Using Macros

The majority of chip designs are not actually synthesized, placed, routed and streamed out all in one flow. Rather, parts of the design are usually hardened first, which are then **integrated** into a **top-level chip**. These pre-hardened parts of a design are called "macros."

There are a number of advantages to using Macros in PnR, including but not limited to:

- **Faster PnR**: If a macro has extremely complex routing, i.e. a latch-based SRAM for example, you can save the time of PnR on that module when you're iterating on your own design's PnR process.
- **Reuse**: Some macros are commonly reused between multiple designs, so the effort of hardening them can be skipped after the first time.
- **Black-boxing**: You can use macros by other designers as a black-box; i.e; you don't know and don't care how they're implemented. Macros are typically **verified**, i.e., have been tested in other designs or extensively simulated to assure functionality.

  For proprietary (closed-source) designs, this has an added benefit of being able to license macros to other designers as intellectual property (IP) without disclosing the register-transfer-level code beyond a simulation model.

All non-trivial designs taped out using OpenLane have involved the use of macros of some kind.

Macro Files

There are a number of views you typically need from a Macro. The two essential views for OpenLane are the LEF (`.lef`) and the GDSII (`.gds`) views- the former of

latest

which is used in PnR and the latter is used for tape-out.

- Library Exchange Format (LEF/`.lef`): Essential
  - The "interface" of a Macro; indicating things such as its dimensions, locations of the pins, other structures on the metal layers (that may obstruct routing,) et cetera.
  - Used during PnR.
- Graphic Design System II Format (GDSII/`.gds`): Essential
  - Contains all of the information from the LEF and also other information that PnR tools are not concerned with, such as the patterns on the actual silicon layers (vs. just the metal layer.)
  - Used during stream-out.
- The Gate-Level Netlist (`.gl.v`/`.nl.v`): Optional
  - May be used during STA (see relevant section).
  - Used as a fallback during hierarchy checks and linting if neither Verilog headers (`.vh`) nor powered gate-level netlists (`.pnl.v`) exist. It is not recommended for this use as the linting may fail.
  - Used as a fallback during synthesis if Verilog headers (`.vh`) do not exist.
- The Powered Gate-Level Netlist (`.pnl.v`): Optional
  - Same as the netlist, but also has the power pins connected
  - Used as a fallback during hierarchy checks and linting if Verilog headers (`.vh`) do not exist.
  - Used as a fallback during synthesis if neither Verilog headers nor regular netlists (`.gl.v`/`.nl.v`) exist. It is not recommended for this use as synthesis checks may fail.
- Lib file (`.lib`): Optional
  - May be used during STA (see relevant section).
  - Used as a last resort for synthesis if no Verilog header (`.vh`) or any netlists (`.nl.v`/`.gl.v`/`.pnl.v`) are available. It is not recommended for this use as synthesis checks may fail.
- SPEF (`.spef`): Optional
  - may be used during STA (see relevant section).
- SPICE (`.spice`): Optional
  - currently unused, but may be used in the futur Read the Docs latest
    OpenLane produces a SPICE view of all macros, it is prudent to include it anyway.

- SDF (`.sdf`): Optional
  - currently unused. As OpenLane produces SDF views of all macros, it is prudent to include it anyway.
- Yosys JSON Header (`.json`): Optional
  - A JSON netlist produced by Yosys that provides the Powered Netlist in a Python-parseable format.

In OpenLane Configurations

Macros are declared in OpenLane configurations in the global variable `MACROS`. Macros is a dictionary, which in JSON is going to be just a dictionary/hashmap, where the keys are the name of the macro (not instances thereof) and the values are a Python dataclass. You can find the API reference for the macros hashmap at `openlane.common.Macro`, but a less mechanical explanation is as follows:

- The keys contain the name of the Macro itself (not instances thereof.)
- The values are:
  - A dictionary of instance names to instance objects
    - The instance objects in turn consist off:
      - `location`: A tuple of two numbers, in microns, indicating the location of the macro (optional)
      - `orientation`: The orientation of the placed macro– see page 250 of the LEFDEFREF for a definition and visual.
  - `gds`: List of GDS files comprising the macro (usually only one)
  - `lef`: List of LEF files comprising the macro (usually only one)
  - `vh`: List of Verilog Headers (if any)
  - `nl`: List of Gate-level Netlists (if any)
  - `pnl`: List of Powered Gate-level Netlists (if any)
  - `spice`: List of SPICE files (if any)
  - `lib`: Dictionary of LIB files.
    - The keys of the dictionary are wildcards matching timing corners, and the values are paths for lib files that are characterized for those timing corners. For example:
    - `"*_tt_025C_1v80": ["dir::lib0.lib"]` will match `lib0.lib` to all of `min_tt_025C_1v80`, `max_tt_025C_1v80`, `and nom_tt_025C_1v80`.
  - `spef`: Dictionary of SPEF files, similar to above.
  - `sdf`: Dictionary of SDF files, similar to above.
  - `json_h`: (Optional) path to a singular Yosys-style JSON header for the design

> **Warning**
>
> The Macro class will throw an error if either `gds` or `lef` do not have at least one file.

