Hammer

Release 1.0.0

Berkeley Architecture Research

Mar 07, 2024

CONTENTS:

Hammer is a physical design framework that wraps around vendor specific technologies and tools to provide a single API to create ASICs. Hammer allows for reusability in ASIC design while still providing the designers leeway to make their own modifications.

Hammer (Highly Agile Masks Made Effortlessly from RTL) is a framework for building physical design generators for digital VLSI flows. It is an evolving set of APIs that enable reuse in an effort to speed up VLSI flows, which have traditionally been entirely rebuilt for different projects, technologies, and tools.

Hammer is able to generate scripts and collateral for a growing range of CAD tools while remaining technology-agnostic using a well-defined set of common APIs. Tool- and technology-specific concerns live inside plugins, implement APIs, and provide a set of working default configurations.

The vision of Hammer is to reduce the cycle time on VLSI designs, enabling rapid RTL design space exploration and allowing a designer to investigate the impact of various parameters like timing constraints and floorplans without needing to worry about low-level details.

For high-level details about Hammer's design principles and capabilities, please refer to our DAC 2022 paper entitled Hammer: A Modular and Reusable Physical Design Flow Tool. We kindly request that this paper be cited in any publications where Hammer was used.

CHAPTER

ONE

HAMMER BASICS

This documentation will give an overview of Hammer, its basic setup, its components, and its structure, as well as some typical project setup.

1.1 Hammer Overview

Hammer has a set of actions and automatically takes the output of one action and converts it into the input for another. For instance, a synthesis action will output a mapped verilog file which will then automatically by piped to the place-and-route input when a place-and-route action is called.

A user's Hammer environment is typically separated into four different components: core Hammer, one or more tool plugins, a technology plugin, and a set of project-specific Hammer input files. Hammer is meant to expose a set of generalized APIs that are then implemented by tool- and technology-specific plugins.

Hammer is included in a larger project called Chipyard which is the unified repo for an entire RTL, simulation, emulation, and VLSI flow from Berkeley Architecture Research. There is an in-depth Hammer demo there, and it is a great place to look at a typical Hammer setup.

1.1.1 Main Hammer

Hammer provides the Python backend for a Hammer project and exposes a set of APIs that are typical of modern VLSI flows. These APIs are then implemented by a tool plugin and a technology plugin of the designer's choice. The structure of Hammer is meant to enable re-use and portability between technologies.

Hammer takes its inputs and serializes its state in the form of YAML and JSON files, called Hammer IR. The designer sets a variety of settings in the form of keys in different namespaces that are designated in Hammer to control its functionality.

Note: Supported keys are contained in *defaults.yml <https://github.com/ucb-bar/hammer/blob/master/hammer/config/defaults.yml>*__. This file provides sensible defaults that may be overridden or are set to null if they must be provided by the designer.

Here is an example of a snippet that would be included in the user's input configuration.

```
vlsi.core.technology: "hammer.technology.asap7"
vlsi.inputs.supplies:
    VDD: "0.7 V"
    GND: "0 V"
```

This demonstrates two different namespaces, `vlsi.core` and `vlsi.inputs`, and then two different keys, `technology` and `supplies`, which are set to the `asap7` technology and 0.7 Volts supply voltage, respectively.

Further details about these keys and how they are manipulated is found in the *Hammer IR and Meta Variables* section.

1.1.2 Tech Plugins

A techonology plugin consists of two or more files: a `*.tech.json` and a `defaults.yml`.

The `*.tech.json` contains pointers to relevant PDK files and fundamental technology constants. These values are not meant to be overriden, nor can they be for the time being.

`defaults.yml` sets default technology variables for Hammer to consume, which may be specific to this technology or generic to all. These values may be overriden by design-specific configurations. An example of this is shown in the open-source technology plugins in `hammer/technology/`, such as `asap7`, and how to setup a technology plugin is documented in more detail in the *Hammer Technology Plugins* section.

Note: Unless you are a UCB BAR or BWRC affiliate or have set up a 3-way technology NDA with us, we cannot share pre-built proprietary technology plugin repositories.

1.1.3 Tool Plugins

A Hammer tool plugin actually implements tool-specific steps of the VLSI flow in Hammer in a template-like fashion. The TCL commands input to the tool are created using technology and design settings provided by the designer.

There are currently three Hammer tool plugin repositories for commercial tools: `hammer-cadence-plugins`, `hammer-synopsys-plugins`, and `hammer-mentor-plugins`. In them are tool plugin implementations for actions including synthesis, place-and-route, DRC, LVS, and simulation. `hammer-cadence-plugins` and `hammer-synopsys-plugins` are publicly available; however, users must request access to `hammer-mentor-plugins`.

Note: If you are not a UCB BAR or BWRC affiliate and have access to Mentor Graphics (now Siemens) tools, please email hammer-plugins-access@lists.berkeley.edu with a request for access to the hammer-mentor-plugins repository. MAKE SURE TO INCLUDE YOUR GITHUB ID IN YOUR EMAIL AND YOUR ASSOCIATION TO SHOW YOU HAVE LICENSED ACCESS TO THE TOOLS. There will be no support guarantee for the plugin repositories, but users are encouraged to file issues and contribute patches where needed.

There are also a set of open-source tools (e.g. Yosys, OpenROAD, Magic, Netgen) provided in `hammer/` under their respective actions.

These plugins implement many of the common steps of a modern physical design flow. However, a real chip flow will require many custom settings and steps that may not be generalizable across technology nodes. Because of this, Hammer has an "escape-hatch" mechanism, called a hook, that allows the designer to inject custom steps between the default steps provided by the CAD tool plugin. Hooks are python methods that emit TCL code and may be inserted before or after an existing step or replace the step entirely. This allows the designer to leverage the APIs built into Hammer while easily inserting custom steps into the flow. Hooks are discussed in more detail in the "Example usage" portion of the Hammer documentation.

1.1.4 Calling Hammer

To use Hammer on the command line, the designer will invoke the `hammer-vlsi` utility. This is calling the `__main__()` method of the `CLIDriver` class. An example invocation is below:

```
hammer-vlsi -e env.yml -p config.yml --obj_dir build par
```

Using hooks requires the designer to extend the `CLIDriver` class. A good example exists in the Chipyard repository (`chipyard/vlsi/example-vlsi`). This would change the invocation to something like the following:

```
example-vlsi -e env.yml -p config.yml --obj_dir build par
```

Hammer configuration files must be in YML or JSON format, and are divided into environment and project configurations. The environment configuration file, which in this case is `env.yml`, is passed to Hammer using the `-e` flag. `env.yml` contains pointers to the required tool licenses and environment variables. The project configuration file is passed in using the `-p` flag. In this case, there is only one, `config.yml`, and it needs to set all the required keys for the step of the flow being run. Passing in multiple files looks like `-p config1.yml -p config2.yml`. Refer to the *Hammer IR and Meta Variables* section for the implications of multiple config files. The environment settings take precedence over all project configurations, and are not propagated to the output configuration files after each action. The order of precedence for the project configs reads from right to left (i.e. each file overrides all files to its left in the command line).

`--obj_dir build` designates what directory Hammer should use as a working directory. All default action run directories and output files will be placed here.

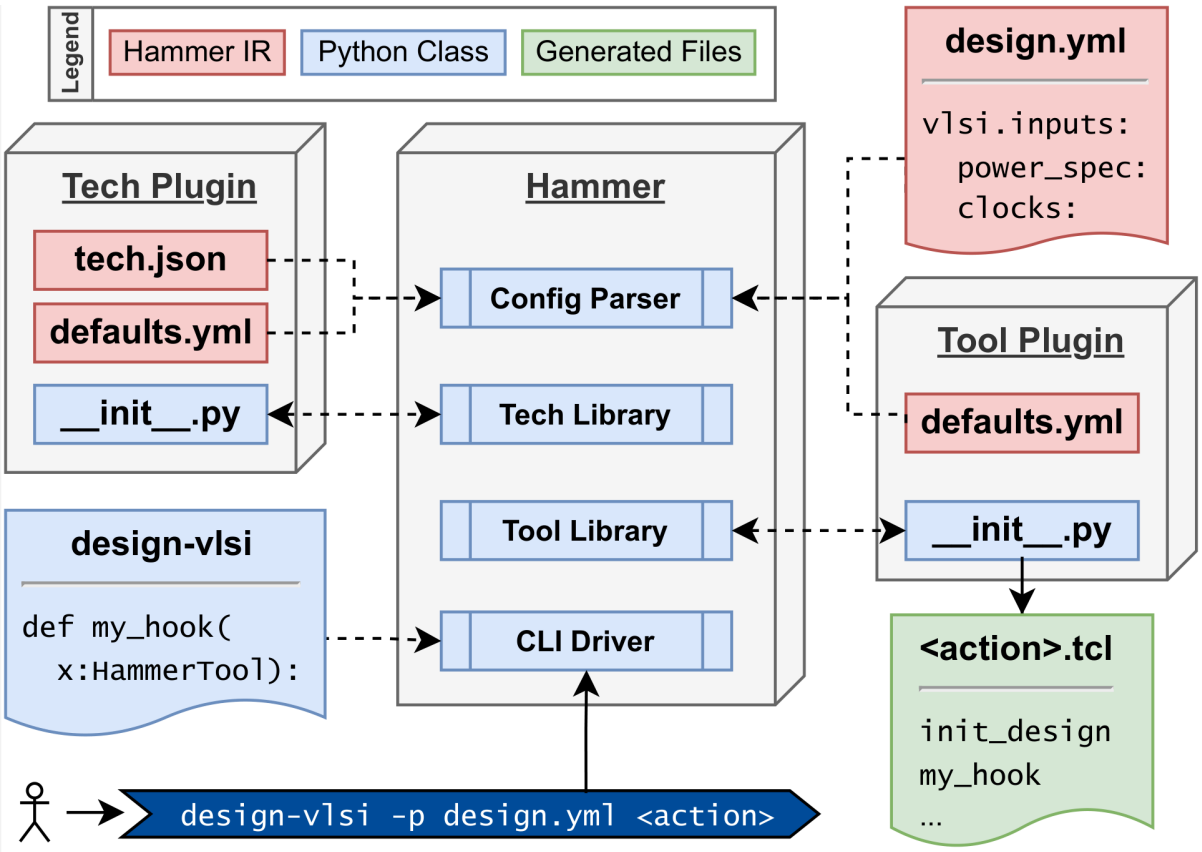
Finally, `par` designates that this is a place-and-route action.

