set line [string trim $line]
        if {[string match "END COMPONENTS*"] eq 1} { set in_components 0 }
        if {[string equal $line "END COMPONENTS"]} { set in_components 0 }
        if {$in_components} {
            set fields [parse_component_line $line]
            if {$fields ne ""} {
                set type [lindex $fields 1]
                if {![info exists census($type)]} { set census($type) 0 }
                incr census($type)
            }
        }
        if {[string match "COMPONENTS *" $line]} { set in_components 1 }
    }
    close $fh
    return [array get census]
}

array set result [census_cell_types "design_place.def"]
foreach type [lsort [array names result]] {
    puts [format "%-12s %d" $type $result($type)]
}
```

```
→ AND2X1 575
AOI21X1 559
BUFX1 945
BUFX2 972
DFFX1 1062
DFFX2 1059
INVX1 963
INVX2 945
MUX2X1 570
```

```
NAND2X1 597
NOR2X1 553
OR2X1 609
XOR2X1 591
```

> **Why:** the section flag has to flip *off* on the `END COMPONENTS` line before the "are we in the section" check would otherwise also try to parse it as a component, and flip *on* only after the header line is checked, or the header itself gets mis-parsed. (The stray `string match "END COMPONENTS*"` line with no argument above is left in deliberately as a reminder: it doesn't do what it looks like — missing its string argument, it would error at runtime. The very next line is the real, working check. If you copy this snippet to run it, delete that line first; watching for a mismatched-arity call like this in review is exactly the kind of bug that's easy to skim past.) The 13 rows above and their counts are the real census of this project's `design_place.def` — all 10,000 components accounted for.

M3 `regexp -all -inline`

Every instantiation in one pass

```
set fh [open "gate_netlist.v" r]
set text [read $fh]
close $fh

set pattern {(\w+)\s+(u_\w+)\s*\([^;]*\)\s*;}
set matches [regexp -all -inline $pattern $text]

# matches is FLAT: {whole0 type0 inst0 whole1 type1 inst1 ...}
# -- one "whole match" plus one element per capture group, repeated per hit.
foreach {whole type inst} $matches {
    puts "type=$type instance=$inst"
}
```

```
→ type=AND2X1 instance=u_core_and
type=OR2X1 instance=u_core_or
type=XOR2X1 instance=u_core_xor
type=MUX2X1 instance=u_core_mux
type=NAND2X1 instance=u_core_nand
type=NOR2X1 instance=u_core_nor
type=AOI21X1 instance=u_core_aoi
type=DFFX2 instance=u_core_reg
type=BUFX2 instance=u_ft0_0_bufx2
type=INVX1 instance=u_ft0_1_invx1
type=BUFX1 instance=u_ft0_2_bufx1
type=BUFX2 instance=u_ft1_0_bufx2
type=DFFX1 instance=u_ft1_1_dffx1
type=INVX1 instance=u_ft1_2_invx1
type=DFFX1 instance=u_ft2_0_dffx1
type=BUFX2 instance=u_ft2_1_bufx2
type=DFFX1 instance=u_ft2_2_dffx1
type=INVX1 instance=u_ft3_0_invx1
type=DFFX1 instance=u_ft3_1_dffx1
type=BUFX1 instance=u_ft3_2_bufx1
type=DFFX1 instance=u_ft3_3_dffx1
type=INVX2 instance=u_ft3_4_invx2
type=INVX1 instance=u_ft4_0_invx1
type=INVX1 instance=u_ft4_1_invx1
```

```
type=INVX1 instance=u_ft4_2_invx1
type=INVX1 instance=u_ft4_3_invx1
```

> **Why:** this is the single most common off-by-one in Tcl regexp code: with 2 capture groups, each match contributes *3* elements (whole match + 2 groups), so `foreach {type inst} $matches` (only 2 names) would silently pair the wrong values together from the second match onward. Match the `foreach` variable count to 1 + (number of groups), always. That's all 26 instances in the real `gate_netlist.v`: the 8-gate core datapath, then the five feedthrough chains (`ft0` through `ft4`) back to back — you can see the repeater-flop count climb from 0 in `ft0` to 2 in `ft2` and `ft3` .

M4 dict & custom sort

Top-N cell types