Let's presume the following Macro as an example, a Macro named `aes_example` with one instance named `mprj` placed in the north orientation at $(10\mu m, 20\mu m)$ would be declared as:

```
    "aes_example": {
        "instances": {
            "mprj": {
                "location": [
                    10,
                    20
                ],
                "orientation": "N"
            }
        },
        "gds": [
            "dir::./gds/aes_example.gds.gz"
        ],
        "lef": [
            "dir::./lef/aes_example.lef"
        ],
        "nl": [
            "dir::./gl/aes_example.v"
        ],
        "spef": {
            "min_*": [
                "dir::./spef/multicorner/aes_example.min.spef.gz"
            ],
            "nom_*": [
                "dir::./spef/multicorner/aes_example.nom.spef.gz"
            ],
            "max_*": [
                "dir::./spef/multicorner/aes_example.max.spef.gz"
            ]
        },
        "lib": {},
        "spice": [],
        "sdf": {}
    }
```

On Instance Names

Instance names should match the name as instantiated in Verilog, i.e., without escaping any characters for layout formats.

For example, if you instantiate an array of macros `spm spm_inst[1:0]`, the first instance's name should be `spm_inst[0]`, not `spm_inst\[0\]` or similar.

latest

STA

Without certain files, STA is performed in what is known as **black-boxed** mode. This has a tendency to hide what are known as “errors at the boundary,” where if STA passes cleanly for the macro individually and for the top-level integration as a black-box; STA still fails for the overall design.

For example, let’s presume a situation where a top-level chip A integrates a macro B. Macro B passes STA successfully, and Top-level A also passes STA successfully, albeit with B as a black-box. However:

- Macro B undergoes static timing analysis with an input delay of 5ns for all pins
- The actual delay of one of B’s inputs connected to a register in A has an actual input delay of 2ns.

This means that the value of the signal connected to B may in fact be changing too quickly, causing a hold violation at the boundary that would not be detectable without hierarchical STA.

Depending on the value of `STA_MACRO_PRIORITIZE_NL`; these files could be used:

- If `true`, a combination of `.nl.v` and `.spef` are prioritized (if available,) and if either is not available, `.lib` is used as a fallback, and if that doesn’t exist, STA is done in black-boxed mode.
- If `false`, `.lib` is used, and if it is not available, `.nl.v` and `.spef` are used as a fallback. If neither are available, STA is done in black-boxed mode.

In commercial flows, typically, just `.lib` is used. However, the lack of an accurate open source `.lib` file generator and OpenSTA producing very accurate timing results with `.nl.v` and `.spef` leads us to prioritize these by default.

However, if you’re using a macro that has been hardened using proprietary tools, you may want to use the `.lib` file.

Regrettably, `STA_MACRO_PRIORITIZE_NL` is universal, i.e., either you prioritize `.lib` for all macros or prioritize `.nl.v` + `.spef` for all macros. You may however say, exclude `.spef` files from the Macro definition so it w latest
`.lib` for a certain design, and so on.

Working with Macros in your RTL

In your RTL, Macros must be instantiated as follows, where the `USE_POWER_PINS` preprocessor guard corresponds to the value of the variable `VERILOG_POWER_DEFINE`.

```
macro_name instance_name(
`ifdef USE_POWER_PINS
  .power_pin1(power_net1),
  .ground_pin1(ground_net1),
`endif
  .signal_pin1(signal_net1),
  .signal_pin2(signal_net2)
);
```

This is used by flows to create a hierarchy of power nets for the design, which is later used in constructing the power distribution network and connecting macros together.

Deciding on Macros *ex post facto*

Sometimes, as a PnR engineer, you're tasked with hardening a giant RTL design where some components that should logically be macros aren't.

In that scenario, you typically exclude the RTL from synthesis and replace it with a Macro's netlist that was hardened separately. The problem with that is, some of these submodules that you want to turn to Macro may have Verilog parameters.

For example, let's assume you want to harden this submodule as a Macro:

```verilog
// BarrelShifter.v
module BarrelShifter #(
  parameter width = 32
) (
  VPWR,
  VGND,
  A,
  O,
  shmt
);
// ...
endmodule

// Top.v
module Top(
  // ...
);
BarrelShifter #(
  width=16
) barrelShifter(
`ifdef USE_POWER_PINS
  .VPWR(VPWR),
  .VGND(VGND),
`endif
  .A(in),
  .O(out)
  .shamt(shamt)
);
endmodule
```

As part of the hardening process, the width parameter becomes “baked-in”; i.e. it can no longer be changed, and `BarrelShifter.nl.v` will have a baked in width value of `16`. However, this means that during linting and synthesis, as they use the Verilog Header or Netlist, the Synthesizer will say that “width” is an unknown parameter.

You may think the solution is simply to then remove the parameter. The problem is during RTL simulation where you use `BarrelShifter.v`, that will end up using THAT default value, which is `32`, breaking the simulation process.

One solution is as follows:

latest

```
module Top(
  // ...
);

`ifdef __pnr__
BarrelShifter barrelShifter (
`ifdef USE_POWER_PINS
  .VPWR(VPWR),
  .VGND(VGND),
`endif
  .A(in),
  .O(out)
  .shamt(shamt)
);
`else
BarrelShifter #(
  width=16
) barrelShifter(
  .A(in),
  .O(out)
  .shamt(shamt)
);
`endif
endmodule
```

This will preserve the original RTL for simulation, but when Synthesizing/ Linting using OpenLane, `__pnr__` will be defined, thereby using the header for the hardened version of the Macro. Additionally, as the power pins have no relevance for the RTL, they can simply be left out when PnR is not defined.

latest