In this case, Hammer will write outputs to the path `$PWD/build/par-rundir`.

For the full list of Hammer command-line arguments, run *hammer-vlsi –help* or take a peek in the `hammer/vlsi/cli_driver.py` file.

1.1.5 Summary

The software architecture as described above is shown in the diagram below, which is taken from the Hammer DAC paper.



1.2 Hammer Setup

Hammer depends on Python 3.9+.

The default technology, ASAP7, has some extra requirements. See its README for instructions.

1.2.1 User Setup

You can install Hammer from PyPI:

```
pip install hammer-vlsi
```

If you are using ASAP7, you need to install hammer-vlsi with the asap7 extra dependency (gdspy or gdstk). By default, gdspy is installed:

```
pip install hammer-vlsi[asap7]
```

If instead, you want to install gdstk:

```
pip install hammer-vlsi[asap7-gdstk]
```

After installation, verify that you can run the `hammer-vlsi` script from the command line.

```
hammer-vlsi -h
```

Note: certain tools and technologies will have additional system requirements. For example, LVS with Netgen requires Tcl/Tk 8.6, which is not installed for CentOS7/RHEL7 and below. Refer to each respective tool and technology's documentation for those requirements.

Installing Hammer as a Source Dependency

In some cases, it is useful to install Hammer as a source dependency. For instance, when developing tool or PDK plugins alongside a new feature or API changes in main Hammer, installing hammer as a source dependency will allow you to make changes in main hammer and see them reflected immediately when running code for your tool/PDK plugin.

From Another Poetry Project

Hammer tool (e.g. `hammer-cadence-plugins`) and PDK plugin repositories are poetry projects (with a `pyproject.toml` in their root). To depend on Hammer as a source dependency, first clone Hammer somewhere on your disk. Then add this snippet to the tool/PDK plugin repo's `pyproject.toml` (and remove any PyPI dependency on Hammer):

```
[tool.poetry.dependencies]
#hammer-vlsi = "^1.0.0"
hammer-vlsi = {path = "path/to/hammer", extras = ["asap7"], develop = true}
```

Run `poetry update` and `poetry install`. Do not commit the changes to `pyproject.toml` or `poetry.lock` without first removing the source dependency. You only need to specify `extras` if you need the `asap7` optional dependency (gdstk).

From a Generic Python Project

Other repos, such as Chipyard, are not poetry projects, but still depend on Hammer. To use Hammer as a source dependency:

1. Remove the PyPI hammer-vlsi dependency from the project (e.g. by editing a conda env.yml file and rerunning dependency resolution)
2. Clone Hammer somewhere on your disk
3. Activate the virtualenv of the project (e.g. Chipyard)
4. Run `pip install -e .` from the root of Hammer *within the project's virtualenv*

1.2.2 Developer Setup

1. Clone Hammer with `git`

```
git clone git@github.com:ucb-bar/hammer
cd hammer
```

2. Install poetry to manage the development virtualenv and dependencies

```
curl -sSL https://install.python-poetry.org | python3 -
```

3. Create a poetry-managed virtualenv using the dependencies from `pyproject.toml`

```
# create the virtualenv inside the project folder (in .venv)
poetry config virtualenvs.in-project true
poetry install
```

4. Activate the virtualenv. Within the virtualenv, Hammer is installed and you can access its scripts defined in `pyproject.toml` (in `[tool.poetry.scripts]`)

```
poetry shell
hammer-vlsi -h
```

Using PyCharm

This project works out of the box with PyCharm. You should install the Pydantic plugin to enable autocomplete for Pydantic BaseModels.

Unit Tests with pytest

Within the poetry virtualenv, from the root of Hammer, run the tests (`-v` will print out each test name explicitly)

```
pytest tests/ -v
```

If you want to skip the single long running-test in `test_stackup.py`:

```
pytest tests/ -m "not long" -v
```

If you want to run only a specific test use `-k` with a snippet of the test function you want to run:

```
pytest tests/ -k "lsf" -v

> tests/test_submit_command.py::TestSubmitCommand::test_lsf_submit[lsf] PASSED
```

By default, `pytest` will only display what a test prints to stdout if the test fails. To display stdout even for a passing test, use `-rA`:

```
pytest tests/test_build_systems.py -k "flat_makefile" -rA -v

> ____________________ TestHammerBuildSystems.test_flat_makefile ____________________
> ----------------------------- Captured stdout call -----------------------------
> [<global>] Loading hammer-vlsi libraries and reading settings
> [<global>] Loading technology 'nop'
> =========================== short test summary info ============================
> PASSED tests/test_build_systems.py::TestHammerBuildSystems::test_flat_makefile
```

Type Checking with mypy

There is a small issue with the ruamel.yaml package typechecking which can be hacked around with (replace the python version with your own):

```
touch .venv/lib/python3.10/site-packages/ruamel/py.typed
touch .venv/lib/python3.10/site-packages/networkx/py.typed
```

Inside your poetry virtualenv, from the root of Hammer, run:

```
mypy --namespace-packages --warn-unused-ignores -p hammer

Success: no issues found in 146 source files

mypy --namespace-packages --warn-unused-ignores tests

Success: no issues found in 25 source files
```

Testing Different Python Versions with tox

Hammer is supposed to work with Python 3.9+, so we run its unit tests on all supported Python versions using `tox` and `pyenv`.

1. Install pyenv

```
curl https://pyenv.run | bash
```

Restart your shell and run `pyenv init` (and follow any of its instructions). Then restart your shell again.

2. Install Python versions

See the `.python-version` file at the root of hammer and install those Python versions using `pyenv`.

```
pyenv install 3.9.13
pyenv install 3.10.6
```

Once the Python interpreters are installed, run `pyenv versions` from the root of hammer.

```
pyenv versions
  system
* 3.9.13 (set by .../hammer/.python-version)
* 3.10.6 (set by .../hammer/.python-version)
```

3. From within your `poetry` virtualenv, run `tox`

```
tox
```

This will run the pytest unit tests using all the Python versions specified in `pyproject.toml` under the `[tool.tox]` key.

You can run tests only on a particular environment with `-e`

```
tox -e py39 # only run tests on Python 3.9
```

You can pass command line arguments to the pytest invocation within a tox virtualenv with `--`

```
tox -e py39 -- -k "lsf" -v
```

Adding / Updating Dependencies

To add a new Python (`pip`) dependency, modify `pyproject.toml`. If the dependency is only used for development, add it under the key `[tool.poetry.dev-dependencies]`, otherwise add it under the key `[tool.poetry.dependencies]`. Then run `poetry update` and `poetry install`. The updated `poetry.lock` file should be committed to Hammer.

To update an existing dependency, modify `pyproject.toml` with the new version constraint. Run `poetry update` and `poetry install` and commit `poetry.lock`.

Building Documentation

First, generate the `schema.json` file from within your poetry virtualenv:

```
python3 -c "from hammer.tech import TechJSON; print(TechJSON.schema_json(indent=2))" >␣
↪doc/Technology/schema.json
```

Then:

- `cd doc`
- Modify any documentation files. You can migrate any rst file to Markdown if desired.
- Run `sphinx-build . build`
- The generated HTML files are placed in `build/`
- Open them in your browser `firefox build/index.html`

Publishing

Build a sdist and wheel (results are in `dist`):

```
poetry build
```

To publish on TestPyPI:

1. Create an account on TestPyPi
2. Note the source repository `testpypi` in `pyproject.toml` under the key `[tool.poetry.source]`
    - To add another PyPI repository to poetry: `poetry source add <source name> <source url>`
3. Publish: `poetry publish --repository testpypi -u <username> -p <password>`

1.3 Migration Guide

Hammer's software infrastructure changed significantly for version 1.0.0. In order to use the latest version of Hammer, you will need to make changes to your existing tool/tech plugins, YML/JSON input files, and Python files that import Hammer.

This guide is relevant for both plugin developers and general users.

The documentation for old Hammer is cached here.

1.3.1 Selecting Plugins

Previously, tech and tool plugins were selected with a combination of tool name and relative plugin path:

```
vlsi.core.technology: <tech_name>
vlsi.core.technology_path: ["hammer-<tech_name>-plugin"]
vlsi.core.technology_path_meta: append

vlsi.core.synthesis_tool: <syn_tool_name>
vlsi.core.synthesis_tool_path: ["hammer-<vendor>-plugin/syn"]
vlsi.core.synthesis_tool_meta: append
```

Now, you simply need to specify the package name:

```
vlsi.core.technology: hammer.technology.<tech_name>

vlsi.core.synthesis_tool: hammer.synthesis.<syn_tool_name>
```

The package name is defined by the file structure of the plugin package. See the *Plugin File Structure* section below for details.

1.3.2 `import` Statements

When importing Hammer classes and/or methods, replace old statements:

```
import hammer_tech
from hammer_vlsi import HammerTool
```

with:

```
import hammer.tech
from hammer.vlsi import HammerTool
```

The rule of thumb is that underscores are replaced with periods. This will match the package directory structure under the `hammer/` directory.