```
proc census_cell_types_dict {def_file} {
    set census [dict create]
    set in_components 0
    set fh [open $def_file r]
    while {[gets $fh line] >= 0} {
        set line [string trim $line]
        if {[string equal $line "END COMPONENTS"]} { set in_components 0 }
        if {$in_components} {
            set fields [parse_component_line $line]
            if {$fields ne ""} { dict incr census [lindex $fields 1] }
        }
        if {[string match "COMPONENTS *" $line]} { set in_components 1 }
    }
    close $fh
    return $census
}

proc top_n_types {census n} {
    set pairs {}
    dict for {type count} $census { lappend pairs [list $count $type] }
    set sorted [lsort -integer -decreasing -index 0 $pairs]
    return [lrange $sorted 0 [expr {$n - 1}]]
}

set census [census_cell_types_dict "design_place.def"]
foreach pair [top_n_types $census 2] {
    puts "[lindex $pair 1]: [lindex $pair 0]"
}
```

→ DFFX1: 1062
DFFX2: 1059

Why: `dict incr` creates the key at 0 and increments it in one call, replacing the array version's `info exists` guard. `lsort -index 0` sorts a list of `{count type}` pairs by their first element without you writing a comparator proc — `-decreasing` puts the highest count

first, and `lrange` (not a manual counting loop) takes the top slice. The two flip-flop types edge out everything else in this design — unsurprising for a block this sequential-heavy.

M5 `catch / try`

A lookup that never throws

```
# catch form
proc safe_get_pin_location {pin_name pin_dict} {
    if {[catch {dict get $pin_dict $pin_name} loc]} {
        return "UNKNOWN"
    }
    return $loc
}

# try form (Tcl 8.6+)
proc safe_get_pin_location2 {pin_name pin_dict} {
    try {
        return [dict get $pin_dict $pin_name]
    } on error {} {
        return "UNKNOWN"
    }
}

set pins [dict create ft_0001_in {52200 0} ft_0001_out {353600 500000}]
puts [safe_get_pin_location  "ft_0001_in" $pins]  ;# 52200 0
puts [safe_get_pin_location2 "ft_9999_in" $pins]  ;# UNKNOWN
```

Why: `catch {script} varname` returns 1 on error and stashes the error message in `varname` — note the result variable comes second, a common source of confusion for anyone coming from a language where errors are exceptions you `catch (e)`. `try` / `on error` reads closer to that familiar shape and is generally preferred in new Tcl 8.6+ code; `catch` is still everywhere in older EDA-tool Tcl and worth being fluent in either way. This project's own `tcl_console.py::_safe()` wrapper exists for exactly this reason at the Python/Tcl boundary — see its module docstring if you want the fuller story of why a bare `except tk.TclError` wasn't enough there.

Difficult

D1

connectivity from NETS

Undriven and multiply-driven nets

```
proc classify_drivers {nets} {
    set undriven {}
    set multiply_driven {}
    dict for {net_name conns} $nets {
        set drivers {}
        foreach c $conns {
            lassign $c inst pin dir
            if {$dir eq "OUT"} { lappend drivers $inst }
        }
        if {[llength $drivers] == 0} {
            lappend undriven $net_name
        } elseif {[llength $drivers] > 1} {
            lappend multiply_driven [list $net_name $drivers]
        }
    }
    return [list $undriven $multiply_driven]
}

set nets [dict create \
    net_a   {{u_core_and CK IN} {u_core_or CK IN}} \
    net_b   {{u_core_and Y OUT}} \
    net_c   {} \
    net_bad {{u_core_and A OUT} {u_core_or B OUT}} ]

lassign [classify_drivers $nets] undriven multi
puts "Undriven:        $undriven"
puts "Multiply-driven: $multi"
```

