Design Configuration Files

Unless the design uses the API directly, each OpenLane-compatible design must come with a configuration file. These configuration files can be written in one of two grammars: JSON or Tcl.

Tcl offers more flexibility at the detriment of security, while JSON is more straightforward at the cost of flexibility. While Tcl allows you to do all manner of computation on your variables, JSON has a limited expression engine that will be detailed later in this document. Nevertheless, for security (and future-proofing), we recommend you use either the JSON format or write Python scripts using the API.

The folder containing your `config.tcl` / `config.json` is known as the **Design Directory** – though the design directory can also be set explicitly over the command-line using `--design-dir`. The design directory is special in that paths in the JSON configuration files can be resolved relative to this directory and that in TCL configuration files, it can be referenced via the environment variable `DESIGN_DIR`. This will be explained in detail in later sections.

> **Note**
>
> When using the API, you can provide the inputs directly as a Python dictionary, enabling you to do complex pre-processing far beyond the capacity of either JSON or Tcl files. You can still use `ref::` and such like JSON files though.

JSON

The JSON files are simple key-value pairs.

The values can be scalars (strings, numbers, Booleans, and `null`s), lists or dictionaries, subject to validation.

All files must be ECMA404-compliant, i.e., pure JSON with no extensions such as comments or the new elements introduced in JSON5.

An minimal demonstrative configuration file would look as follows:

```
{
  "DESIGN_NAME": "spm",
  "VERILOG_FILES": "dir::src/*.v",
  "CLOCK_PORT": "clk",
  "CLOCK_PERIOD": 100,
  "pdk::sky130A": {
    "MAX_FANOUT_CONSTRAINT": 6,
    "FP_CORE_UTIL": 40,
    "PL_TARGET_DENSITY_PCT": "expr::($FP_CORE_UTIL + 10.0)",
    "scl::sky130_fd_sc_hd": {
      "CLOCK_PERIOD": 15
    }
  }
}
```

Pre-processing

The JSON files are pre-processed at runtime. Features include conditional execution, a way to reference the design directory, other variables, and a basic numeric expression engine.

Conditional Execution

The JSON configuration files support conditional execution based on PDK or standard cell library (or, by nesting as shown above, a combination thereof.) You can do this using the `pdk::` or `scl::` key prefixes.

The value for this key would be a `dict` that is only evaluated if the PDK or SCL matches those in the key, i.e., for `pdk::sky130A` as shown above, this particular `dict` will be evaluated and its values used if and only if the PDK is set to `sky130A`, meanwhile with say, `asap7`, it will not be evaluated.

The match is evaluated using `fnmatch`, giving it limited wildcard support: meaning that `pdk::sky130*` would match both `sky130A` and `sky130B`.

Note that ***the order of declarations matter here***: as seen in the following example, despite a more specific value for a PDK existing, the unconditionally declared value later in the code would end up overwriting it:

```
{
    "pdk::sky130A": {
        "A": 40
    },
    "A": 4
}

{
    "A": 4,
    "pdk::sky130A": {
        "A": 40
    }
}
```

> In the first example, the final value for A would always be 4 given the order of declarations. In the second example, it would be 40 is the PDK is sky130A and 4 otherwise.

It is worth nothing that the final resolved configuration would have the symbol in the parent object with no trace left of the conditionally-executed, dict i.e., the second example with the sky130A PDK simply becomes:

```
{
  "A": 40
}
```

Variable Reference

If a string's value starts with `ref::`, you can interpolate exactly one **string** variable at the beginning of your string.

Like conditional execution, the order of declarations matter: i.e., you cannot reference a variable that is declared after the current expression.

```
{
    "A": "ref::$B",
    "B": "vdd gnd"
}

{
    "B": "vdd gnd",
    "A": "ref::$B"
}
```

> In this example, the first configuration is invalid, as B is referenced before it is declared, but the latter is OK, where the value will be “vdd gnd” as well.

Do note that unlike Tcl config files, environment variables (other than `DESIGN_DIR`, `PDK`, `PDKPATH`, `STD_CELL_LIBRARY`) are not exposed to `config.json` by default.

If the files you choose lie **inside** the design directory, a different prefix, `refg::`, supports non-recursive globs, i.e., you can use an asterisk as a wildcard to pick multiple files in a specific folder.

- Outside the design directory, this is disabled for security reasons and the final path will continue to include the asterisk.
- `refg::` will always return an array, even if only one element was found, for consistency.
  - If no elements were found, the glob string is returned verbatim as a single element in array.

As shown below, `refg::$DESIGN_DIR/src/*.v` would find all files ending with `.v` in the `src` folder inside the design directory.

```
{
  "VERILOG_FILES": "refg::$DESIGN_DIR/src/*.v"
}
```

There are some shorthands for the exposed default variables:

- `dir::` is equivalent to `refg::$DESIGN_DIR/`
- `pdk_dir::` is equivalent to `refg::$PDK_ROOT/$PDK`

Expression Engine

By adding `expr::` to the beginning of a string, you can write basic infix mathematical expressions. Binary operators supported are `**`, `*`, `/`, `+`, and `-`, while operands can be any floating-point value, and previously evaluated numeric variables prefixed with a dollar sign. Unary operators are not supported, though negative numbers with the - sign stuck to them are. Parentheses (`()`) are also supported to prioritize certain operations.

Your expressions must return exactly one value: multiple expressions in the same `expr::`-prefixed value are considered invalid and so are empty expressions.

It is important to note that, like variable referencing and conditional execution, the order of declarations matter: i.e., you cannot reference a variable that is declared after the current expression.

```
{
    "A": "expr::$B * 2",
    "B": 4
}

{
    "B": 4,
    "A": "expr::$B * 2"
}
```

> In this example, the first configuration is invalid, as B is used in a mathematical expression before declaration, but the latter is OK, evaluating to 8.

You can also simply reference another number using this prefix:

```
{
  "A": 10,
  "B": "expr::$A"
}
```

> In this example, B will simply hold the value of A.

Tcl

These configuration files are simple Tcl scripts with environment variables that are sourced by the OpenLane flow. Again, Tcl config files are not recommended for newer designs, but is still maintained and supported at the moment.

Each design using the Tcl format has a global config.tcl and other config files one for each PDK:

```
designs/<design_name>
├── config.tcl
├── sky130A_sky130_fd_sc_hs_config.tcl
├── sky130A_sky130_fd_sc_hd_config.tcl
├── src
│   ├── design.v
```

You can see `designs/xtea` for an example of a design that still uses the Tcl format.

To support the technology-specific config files, the global `config.tcl` files should end with these lines:

```
set filename $::env(DESIGN_DIR)/$::env(PDK)_$::env(STD_CELL_LIBRARY)_config.tcl
if { [file exists $filename] == 1} {
        source $filename
}
```

This implies that if the `{PDK}_{STD_CELL_LIBRARY}_config.tcl` doesn't exist for a specific technology combination the flow would resume normally with only the global config.tcl.

This structure allows for storing the best configurations for a given design on all different PDKs and their STD_CELL_LIBRARYs. The best configuration for a given design differ from one PDK and STD_CELL_LIBRARY to another.