1.3.3 Technology JSON

Previously, the technology JSON file may have contained entries like this:

```
"gds map file": "path/to/layermap.file"
```

Now, keys must not contain spaces in line with JSON syntax:

```
"gds_map_file": "path/to/layermap.file"
```

The fields for `installs` and `tarballs` have also changed. Generally, `path` is now `id` and `base var` is now `path` to remove confusion. For example, previously (ASAP7 example for reference):

```
"installs": [
  {
    "path": "$PDK",
    "base var": "technology.asap7.pdk_install_dir"
  }
],
"tarballs": [
  {
    "path": "ASAP7_PDK_CalibreDeck.tar",
    "homepage": "http://asap.asu.edu/asap/",
    "base var": "technology.asap7.tarball_dir"
  }
]
```

is now:

```
"installs": [
  {
    "id": "$PDK",
    "path": "technology.asap7.pdk_install_dir"
  }
],
"tarballs": [
  {
    "root": {
      "id": "ASAP7_PDK_CalibreDeck.tar",
      "path": "technology.asap7.tarball_dir"
    },
    "homepage": "http://asap.asu.edu/asap/",
    "optional": true
  }
]
```

1.3.4 Plugin File Structure

Plugins previously did not have a file structure requirement. For example, it could have looked like this:

```
mytech/
    __init__.py
    defaults.yml
    mytech.tech.json
    layer.map         # some tech-specific file
action/
    action_tool/
        __init__.py   # contains action_tool-specific hooks for mytech technology
        tool.options  # some tech-specific file needed by this tool
```

The new structure must follow Python package convention, hence it will look as follows:

```
hammer/
    mytech/
        __init__.py
        defaults.yml
        mytech.tech.json
        layer.map         # some tech-specific file
        action/
            action_tool/
                __init__.py   # contains action_tool-specific hooks for mytech technology
                tool.options  # some tech-specific file needed for this tool
```

In this case, the technology package name is `hammer.mytech` (note that standalone plugin repositories may not be `hammer.technology.mytech`, unlike the ones included in the main Hammer repository).

1.3.5 Resource Files

Technology plugins will often provide special static files needed by tools, such as the `layer.map` file in the above example. If this file is already specified with the `prependlocal` meta action, the behavior will remain the same:

```
action.tool.layer_map: "layer.map"
action.tool.layer_map_meta: prependlocal
```

However, within a plugin, if you need to access to a specific file for a tool-specific hook, for example, in `action/action_tool/__init__.py`, replace this:

```
with open(os.path.join(self.tool_dir, "tool.options")) as f:
```

With this and append `Pathlib` methods like `read_text()` as required:

```
importlib.resources.files(self.package).joinpath("tool.options")
```

1.3.6 pyproject.toml

Plugins that are repositories separate from Hammer must now have a `pyproject.toml` file so that they become poetry projects. Refer to *From Another Poetry Project* for details.

CHAPTER

TWO

HAMMER TECHNOLOGY PLUGINS

This guide will walk you through how to set up a technology plugin to be used in Hammer.

You may use the included free ASAP7 PDK or the open-source Sky130 PDK plugins as reference when building your own technology plugin.

Technology plugins must be structured as Python packages underneath the `hammer` package. The package should contain an class object named `tech` to create an instance of the technology. This object should be a subclass of `HammerTechnology`.

Technology plugins must also have `.tech.json` and `defaults.yml` files. See the following sections for how to write them.

2.1 Hammer Technologies

Hammer currently has open-source technology plugins in the `hammer/technology` folder. Other proprietary technology plugins cannot be made public without 3-way NDAs or special agreements with their respective foundries.

The structure of each technology plugin package is as follows:

- hammer
  - TECHNOLOGY_NAME
    - `__init__.py` contains the tech class object and methods for PDK installation/extraction and getting *hooks*.
    - `<name>.tech.json` contains the static information about the technology. See the section *Hammer Tech JSON* for a guide.
    - `defaults.yml` contains the default tech-specific *Hammer IR and Meta Variables*. See the section *Hammer Tech defaults.yml* for a guide.
    - ACTION_NAME
      - `__init__.py` (optional) can contain a technology's implementation of an action. Commonly, this is used for the SRAM compilation action.
      - TOOL_NAME
      - `__init__.py` (optional) contains callable *hook* methods specific to the technology and tool, if there are too many to put in TECHNOLOGY_NAME/__init__.py.

Resources that are needed may go in this directory structure where necessary.

2.1.1 HammerTechnology Class

The HammerTechnology class is the base class that all technology plugins should inherit from and is defined in `hammer/tech/__init__.py`. Particularly useful methods are:

- `post_install_script`: the plugin subclass should override this method to apply any non-default PDK installation steps or hotfixes in order to set up the technology libraries for use with Hammer.
- `read_libs`: this is a particularly powerful method to read the libraries contained in the `tech.json` file and filter them using the filters also contained the same file. See the *Library Filters* section for details.
- `get_tech_<action>_hooks`: these methods are necessary if tech–specific hooks are needed, and must return a list of hooks when a given tool name implementing the relevant action is called. Refer to the *Extending Hammer with Hooks* section for details about how to write hooks.

2.2 Hammer Tech JSON

The `tech.json` for a given technology sets up some general information about the install of the PDK, sets up DRC rule decks, sets up pointers to PDK files, and supplies technology stackup information. For the full schema of the tech JSON, please see the *Full Schema* section below, which is derived from the `TechJSON` Pydantic BaseModel in `hammer/tech/__init__.py`.

2.2.1 Technology Install

The user may supply the PDK to Hammer as an already extracted directory and/or as a tarball that Hammer can automatically extract. Setting `technology.TECH_NAME. install_dir` and/or `tarball_dir` (key is setup in the defaults.yml) will fill in as the path prefix for paths supplied to PDK files in the rest of the `tech.json`. Below is an example of the installs and tarballs from the ASAP7 plugin.

```
"name": "ASAP7 Library",
"grid_unit": "0.001",
"installs": [
  {
    "id": "$PDK",
    "path": "technology.asap7.pdk_install_dir"
  },
  {
    "id": "$STDCELLS",
    "path": "technology.asap7.stdcell_install_dir"
  }
],
"tarballs": [
  {
    "root": {
      "id": "ASAP7_PDK_CalibreDeck.tar",
      "path": "technology.asap7.tarball_dir"
    },
    "homepage": "http://asap.asu.edu/asap/",
    "optional": true
  }
],
```

The `id` field is used within the file listings further down in the file to prefix `path`, as shown in detail below. If the file listing begins with `cache`, then this denotes files that exist in the tech cache, which are generally placed there by the tech plugin's post-installation script (see ASAP7's `post_install_script` method). Finally, the encrypted Calibre decks are provided in a tarball and denoted as optional.

2.2.2 DRC/LVS Deck Setup

As many DRC & LVS decks for as many tools can be specified in the `drc decks` and `lvs decks` keys. Additional DRC/LVS commands can be appended to the generated run files by specifying raw text in the `additional_drc_text` and `additional_lvs_text` keys. Below is an example of an LVS deck from the ASAP7 plugin.

```
"lvs_decks": [
  {
    "tool_name": "calibre",
    "deck_name": "all_lvs",
    "path": "ASAP7_PDK_CalibreDeck.tar/calibredecks_r1p7/calibre/ruledirs/lvs/lvsRules_
↪calibre_asap7.rul"
  }
],
"additional_lvs_text": "LVS SPICE EXCLUDE CELL \*SRAM*RW*\"\nLVS BOX \"SRAM*RW*\"\nLVS␣
↪FILTER \*SRAM*RW*\" OPEN",
```

The file pointers, in this case, use the tarball prefix because Hammer will be extracting the rule deck directly from the ASAP7 tarball. The additional text is needed to tell Calibre that the dummy SRAM cells need to be filtered from the source netlist and boxed and filtered from the layout.

2.2.3 Library Setup

The `libraries` key also must be setup in the JSON plugin. This will tell Hammer where to find all of the relevant files for standard cells and other blocks for the VLSI flow. Below is an example of the start of the library setup and one entry from the ASAP7 plugin.

```
"libraries": [
  {
    "lef_file": "$STDCELLS/techlef_misc/asap7_tech_4x_201209.lef",
    "provides": [
      {
        "lib_type": "technology"
      }
    ]
  },
  {
    "nldm_liberty_file": "$STDCELLS/LIB/NLDM/asap7sc7p5t_SIMPLE_RVT_TT_nldm_201020.lib.gz
↪",
    "verilog_sim": "$STDCELLS/Verilog/asap7sc7p5t_SIMPLE_RVT_TT_201020.v",
    "lef_file": "$STDCELLS/LEF/scaled/asap7sc7p5t_27_R_4x_201211.lef",
    "spice_file": "$STDCELLS/CDL/LVS/asap7sc7p5t_27_R.cdl",
    "gds_file": "$STDCELLS/GDS/asap7sc7p5t_27_R_201211.gds",
    "qrc_techfile": "$STDCELLS/qrc/qrcTechFile_typ03_scaled4xV06",
    "spice_model_file": {
      "path": "$PDK/models/hspice/7nm_TT.pm"
    },
```

(continues on next page)

(continued from previous page)

```
    "corner": {
      "nmos": "typical",
      "pmos": "typical",
      "temperature": "25 C"
    },
    "supplies": {
      "VDD": "0.70 V",
      "GND": "0 V"
    },
    "provides": [
      {
        "lib_type": "stdcell",
        "vt": "RVT"
      }
    ]
  },
```

The file pointers, in this case, use the `$PDK` and `$STDCELLS` prefix as defined in the installs. The `corner` key tells Hammer what process and temperature corner that these files correspond to. The `supplies` key tells Hammer what the nominal supply for these cells are. The `provides` key has several sub-keys that tell Hammer what kind of library this is (examples include `stdcell`, `fiducials`, `io pad cells`, `bump`, and `level shifters`) and the threshold voltage flavor of the cells, if applicable. Adding the tech LEF for the technology with the `lib_type` set as `technology` is necessary for place and route.

Library Filters

Library filters are defined in the `LibraryFilter` class in `hammer/tech/__init__.py`. These allow you to filter the entire set of libraries based on specific conditions, such as a file type or corner. Additional functions can be used to extract paths, strings, sort, and post-process the filtered libraries.