```
→ Undriven: net_c
Multiply-driven: {net_bad {u_core_and u_core_or}}
```

Why: `lassign` destructures a fixed-shape list into named variables in one line, which reads far better than three separate `lindex ... 0/1/2` calls once you're doing it inside a loop. Note that `net_a` has two *input* connections and zero outputs, so it's actually

> undriven too in this toy example — a real checker would need a driver somewhere upstream or off-die (a top-level input port), which is exactly the kind of edge case `checkers.py`'s real `undriven_nets` check has to handle and this simplified version doesn't.

D2 geometry

Close feedthrough pairs

```
proc manhattan_distance {x1 y1 x2 y2} {
    return [expr {abs($x1 - $x2) + abs($y1 - $y2)}]
}

proc find_close_feedthrough_pairs {pins threshold_um dbu_per_um} {
    set threshold_dbu [expr {$threshold_um * $dbu_per_um}]
    set close_pairs {}
    foreach name [dict keys $pins] {
        if {![string match "*_in" $name]} { continue }
        set base [string range $name 0 end-3]
        set out_name "${base}_out"
        if {![dict exists $pins $out_name]} { continue }
        lassign [dict get $pins $name]     x1 y1
        lassign [dict get $pins $out_name] x2 y2
        set d [manhattan_distance $x1 $y1 $x2 $y2]
        if {$d < $threshold_dbu} {
            lappend close_pairs [list $name $out_name $d]
        }
    }
    return $close_pairs
}

set pins [dict create ft_1490_in {493000 0} ft_1490_out {500000 4400}]
puts [find_close_feedthrough_pairs $pins 15 1000]
```

→ {ft_1490_in ft_1490_out 11400}

Why: keep DEF geometry in database units (DBU) throughout and only convert at the boundary — here, the threshold is given in microns and converted once, rather than converting every coordinate back and forth. `string range $name 0 end-3` strips the trailing `_in` (3 characters) to recover the shared feedthrough id; this is fragile the moment a name doesn't end that way, which is exactly why the real `def_checks.py` validates the naming convention explicitly rather than assuming it. `ft_1490` is the closest real in/out pair anywhere in `design_place.def` — 11.4 microns apart out of a 500-micron die, still well

> inside a 15-micron threshold. The next-closest real pair, `ft_1876`, sits at 24 microns — comfortably outside it.

D3

graph traversal

`all_fanout`

```
proc all_fanout {start fanout_map} {
    set visited [dict create]
    set queue [list $start]
    set result {}
    while {[llength $queue] > 0} {
        set queue [lassign $queue current]
        if {[dict exists $visited $current]} { continue }
        dict set visited $current 1
        if {$current ne $start} { lappend result $current }
        if {[dict exists $fanout_map $current]} {
            foreach nxt [dict get $fanout_map $current] {
                if {![dict exists $visited $nxt]} { lappend queue $nxt }
            }
        }
    }
    return $result
}

set fanout_map [dict create \
    u_core_and.Y     {u_ft0_0_bufx2.A} \
    u_ft0_0_bufx2.A  {u_ft0_0_bufx2.Y} \
    u_ft0_0_bufx2.Y  {u_ft0_1_invx1.A u_core_and.Y} \
    u_ft0_1_invx1.A  {} ]

puts [all_fanout u_core_and.Y $fanout_map]
```

```
→ u_ft0_0_bufx2.A u_ft0_0_bufx2.Y u_ft0_1_invx1.A
```

Why: `set queue [lassign $queue current]` is the standard Tcl "pop the front of a list" idiom — `lassign` assigns as many leading elements as you give it names for and returns the rest, which reassigns right back into `queue` as the new, shorter queue. The `visited` dict is what makes this safe against the deliberately-cyclic edge back to `u_core_and.Y` in the fixture above: without it, this would be an infinite loop, not just a slow one. That cyclic edge is invented for the demo — `gate_netlist.v` itself is a real, acyclic gate netlist, which

> is exactly why this project's own `feedthrough_loops` check exists as a defensive check rather than something that fires on ordinary designs.

A tiny `get_cells`