For a list of pre-built library filters, refer to the properties in the `LibraryFilterHolder` class in the same file, accessed as `hammer.tech.filters.<filter_method>`

2.2.4 Stackup

The `stackups` sets up the important metal layer information for Hammer to use. Below is an example of one metal layer in the `metals` list from the ASAP7 example tech plugin.

```
{"name": "M3", "index": 3, "direction": "vertical", "min_width": 0.072, "pitch": 0.144,
↪"offset": 0.0, "power_strap_widths_and_spacings": [{"width_at_least": 0.0, "min_spacing
↪": 0.072}], "power_strap_width_table": [0.072, 0.36, 0.648, 0.936, 1.224, 1.512]}
```

All this information is typically taken from the tech LEF and can be automatically filled in with a script. The metal layer name and layer number is specified. `direction` specifies the preferred routing direction for the layer. `min_width` and `pitch` specify the minimum width wire and the track pitch, respectively. `power_strap_widths_and_spacings` is a list of pairs that specify design rules relating to the widths of wires and minimum required spacing between them. This information is used by Hammer when drawing power straps to make sure it is conforming to some basic design rules.

2.2.5 Sites

The `sites` field specifies the unit standard cell size of the technology for Hammer.

```
"sites": [
  {"name": "asap7sc7p5t", "x": 0.216, "y": 1.08}
]
```

This is an example from the ASAP7 tech plugin in which the `name` parameter specifies the core site name used in the tech LEF, and the `x` and `y` parameters specify the width and height of the unit standard cell size, respectively.

2.2.6 Special Cells

The `special_cells` field specifies a set of cells in the technology that have special functions. The example below shows a subset of the ASAP7 tech plugin for 2 types of cells: `tapcell` and `stdfiller`.

```
"special_cells": [
  {"cell_type": "tapcell", "name": ["TAPCELL_ASAP7_75t_L"]},
  {"cell_type": "stdfiller", "name": ["FILLER_ASAP7_75t_R", "FILLER_ASAP7_75t_L",
↪"FILLER_ASAP7_75t_SL", "FILLER_ASAP7_75t_SRAM", "FILLERxp5_ASAP7_75t_R", "FILLERxp5_
↪ASAP7_75t_L", "FILLERxp5_ASAP7_75t_SL", "FILLERxp5_ASAP7_75t_SRAM"]},
```

See the `SpecialCell` subsection in the *Full Schema* for a list of special cell types. Depending on the tech/tool, some of these cell types can only have 1 cell in the `name` list.

There is an optional `size` list. For each element in its corresponding `name` list, a size (type: str) can be given. An example of how this is used is for `decap` cells, where each listed cell has a typical capacitance, which a place and route tool can then use to place decaps to hit a target total decapacitance value. After characterizing the ASAP7 decaps using Voltus, the nominal capacitance is filled into the `size` list:

```
{"cell_type": "decap", "name": ["DECAPx1_ASAP7_75t_R", "DECAPx1_ASAP7_75t_L", "DECAPx1_
↪ASAP7_75t_SL", "DECAPx1_ASAP7_75t_SRAM", "DECAPx2_ASAP7_75t_R", "DECAPx2_ASAP7_75t_L",
↪"DECAPx2_ASAP7_75t_SL", "DECAPx2_ASAP7_75t_SRAM", "DECAPx2b_ASAP7_75t_R", "DECAPx2b_
↪ASAP7_75t_L", "DECAPx2b_ASAP7_75t_SL", "DECAPx2b_ASAP7_75t_SRAM", "DECAPx4_ASAP7_75t_R
↪", "DECAPx4_ASAP7_75t_L", "DECAPx4_ASAP7_75t_SL", "DECAPx4_ASAP7_75t_SRAM", "DECAPx6_
↪ASAP7_75t_R", "DECAPx6_ASAP7_75t_L", "DECAPx6_ASAP7_75t_SL", "DECAPx6_ASAP7_75t_SRAM",
↪"DECAPx10_ASAP7_75t_R", "DECAPx10_ASAP7_75t_L", "DECAPx10_ASAP7_75t_SL", "DECAPx10_
↪ASAP7_75t_SRAM"], "size": ["0.39637 fF", "0.402151 fF", "0.406615 fF", "0.377040 fF",
↪"0.792751 fF", "0.804301 fF", "0.813231 fF", "0.74080 fF", "0.792761 fF", "0.804309 fF
↪", "0.813238 fF","0.75409 fF", "1.5855 fF", "1.6086 fF", "1.62646 fF", "1.50861 fF",
↪"2.37825 fF", "2.4129 fF", "2.43969 fF", "2.26224 fF", "3.96376 fF", "4.02151 fF", "4.
↪06615 fF", "3.7704 fF"]},
```

2.2.7 Don't Use, Physical-Only Cells

The `dont_use_list` is used to denote cells that should be excluded due to things like bad timing models or layout. The `physical_only_cells_list` is used to denote cells that contain only physical geometry, which means that they should be excluded from netlisting for simulation and LVS. Examples from the ASAP7 plugin are below:

```
"dont_use_list": [
    "ICGx*DC*",
    "AND4x1*",
```

(continues on next page)

(continued from previous page)

```
    "SDFLx2*",
    "AO21x1*",
    "XOR2x2*",
    "OAI31xp33*",
    "OAI221xp5*",
    "SDFLx3*",
    "SDFLx1*",
    "AOI211xp5*",
    "OAI322xp33*",
    "OR2x6*",
    "A2O1A1O1Ixp25*",
    "XNOR2x1*",
    "OAI32xp33*",
    "FAx1*",
    "OAI21x1*",
    "OAI31xp67*",
    "OAI33xp33*",
    "AO21x2*",
    "AOI32xp33*"
],
"physical_only_cells_list": [
  "TAPCELL_ASAP7_75t_R", "TAPCELL_ASAP7_75t_L", "TAPCELL_ASAP7_75t_SL", "TAPCELL_ASAP7_
↪75t_SRAM",
  "TAPCELL_WITH_FILLER_ASAP7_75t_R", "TAPCELL_WITH_FILLER_ASAP7_75t_L", "TAPCELL_WITH_
↪FILLER_ASAP7_75t_SL", "TAPCELL_WITH_FILLER_ASAP7_75t_SRAM",
  "FILLER_ASAP7_75t_R", "FILLER_ASAP7_75t_L", "FILLER_ASAP7_75t_SL", "FILLER_ASAP7_75t_
↪SRAM",
  "FILLERxp5_ASAP7_75t_R", "FILLERxp5_ASAP7_75t_L", "FILLERxp5_ASAP7_75t_SL", "FILLERxp5_
↪ASAP7_75t_SRAM"
],
```

2.2.8 Full Schema

Note that in the the schema tables presented below, items with `#/definitions/<class_name>` are defined in other schema tables. This is done for documentation clarity, but in your JSON file, those items would be hierarchically nested.

TechJSON

type	*object*		
properties			
• **name**	*Name*		
	type	*string*	
• grid_unit	*Grid Unit*		
	default	null	
	anyOf	type	*string*
		type	*null*
• shrink_factor	*Shrink Factor*		
	default	null	
	anyOf	type	*string*

continues on next page

Table 1 – continued from previous page

		type	*null*	
• installs	*Installs*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/PathPrefix	
		type	*null*	
• libraries	*Libraries*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/Library	
		type	*null*	
• gds_map_file	*Gds Map File*			
	default	null		
	anyOf	type	*string*	
		type	*null*	
• physical_only_cells_list	*Physical Only Cells List*			
	default	null		
	anyOf	type	*array*	
		items	type	*string*
		type	*null*	
• dont_use_list	*Dont Use List*			
	default	null		
	anyOf	type	*array*	
		items	type	*string*
		type	*null*	
• drc_decks	*Drc Decks*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/DRCDeck	
		type	*null*	
• lvs_decks	*Lvs Decks*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/LVSDeck	
		type	*null*	
• tarballs	*Tarballs*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/Tarball	
		type	*null*	
• sites	*Sites*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/Site	
		type	*null*	
• stackups	*Stackups*			
	default	null		
	anyOf	type	*array*	
		items	#/$defs/Stackup	
		type	*null*	
• special_cells	*Special Cells*			

continues on next page

Table 1 – continued from previous page

	default	null	
	anyOf	type	*array*
		items	#/$defs/SpecialCell
		type	*null*
• extra_prefixes	*Extra Prefixes*		
	default	null	
	anyOf	type	*array*
		items	#/$defs/PathPrefix
		type	*null*
• additional_lvs_text	*Additional Lvs Text*		
	default	null	
	anyOf	type	*string*
		type	*null*
• additional_drc_text	*Additional Drc Text*		
	default	null	
	anyOf	type	*string*
		type	*null*

CellType

type	*string*
enum	tiehicell, tielocell, tiehilocell, endcap, iofiller, stdfiller, decap, tapcell, driver, ctsbuffer, ctsinverter, ctsgate, ctslogic

Corner

type	*object*	
properties		
• **nmos**	*Nmos*	
	type	*string*
• **pmos**	*Pmos*	
	type	*string*
• **temperature**	*Temperature*	
	type	*string*

DRCDeck

type	*object*	
properties		
• **tool_name**	*Tool Name*	
	type	*string*
• **deck_name**	*Deck Name*	
	type	*string*
• **path**	*Path*	
	type	*string*

LVSDeck

type	*object*	
properties		
• **tool_name**	*Tool Name*	
	type	*string*
• **deck_name**	*Deck Name*	
	type	*string*
• **path**	*Path*	
	type	*string*

Library

type	*object*		
properties			
• name	*Name*		
	default	null	
	anyOf	type	*string*
		type	*null*
• ccs_liberty_file	*Ccs Liberty File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• ccs_library_file	*Ccs Library File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• ecsm_liberty_file	*Ecsm Liberty File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• ecsm_library_file	*Ecsm Library File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• corner	default	null	
	anyOf	#/$defs/Corner	
		type	*null*
• itf_files	default	null	
	anyOf	#/$defs/MinMaxCap	
		type	*null*
• lef_file	*Lef File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• klayout_techfile	*Klayout Techfile*		
	default	null	
	anyOf	type	*string*

continues on next page

Table 2 – continued from previous page

		type	*null*
• spice_file	*Spice File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• gds_file	*Gds File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• milkyway_lib_in_dir	*Milkyway Lib In Dir*		
	default	null	
	anyOf	type	*string*
		type	*null*
• milkyway_techfile	*Milkyway Techfile*		
	default	null	
	anyOf	type	*string*
		type	*null*
• nldm_liberty_file	*Nldm Liberty File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• nldm_library_file	*Nldm Library File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• openaccess_techfile	*Openaccess Techfile*		
	default	null	
	anyOf	type	*string*
		type	*null*
• provides	*Provides*		
	default	null	
	anyOf	type	*array*
		items	#/$defs/Provide
		type	*null*
• qrc_techfile	*Qrc Techfile*		
	default	null	
	anyOf	type	*string*
		type	*null*
• supplies	default	null	
	anyOf	#/$defs/Supplies	
		type	*null*
• tluplus_files	default	null	
	anyOf	#/$defs/MinMaxCap	
		type	*null*
• tluplus_map_file	*Tluplus Map File*		
	default	null	
	anyOf	type	*string*
		type	*null*
• verilog_sim	*Verilog Sim*		
	default	null	
	anyOf	type	*string*

continues on next page

Table 2 – continued from previous page

		type	*null*
• verilog_synth	*Verilog Synth*		
	default	null	
	anyOf	type	*string*
		type	*null*
• spice_model_file	default	null	
	anyOf	#/$defs/SpiceModelFile	
		type	*null*
• power_grid_library	*Power Grid Library*		
	default	null	
	anyOf	type	*string*
		type	*null*
• extra_prefixes	*Extra Prefixes*		
	default	null	
	anyOf	type	*array*
		items	#/$defs/PathPrefix
		type	*null*

Metal

A metal layer and some basic info about it.
name: Metal layer name (e.g. M1, M2). index: The order in the stackup (lower is closer to the substrate). direction: The preferred routing direction of this metal layer, or
RoutingDirection.Redistribution for non-routing top-level redistribution metals like Aluminium.
min_width: The minimum wire width for this layer. max_width: The maximum wire width for this layer. pitch: The minimum cross-mask pitch for this layer (NOT same-mask pitch
for multiple-patterned layers). Width of routing grid for a given layer. To route denser wires on chip, multiple masks are required. During fabrication, the masks are applied separately with some spatial offsets to achieve denser line patterning. For more information on multiple-patterning, check https://en.wikipedia.org/wiki/Multiple_patterning
offset: The routing track offset from the origin for the first track in this layer.
(0 = first track is on an axis).
power_strap_widths_and_spacings: A list of WidthSpacingTuples that specify the minimum
spacing rules for an infinitely long wire of variying width.
power_strap_width_table: A list of allowed metal widths in the technology.
Widths smaller than the last number must be quantized to a value in the table.
grid_unit: The fixed-point decimal value of a minimum grid unit (e.g. 1nm = 0.001).
For most technologies, this comes from the technology plugin and is the same for all layers.

type	*object*		
properties			
• **name**	*Name*		
	type	*string*	
• **index**	*Index*		
	type	*integer*	
• **direction**	#/$defs/RoutingDirection		
• **min_width**	*Min Width*		
	anyOf	type	*number*

continues on next page

Table 3 – continued from previous page

		type	*string*	
• max_width	*Max Width*			
	default	null		
	anyOf	type	*number*	
		type	*string*	
		type	*null*	
• **pitch**	*Pitch*			
	anyOf	type	*number*	
		type	*string*	
• **offset**	*Offset*			
	anyOf	type	*number*	
		type	*string*	
• **power_strap_widths_and_spacings**	*Power Strap Widths And Spacings*			
	type	*array*		
	items	#/$defs/WidthSpacingTuple		
• power_strap_width_table	*Power Strap Width Table*			
	type	*array*		
	default			
	items	anyOf	type	*number*
			type	*string*
• **grid_unit**	*Grid Unit*			
	anyOf	type	*number*	
		type	*string*	

MinMaxCap

type	*object*	
properties		
• **max_cap**	*Max Cap*	
	type	*string*
• **min_cap**	*Min Cap*	
	type	*string*

PathPrefix

A path prefix which defines an identifier and its corresponding path. Example: A PathPrefix(id = “Alib”, path = “/scratch/projectA/mylib”) maps the identifier ‘Alib’ to the path ‘/scratch/projectA/mylib’		
type	*object*	
properties		
• **id**	*Id*	
	type	*string*
• **path**	*Path*	
	type	*string*

Provide

type	*object*		
properties			
• **lib_type**	*Lib Type*		
	type	*string*	
• vt	*Vt*		
	default	null	
	anyOf	type	*string*
		type	*null*

RoutingDirection

Represents a preferred routing direction for a metal layer. Note that this represents a *preferred* direction, not a DRC rule.	
type	*string*
enum	vertical, horizontal, redistribution

Site

A standard cell site, which is the minimum unit of x and y dimensions a standard cell can have. name: The name of this site (often something like “core”) as defined in the tech and standard cell LEFs x: The x dimension y: The y dimension			
type	*object*		
properties			
• **name**	*Name*		
	type	*string*	
• **x**	*X*		
	anyOf	type	*number*
		type	*string*
• **y**	*Y*		
	anyOf	type	*number*
		type	*string*

SpecialCell

type	*object*		
properties			
• **cell_type**	#/$defs/CellType		
• **name**	*Name*		
	type	*array*	
	items	type	*string*
• size	*Size*		
	default	null	
	anyOf	type	*array*
		items	type *string*
		type	*null*
• input_ports	*Input Ports*		
	default	null	
	anyOf	type	*array*
		items	type *string*
		type	*null*
• output_ports	*Output Ports*		
	default	null	
	anyOf	type	*array*
		items	type *string*
		type	*null*

SpiceModelFile

type	*object*		
properties			
• **path**	*Path*		
	type	*string*	
• lib_corner	*Lib Corner*		
	default	null	
	anyOf	type	*string*
		type	*null*

Stackup

A stackup is a list of metals with a meaningful keyword name (for now). TODO: add vias, etc when we need them			
type	*object*		
properties			
• **grid_unit**	*Grid Unit*		
	anyOf	type	*number*
		type	*string*
• **name**	*Name*		
	type	*string*	
• **metals**	*Metals*		
	type	*array*	
	items	#/$defs/Metal	

Supplies

type	*object*	
properties		
• **GND**	*Gnd*	
	type	*string*
• **VDD**	*Vdd*	
	type	*string*

Tarball

type	*object*	
properties		
• **root**	#/$defs/PathPrefix	
• **homepage**	*Homepage*	
	type	*string*
• optional	*Optional*	
	type	*boolean*
	default	False

WidthSpacingTuple

A tuple of wire width limit and spacing for generating a piecewise linear rule for spacing based on wire width. width_at_least: Any wires larger than this must obey the minSpacing rule. min_spacing: The minimum spacing for this bin. If a wire is wider than multiple entries, the worst-case (larger) minSpacing wins.			
type	*object*		
properties			
• **width_at_least**	*Width At Least*		
	anyOf	type	*number*
		type	*string*
• **min_spacing**	*Min Spacing*		
	anyOf	type	*number*
		type	*string*

2.3 Hammer Tech defaults.yml

The `defaults.yml` for a technology specifies some technology-specific *Hammer IR and Meta Variables* that should be left as default unless you desire to override them. Some of the them work directly with the keys in the `tech.json`.

Most of the keys in the `defaults.yml` are a part of the `vlsi` and `technology` namespaces. An example of the setup of the `defaults.yml` is located in `hammer/technology/asap7/defaults.yml` and certain important keys should be common to most technology plugins:

- `vlsi.core.node` defines the node that the place-and-route tool expects. It affects what kind of licenses are needed.
- `vlsi.inputs` should at least have the nominal supplies and a typical pair of characterized setup & hold corners.
- `vlsi.technology` needs to specify a `placement_site` as defined in the technology LEF, a `bump_block_cut_layer` to set blockages under bumps, and optional `tap_cell_interval` and `tap_cell_offset` for placing well taps.
- `technology.core` needs to specify the stackup to use, which layer the standard cell power rails are on, and a reference cell to draw the lowest layer power rails over.

Tool environment variables (commonly needed for DRC/LVS decks) and other necessary default options specific to the technology should be set in this file. This file can also define technology-specific Hammer IR, in the the namespace `technlogy.<tech_name>.<key_name>`. As always, they can be overriden by other snippets of Hammer IR.

The data types for all keys in `defaults.yml` can be found in `defaults_types.yml`. When adding or overriding to `defaults.yml`, make sure that said data types are updated accordingly to prevent problems with the type checker.

2.4 Sky130 Technology Library

Hammer supports the Skywater 130nm Technology process. The SkyWater Open Source PDK is a collaboration between Google and SkyWater Technology Foundry to provide a fully open source Process Design Kit (PDK) and related resources, which can be used to create manufacturable designs at SkyWater's facility.

2.4.1 PDK Setup

The Skywater 130nm PDK files are located in a repo called skywater-pdk. A tool called Open-PDKs (open_pdks) was developed to generate all the files typically found in a PDK. Open-PDKs uses the contents in `skywater-pdk`, and outputs files to a directory called `sky130A`.