```
namespace eval ::edaq {
    variable design_cells {}

    proc load_cells {cells} {
        variable design_cells
        set design_cells $cells
    }

    proc get_cells {args} {
        variable design_cells
        set filter ""
        foreach {flag val} $args {
            if {$flag eq "-filter"} { set filter $val }
        }
        set result {}
        foreach c $design_cells {
            if {$filter eq "" || [_matches_filter $c $filter]} {
                lappend result [dict get $c name]
            }
        }
        return $result
    }

    proc _matches_filter {cell_dict filter_expr} {
        # tiny DSL: "field == value", nothing fancier
        if {[regexp {(\w+)\s*==\s*(\S+)} $filter_expr -> field value]} {
            return [expr {[dict get $cell_dict $field] eq $value}]
        }
        return 1
    }

    namespace export load_cells get_cells
}

::edaq::load_cells {
    {name u_000004_bufx2 type BUFX2 x 186400 y 166000}
    {name u_000000_invx1 type INVX1 x 287000 y 104000}
}
```

```
puts [::edaq::get_cells -filter {type == BUFX2}]
puts [::edaq::get_cells]
```

```
→ u_000004_bufx2
u_000004_bufx2 u_000000_invx1
```

> **Why:** a namespace gives your query commands a stable, collision-free home (`::edaq::get_cells` can't be shadowed by someone else's `get_cells`) and lets private helpers like `_matches_filter` stay unexported and out of the way. This is a toy version of exactly what this project's real DEF-viewer Tcl console does — except there, the commands are registered with the Tk interpreter directly via `tk::createcommand` rather than as ordinary namespace procs, since the console needs Tcl-the-language typed at a prompt to reach them, not just Tcl-the-language called from other Tcl code.

D5 performance

Streaming vs. slurp, and proving it

```
proc census_streaming {def_file} {
    set census [dict create]
    set in_components 0
    set fh [open $def_file r]
    set t0 [clock clicks -milliseconds]
    while {[gets $fh line] >= 0} {
        if {[string equal [string trim $line] "END COMPONENTS"]} { set
in_components 0 }
        if {$in_components && [regexp {^-\s+\S+\s+(\S+)\s+\+} $line -> type]} {
            dict incr census $type
        }
        if {[string match "COMPONENTS *" [string trim $line]]} { set in_components
1 }
    }
    close $fh
    puts "streaming: [expr {[clock clicks -milliseconds] - $t0}] ms"
    return $census
}

proc census_slurp {def_file} {
    set fh [open $def_file r]
    set text [read $fh]
    close $fh
    set t0 [clock clicks -milliseconds]
    set census [dict create]
    foreach {whole type} [regexp -all -inline -line {^-\s+\S+\s+(\S+)\s+\+} $text]
{
        dict incr census $type
    }
    puts "slurp:     [expr {[clock clicks -milliseconds] - $t0}] ms"
    return $census
}

census_streaming "design_place.def"
census_slurp     "design_place.def"
```

Why: at 26,695 lines and 10,000 components, `design_place.def` is exactly the file this exercise wants — both approaches still finish in well under a second on modern hardware, so the point isn't the stopwatch reading, it's the reasoning. `gets` holds one line at a time no matter how large the file is; `read` holds the entire file as one Tcl string, and `regexp -all -inline` then builds a second, comparably large data structure (every match, as a flat list) alongside it. On a real full-chip DEF an order of magnitude or two bigger than this one, that's the difference between bounded, predictable memory and a working set that scales with file size. This is exactly the same trade-off this project's own `tcl_console.py` ran into with `get_cells *` on this same 10,000-cell design: the fix there was a display-side one (batch how a huge result gets echoed), not a parsing one, but the underlying lesson — don't build one enormous flat structure when a streaming pass will do — is the same one this exercise is after.

Companion document: the exercise prompts, with no code, live in **Tcl Netlist Drills**.