PDK Install

```
# create a root directory that will contain all PDK files and supporting tools (install␣
↪size is ~42GB)
export PREFIX=/path/to/install/root
mkdir -p $PREFIX
cd $PREFIX

# install magic via conda, required for open_pdks
conda create -y -c litex-hub --prefix $PREFIX/.conda-signoff magic
export PATH=$PREFIX/.conda-signoff/bin:$PATH

# clone required repos
git clone https://github.com/google/skywater-pdk.git
git clone https://github.com/RTimothyEdwards/open_pdks.git

# install Sky130 PDK via Open-PDKs
#    we disable some install steps to save time
cd $PREFIX/open_pdks
./configure \
    --enable-sky130-pdk=${PREFIX}/skywater-pdk/libraries --prefix=$PREFIX \
    --disable-gf180mcu-pdk --disable-alpha-sky130 --disable-xschem-sky130 --disable-
↪primitive-gf180mcu \
    --disable-verification-gf180mcu --disable-io-gf180mcu --disable-sc-7t5v0-gf180mcu \
    --disable-sc-9t5v0-gf180mcu --disable-sram-gf180mcu --disable-osu-sc-gf180mcu
make
make install
```

This generates all the Sky130 PDK files and installs them to `$PREFIX/share/pdk/sky130A`

Now in your Hammer YAML configs, point to the location of this install:

```
technology.sky130.sky130A: "<PREFIX>/share/pdk/sky130A"
```

2.4.2 SRAM Macros

If you are using SRAMs in your design, such as for the Chipyard Sky130 tutorial, you will require a set of SRAM macros. The Sky130 PDK did not come with it's own SRAM macros, so these have been generated by various third-party contributors. Hammer is compatible with Sky130 SRAM macros generated by Sram22 and OpenRAM, but starting with Hammer v1.0.2 only Sram22 macros will be supported. To obtain these macros, clone their github repo:

```
git clone https://github.com/rahulk29/sram22_sky130_macros
```

Then set the respective Hammer YAML key:

```
technology.sky130.sram22_sky130_macros: "/path/to/sram22_sky130_macros"
```

Note that the various configurations of the SRAMs available are encoded in the file `sram-cache.json`. To modify this file to include different configurations, or switch to using the OpenRAM SRAMs, navigate to `./extra/[sram22|openram]` and run the script `./sram-cache-gen.py` for usage information.

2.4.3 IO Library

The IO ring required by efabless for MPW/ChipIgnite can be created in Innovus using the `sky130_fd_io` and `sky130_ef_io` IO cell libraries. Here are the steps to use them:

1. `extra/efabless_template.io` is a template IO file. You should modify this by replacing the `<inst_path>`s with the netlist paths to your GPIO & analog pads. **DO NOT MODIFY ANY POSITIONS OR REPLACE CLAMP CELLS WITH IO CELLS**.

   a. For pad assignment: the ordering in the instance lists are from left to right (for top/bottom edges) and **bottom to top (for left/right edges)**.

   b. Refer to this documentation for how to configure the pins of the IO cells (not exhaustive).

   c. Your chip reset signal must go thru the `xres4v2` cell. Since this is in your netlist, you must remove the `cell=...` instantiation from your IO file (it is only in the template for clarity) and update the inst name. Otherwise a separate instance will be placed instead.

   d. The `ENABLE_INP_H` pin must be hard-tied to `TIE_HI_ESD` or `TIE_LO_ESD`. Since this is at a higher voltage, verify that this is routed as a wire only (no buffers can be inserted).

   e. `ENABLE_H` must be low at chip startup before going high. Absent using the power detector cell from the NDA IO library, you may elect to connect this to a reset signal.

   f. This template file does not contain dedicated clamps for the `VSWITCH` or `VCCHIB` supplies (following Caravel). EFabless provides a `sky130_ef_io__connect_vcchib_vccd_and_vswitch_vddio_slice_20um` slice in `open_pdks` that replaces a standard 20um spacer with a slice that connects `VCCHIB` and `VCCD` together, and `VSWITCH` and `VDDIO` together. Note that this slice cannot be placed immediately to the right (in the R0 orientation) of a `*_clamped3_pad` cell, because otherwise they *will* create a supply short. The template IO file contains normal 20um spacer slices explicitly placed at these critical locations, and the provided hook instantiates the `connect` slice in place of the standard 20um spacer. This can be modified if desired. Caravel distributes the `*connect*` slices across the bottom edge of the padframe.

2. Then, in your design YAML file, specify your IO file with the following. The top-level constraint must be exactly as below:

```
technology.sky130.io_file: <path/to/ring.io>
technology.sky130.io_file_meta: prependlocal

path: Top
type: toplevel
x: 0
y: 0
width: 3588
height: 5188
margins:
  left: 249.78
  right: 249.78
  top: 252.08
  bottom: 252.08
```

3. In your CLIDriver, you must import the following hook from the tech plugin and insert it as a `post_insertion_hook` after `floorplan_design`.

```
from hammer.technology.sky130 import efabless_ring_io
```

In addition, to ensure ties to `TIE_HI_ESD` / `TIE_LO_ESD` are preserved during synthesis, a `post_insertion_hook` to `init_environment` should be added to `dont_touch` the IO cells

```
def donttouch_iocells(x: HammerTool) -> bool:
    x.append('set_dont_touch [get_db insts -if {.base_cell.name == sky130_ef_io__*}
↪] true')
    return True
```

4. If you want to use the NDA s8iom0s8 library, you must include the `s8io.yml` file with `-p` on the `hammer-vlsi` command line, and then change the cells to that library in the IO file. Net names in the hook above will need to be lower-cased.
5. DRC requires a rectangle of `areaid.lowTapDensity` (GDS layer 81/14) around the core area to check latchup correctly. Currently, this is not yet implemented in Hammer, and will need to be added manually in a GDS editor after GDS streamout.

2.4.4 NDA Files

The NDA version of the Sky130 PDK is only required for Siemens Calibre to perform DRC/LVS signoff with the commercial VLSI flow. It is NOT REQUIRED for the remaining commercial flow, as well as the open-source tool flow. Therefore this NDA PDK is not used to generate a GDS.

If you have access to the NDA repo, you should add this path to your Hammer YAML configs:

```
technology.sky130.sky130_nda: "/path/to/skywater-src-nda"
```

We use the Calibre decks in the `s8` PDK, version `V2.0.1`, see here for the DRC deck path and here for the LVS deck path.

2.4.5 Resources

The good thing about this process being open-source is that most questions about the process are answerable through a google search. The tradeoff is that the documentation is a bit of a mess, and is currently scattered over a few pages and Github repos. We try to summarize these below.

SRAMs:

- Sram22 pre-compiled macros
  - Various pre-compiled sizes of SRAM macros to support Hammer Sky130 flow
- Sram22
  - Open-source SRAM generator
  - Currently only supports the Sky130 process
  - Very much under development, and some parts currently require commercial tools (for LEF and LIB generation)
- OpenRAM pre-compiled macros
  - Precompiled sizes are 1kbytes, 2kbytes and 4kbytes
- OpenRAM
  - Open-source static random access memory (SRAM) compiler

Git repos:

- SkyWater Open Source PDK
  - Git repo of the main Skywater 130nm files
- Open-PDKs
  - Git repo of Open-PDKs tool that compiles the Sky130 PDK
- Git repos on foss-eda-tools
  - Additional useful repos, such as Berkeley Analog Generator (BAG) setup

Documentation:

- SkyWater SKY130 PDK's documentation
  - Main documentation site for the PDK
- Join the SkyWater PDK Slack Channel
  - By far the best way to have questions about the process answered, with 80+ channels for different topics
- Skywater130 Standard Cell and Primitives Overview
  - Additional useful documentation for the PDK

2.5 ASAP7 Technology Library

Hammer's default demonstration PDK is ASAP7. There are some special setup and known issues with this open PDK.

2.5.1 Setup and Environment

In addition to requirements for `hammer-vlsi`, using ASAP7 also requires:

- ASAP7 PDK version 1p7, available here on GitHub. We recommend downloading an archive of or shallow cloning the repository.
- The Calibre deck tarball (downloaded separately from the website) must not be extracted. It should be placed in the directory specified by `technology.asap7.tarball_dir`. For ease, this can be same directory as the repository.
- Either the `gdstk` or `gdspy` GDS manipulation utility is required for 4x database downscaling. `gdstk` (available here on GitHub, version >0.6) is highly recommended; however, because it is more difficult to install, `gdspy` (available here on Github, specifically version 1.4 can also be used instead, but it is much slower.
- Calibre must be the DRC/LVS tool. The rule decks only support 2017-year Calibre versions.

*At this moment, for BWRC affiliates, the environment needed for a `gdstk` or `gdspy` install is setup only on the LSF cluster machines. To install it, first enable the Python dev environment:

```
scl enable rh-python36 bash
```

`gdstk` must be built from source, with dependencies such as CMake, LAPACK, and ZLIB. All must be installed from source into your own environment. For the ZLIB build, make sure the `-fPIC` CFLAG is used. Then, in the `gdstk` source, add the following CMake flags at the top of `setup.py`: `-DZLIB_LIBRARY=/path/to/your/libz.a` and `-DZLIB_INCLUDE_DIR=/path/to/your/include`. Then, in the `gdstk` source folder:

```
python3 setup.py install --user
```

For `gdspy`, the installation is much easier, depending just on `pip` and `numpy`:

```
python -m pip install gdspy --user
```

Or, replace the pip installation with installation from source in the `hammer/src/tools/gdspy` submodule.

2.5.2 Dummy SRAMs

The ASAP7 plugin comes with a set of dummy SRAMs, which are **NOT** used by default (not included in the default tech.json).

They are **completely blank** (full obstructions on layers M1-M3, will not pass DRC & LVS). All pins are on M4, with the signal all on the left side and the power stripes running across. The M5 power stripes are able to connect up.

All SRAMs are scaled up by 4x, so they are subject to the scaling script.

`sram-cache-gen.py` generates `sram-cache.json` using `srams.txt`, which contains a list of available SRAMs in Hammer IR. `sram-cache.json` memories is used by MacroCompiler to insert these memories into the design.

Finally, the SRAMCompiler in `sram_compiler/__init__.py` is used to generate the ExtraLibrarys (including .lib, .lef, .gds) needed by the particular design.

2.5.3 Known Issues

1. `ICG*DC*` cells are set as don't use due to improper LEF width.
2. Many additional cells are not LVS clean, and are set as don't use.
3. Innovus tries to fix non-existent M3 and M5 enclosure violations, unfortunately lengthening violation fixing time. Ignore these when reviewing the violations in Innovus.
4. If you specify core margins in the placement constraints, they left and bottom margins should be a multiple of 0.384 to avoid DRC violations. Layer offsets for M4-M7 are adjusted manually to keep all wires on-grid.
5. Common expected DRC violations (at reasonable, <70% local density):
    - M(4,5,6,7).AUX.(1,2) only if the floorplan size requirement above is not satisfied
    - V7.M8.AUX.2 and V2.M2.EN.1 due to incomplete via defs for V2 and V7 in power grid
    - FIN.S.1 appears to be incorrect, standard cell fins are indeed on the right pitch
    - LVT.W.1 caused by 0.5-width isolated-VT filler cells due to lack of implant layer spacing rules
    - LISD.S.3, LIG.S.4 due to some combinations of adjacent cells
    - V0.S.1 in ASYNC_DFFHx1
    - Various M4, GATE violations in/around dummy SRAMs

2.6 Nangate45 Technology Library

This is a PDK that comes packaged with OpenROAD-flow. Right now, it is only validated against the OpenROAD-flow backend of hammer.

2.6.1 Dummy SRAMs

The Nangate45 plugin comes with fake SRAMs without behavioral models or GDS files. They only have timing libs for the TT corner (same as the pdk's stcell lib).

2.6.2 Supported Tools

Only the OpenROAD-flow tools are suported, but compatibility between the two plugins is no longer maintained.

CHAPTER

THREE

HAMMER TOOL PLUGINS

This guide discusses the use and creation of CAD tool plugins in Hammer. A CAD tool plugin provides the actual implementation of Hammer APIs and outputs the TCL necessary to control its corresponding CAD tool.

Tool plugins must be structured as Python packages under the `hammer.action` package hierarchy. For example, if `dc` is a synthesis tool, it must be contained under `hammer/synthesis/dc`. The package should contain an class object named 'tool' to create an instance of the tool. `tool` should be a class object of an appropriate subclass of HammerTool (e.g. `HammerSynthesisTool`).

3.1 Hammer CAD Tools

Hammer currently has open-source CAD tool plugins in the `hammer/<action>` folders and three repos for CAD tools from commercial vendors: `hammer-cadence-plugins`, `hammer-synopsys-plugins`, and `hammer-mentor-plugins`. `hammer-mentor-plugins` is a private repo since it contains tool-specific commands not yet cleared for public release. Access to them may be granted for Hammer users who already have licenses for those tools. See the note about *plugins access* for instructions for how to request access.

The structure of each repository is as follows:

- hammer
  - ACTION
    - TOOL_NAME
      - `__init__.py` contains the methods needed to implement the tool
      - `defaults.yml` contains the default *Hammer IR and Meta Variables* needed by the tool

ACTION is the Hammer action name (e.g. `par`, `synthesis`, `drc`, etc.). TOOL_NAME is the name of the tool, which is referenced in your configuration. For example, if ACTION is `par` and TOOL_NAME is `par_tool_foo`, the configuration would reference it as `hammer.par.par_tool_foo`.

3.2 Setting up a Hammer CAD Tool Plugin

This guide will discuss what a Hammer user may do if they want to implement their own CAD tool plugin or extend the current CAD tool plugins. There are some basic mock-up examples of how this can be done in the `par` and `synthesis` directories inside `hammer/`.

3.2.1 Tool Class

Writing a tool plugin starts with writing the tool class. Hammer already provides a set of classes and mixins for a new tool to extend. For example, the Hammer Innovus plugin inherits from `HammerPlaceAndRouteTool` and `CadenceTool`.

3.2.2 Steps

Each tool implements a steps method. For instance, a portion of the steps method for a place-and-route tool may look like:

```
@property
def steps(self) -> List[HammerToolStep]:
    steps = [
        self.init_design,
        self.floorplan_design,
        self.route_design
    ]
    return self.make_steps_from_methods(steps)
```

Each of the steps are their own methods in the class that will write TCL that will execute with the tool.

3.2.3 Getting Settings

Hammer provides the method `get_setting("KEY_NAME")` for the tool to actually grab the settings from the user's input YML or JSON files. One example would be `self.get_setting("par.blockage_spacing")` so that Hammer can specify to the desired P&R tool what spacing to use around place and route blockages.

3.2.4 Writing TCL

Hammer provides two main methods for writing TCL to a file: `append` and `verbose_append`. Both do similar things but `verbose_append` will emit additional TCL code to print the command to the terminal upon execution.

3.2.5 Executing the Tool

When all the desired TCL has been written by various step methods, it is time to execute the tool itself. Hammer provides the method `run_executable(args, cwd=self.run_dir)` to do so. `args` is a Python list of flags to be run with the tool executable. `cwd=self.run_dir` sets the "current working directory" and allows the plugin to specify in what directory to execute the command.

3.2.6 Tool Outputs

After execution, the Hammer driver will emit a copy of the Hammer IR database in JSON format to the run directory as well as specific new fields created by the activity. The name of the output JSON files will be related to the activity type (e.g. `par-output.json` and `par-output-full.json` for the `par` activity). The `-full` verison contains the entire Hammer IR database, while the other version contains only the output entries created by this activity. The individual fields are created when the `export_config_outputs` method is called. Each implementation of this tool must override this method with a new one that calls its `super` method and appends any additional output fields to the output dictionary, as necessary.

3.2.7 defaults.yml

Each tool may optionally implement a set of default *Hammer IR and Meta Variables*. These defaults may set different values for global Hammer IR configuration, or add new tool-specific keys. Tool-specific keys must have a namespace in the format `<action>.<tool_name>.<key_name>`. This file should have a companion `defaults_types.yml` that defines the required types as well.

3.3 OpenROAD Place-and-Route Tool Plugin

3.3.1 Tool Setup

- OpenROAD - install using conda or from source
  - we recommend installing from source because the conda package was compiled with the GUI disabled
- KLayout (DEF to GDSII conversion) - install using conda or from source.

3.3.2 Tool Steps

See `__init__.py` for the implementation of these steps.

```
# INITIALIZE
init_design

# FLOORPLAN
floorplan_design
place_bumps
macro_placement
place_tapcells
power_straps

# PLACE
global_placement
io_placement
resize
detailed_placement

# CTS
clock_tree
clock_tree_resize
add_fillers

## ROUTING
global_route
global_route_resize
detailed_route

# FINISHING
extraction
write_design
```

3.3.3 Step Details

A few of the steps are detailed below, mainly to document some details that aren't present in the comments within the plugin. The plugin (`__init__.py` and `defaults.yml`) is still the best reference for what each step does, the explanations below are just more nuanced.

Macro Placement

At the beginning of the `floorplan_design` step, a file called `macros.txt` will be generated in `build/par-rundir` that lists for each macro: their name/path in the design, master cell name, width/height, and origin (x,y). These may help in setting the `vlsi.inputs.placement_constraints` key in your design configuration.

Currently, there are two ways to place macros.

If `par.openroad.floorplan_mode == generate` (default value), use the `vlsi.inputs.placement_constraints` key to specify the path and (x,y) coordinate of each macro in the design. The placement constraints for ALL macros must be specified, or OpenROAD will eventually throw an error about an unplaced instance. Note that the (x,y) position is fixed to the bottom left corner of the macro, then the orientation is applied. This means that for rotations besides `r0`, the (x,y) position WILL NOT correspond to the bottom left corner of the final placed macro (e.g. the (x,y) position for a macro with orientation `mx` will correspond to the top left corner of the final placed macro). To get around this, when `par.openroad.floorplan_origin_pos == bottom_left` (default value), the placement (x,y) coordinates are translated based on the rotation of the macro and the width/height specified in the `vlsi.inputs.placement_constraints`, so that the (x,y) position specified in `vlsi.inputs.placement_constraints` corresponds with the bottom left corner of the final placed macro. If `par.openroad.floorplan_origin_pos == rotated`, then the (x,y) position of the macro is not translated.

If `par.openroad.floorplan_mode == auto_macro`, you must still use the `vlsi.inputs.placement_constraints` key to specify the top-level width/height and margins constraints of the chip, but all macros will be placed automatically using OpenROAD's `macro_placement` command, leading to poor-quality but sane results.

Write Design

OpenROAD uses an external tool called KLayout, which executes a custom Python script `def2stream.py` (see `par.openroad.def2stream_file` for this file's path) to write the final design to a GDS file. This is the same process used in the default VLSI flow from OpenROAD.

3.3.4 Issue Archiving

This plugin supports generating a `tar.gz` archive of the current `build/par-rundir` directory with a `runme.sh` script to reproduce the results. This feature is enabled with the `par.openroad.create_archive_mode` YAML key, which can take the following values:

- `none` - never create an archive (default)
- `after_error` - if OpenROAD errors, create an archive of the run
- `always` - create an archive after every par run (regardless of whether OpenROAD errors)
- `latest_run` - create an archive of latest par run (don't run OpenROAD)
  - useful if OpenROAD gets stuck on endless optimization iterations but never actually "errors"

Creating an archive does the following:

- Generates a issue directory based on the date/time, top module, OS platform

- Dumps the logger output
- Generates a runme.sh script
- Copies .tcl, .sdc, .pdn, and .lef files in par-rundir to the issue dir
- Copies all input Verilog to issue dir
- Copies all input LEFs to issue dir (some may not be used since they are hacked in read_lef())
- Copies all input LIBs to issue dir
- Hacks copied par.tcl script to remove all abspaths
- Hacks copied par.tcl script to comment out write_gds block w/ KLayout
- Tars up issue dir

3.4 Cadence Joules RTL Power Tool Plugin

3.4.1 Tool Steps

See `__init__.py` for the implementation of these steps.

```
init_design
synthesize_design
report_power
```

A variety of different output reports may be generated with this tool. Within the Hammer default configs file, the `power.inputs.report_configs` struct description contains a summary of the different reporting options, specified via the `output_formats` struct field.

3.4.2 Known Issues

- Joules supports saving the read stimulus file to the SDB (stimulus database) format via the `write_sdb` command. However, subsequent reads of this SDB file via the `read_sdb` command fail for no apparent reason
  - As a result, `read_stimulus/compute_power` cannot be a separate step in the plugin, because there is no way to save the results of these commands before running the various power reporting commands. Thus these two commands are run as part of the `report_power` step.
  - NOTE: this might not be a problem anymore with the new Joules version, so we should re-try this!!

3.5 DRC/LVS with IC Validator

IC Validator is very command-line driven. Here are some usage notes:

- Many PDK decks will use variables to control switches. These are defined on command line with `-D` and can be defined in Hammer config using the `<drc/lvs>.icv.defines` key (type: List[Dict[str, str]]).
- Any deck directories that need to be included are defined on command line with `-I` and can be defined in Hammer config using the `<drc/lvs>.icv.include_dirs` key (type: List[str]).
- Extensibility is enabled by passing a file to the icv command with `-clf`. This file contains additional command line arguments, and is generated in the `generate_<drc/lvs>_args_file` step (can be overridden).

- Decks are included using the `generate_<drc/lvs>_run_file` step (can be overridden with additional ICV method calls).
- Results/violations are generated in a format readable by VUE (interactive violation browser) using the `-vue` option.
- Layout is viewed using IC Validator Workbench (ICVBV). It can communicate with VUE to generate violation markers by opening up a socket to ICV. The socket number can range between 1000 and 65535 (selectable by `<drc/lvs>.icv.icvwb_port`. Running the `generated_scripts/view_<drc/lvs>` script will handle this automatically, by starting ICVWB, opening the port, waiting for it to be listening, and then starting VUE.
- ICVWB layer mapping can be specified in `synopsys.layerprops` key.

Tested with:

- hammer-intech22-plugin

3.6 DRC/LVS with Pegasus

3.6.1 Pegasus usage notes

Although batch Pegasus is primarily command-line option driven, the only way to override certain directives in the rule decks is to generate a control file called `pegasusdrcctl` for DRC or `pegasuslvsctl` for LVS.

- This file can also be used to fill in fields in the GUI version for debugging, and changes in the the GUI will then overwrite this file.
- This file is generated by the `generate_drc_ctl_file/generate_lvs_ctl_file` step. Changes to what appears in this file must be done with either the key `drc.inputs.additional_drc_text/lvs.inputs.additional_lvs_text` with each line terminated by a semicolon (appends to the end of the generated file), or with a replacement hook.

3.6.2 Pegasus Design Review usage notes

Pegasus Design Review (DR) and Results Viewer (RV) together are the GUI DRC/LVS results browser.

The first time you open DR with the `view_drc/view_lvs` script, you will get an error message:

A Valid Technology File Path is Required? "": Not Found?

This is because you need to load the DRC/LVS deck as a technology.

To import the technology, in the main Design Review windows, go to:

Tools -> Technology Manager (window pops up) -> File -> Import -> PVL (tab)

In this dialog window, enter the technology basename in the Hammer input key `vlsi.core.technology` (e.g. "sky130", not "hammer.technology.sky130") in the "Technology Name" field and browse to the DRC/LVS deck in the PVL Rule File dialog. Then hit Import, close the Technology Manager, close DR, and run the `view_drc/view_lvs` script again.

Subsequent openings of databases anywhere will map to this technology and label layers properly because the technology information is stored your home directory.

Tested with:

- sky130

CHAPTER

FOUR

HAMMER FLOW STEPS

This documentation will walk through the currently supported steps of the Hammer flow: synthesis, place-and-route, DRC, LVS, and simulation. Two other action types are supported, but not discussed: PCB collateral generation, and SRAM generation.

4.1 Hammer Actions

Hammer has a set of actions including synthesis, place-and-route, DRC, LVS, simulation, SRAM generation, and PCB collateral generation. All of the Hammer actions and their associated key inputs can be found in `hammer/config/defaults.yml` and are documented in detail in each action's documentation.

Hammer will automatically pass the output files of one action to subsequent actions if those actions require the files. Hammer does this with conversion steps (e.g. `syn-to-par` for synthesis outputs to place-and-route inputs), which map the outputs from one tool into the inputs of another (e.g. `synthesis.outputs.output_files` maps to `par.outputs.input_files`). The Hammer make infrastructure builds these conversion rules automatically, so using the Make infrastructure is recommended. See the *Hammer Buildfile* section for more details.

Hammer actions are implemented using tools. See the *Hammer CAD Tools* section for details about how these tools are set up.

4.2 Synthesis

Hammer supports synthesizing Verilog-based RTL designs to gate-level netlists. This action requires a tool plugin to implement `HammerSynthesisTool`.

4.2.1 Synthesis Setup Keys

- Namespace: `vlsi.core`
  - **`synthesis_tool`**
    - Module of the synthesis tool, e.g. `hammer.synthesis.genus`

4.2.2 Synthesis Input Keys

- Namespace: `synthesis`
  - `inputs.input_files` ([])
    - A list of file paths to source files to be passed to the synthesis tool. The paths may be relative to the directory in which `hammer-vlsi` is called.
  - `inputs.top_module` (str)
    - Name of the top level verilog module of the design.
  - `clock_gating_mode` (str)
    - `auto`: turn on automatic clock gating inference in CAD tools
    - `empty`: do not do any clock gating

4.2.3 Synthesis Inputs

There are no prerequisites to running synthesis other than setting the keys that are described above.

4.2.4 Synthesis Outputs

- Mapped verilog file: `obj_dir/syn-rundir/{TOP_MODULE}.mapped.v`
- Mapped design SDF: `obj_dir/syn-rundir/{TOP_MODULE}.mapped.sdf`
- Synthesis output Hammer IR is contained in `obj_dir/syn-rundir/syn-output.json`
- Synthesis reports for gates, area, and timing are output in `obj_dir/syn-rundir/reports`

The synthesis output Hammer IR is converted to inputs for the P&R tool and the simulation tool by the Hammer `syn-to-par` and `syn-to-sim` commands, respectively.

4.2.5 Synthesis Commands

- Synthesis Command
  - `hammer-vlsi -e env.yml -p config.yml --obj_dir OBJ_DIR syn`
- Synthesis to Place-and-route
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json --obj_dir OBJ_DIR syn-to-par`
- Synthesis to Simulation
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json --obj_dir OBJ_DIR syn-to-sim`

4.3 Place-and-Route

Hammer has an action for placing and routing a synthesized design. This action requires a tool plugin to implement `HammerPlaceAndRouteTool`.

4.3.1 P&R Setup Keys

- Namespace: `vlsi.core`
  - `par_tool`
    - Module of the P&R tool, e.g. `hammer.par.innovus`

4.3.2 P&R Input Keys

- Namespace: `vlsi`
  - These are built-in Hammer APIs covered in *Hammer APIs*
- Namespace: `par`
  - `inputs.input_files` ([])
    - List of paths to post-synthesis netlists. Auto-populated after syn-to-par.
  - `inputs.top_module` (str)
    - Name of top RTL module to P&R. Auto-populated after syn-to-par.
  - `inputs.post_synth_sdc` (str)
    - Post-synthesis generated SDC. Auto-populated after syn-to-par.
  - `inputs.gds_map_mode` (str)
    - Specify which GDS layermap file to use. `auto` uses the technology-supplied file, whereas `manual` requires a file to specified via `inputs.gds_map_file`.
  - `inputs.gds_merge` (bool)
    - `True` tells the P&R tool to merge all library & macro GDS before streamout. Otherwise, only references will exist and merging needs to be done later, by a tool such as `Calibre`, `gdstk`, or `gdspy`.
  - `inputs.physical_only_cells_mode` (str)
    - Specifies which set of cells to exclude from SPICE netlist because they have no logical function. `auto` uses the technology-supplied list, whereas `manual` and `append` overrides and appends to the supplied list, respectively.
  - `submit` (dict)
    - Can override global settings for submitting jobs to a workload management platform.
  - `power_straps_mode` (str)
    - Power straps configuration. `generate` enables Hammer's power straps API, whereas `manual` requires a TCL script in `power_straps_script_contents`.
  - `blockage_spacing` (Decimal)
    - Global obstruction around every hierarchical sub-block and hard macro
  - `generate_power_straps_options` (dict)

* If `generate_power_straps_method` is `by_tracks`, this struct specifies all the options for the power straps API. See *Hammer APIs* for more detail.

4.3.3 P&R Inputs

There are no other prerequisites to running place & route other than setting the keys described above.

4.3.4 P&R Outputs

- Hierarchical (e.g. Cadence ILMs) and between-step snapshot databases in `OBJ_DIR/par-rundir`
- GDSII file with final design: `{OBJ_DIR}/par-rundir/{TOP_MODULE}.gds`
- Verilog gate-level netlist: `{OBJ_DIR}/par-rundir/{TOP_MODULE}.lvs.v`
- SDF file for post-par simulation: `{OBJ_DIR}/par-rundir/{TOP_MODULE}.sdf`
- Timing reports: `{OBJ_DIR}/par-rundir/timingReports`
- A script to open the final chip: `{OBJ_DIR}/par-rundir/generated_scripts/open_chip`

P&R output Hammer IR `{OBJ_DIR}/par-rundir/par-output.json` is converted to inputs for the DRC, LVS, and simulation tools by the `par-to-drc`, `par-to-lvs`, and `par-to-sim` actions, respectively.

4.3.5 P&R Commands

- P&R Command (after syn-to-par is run)
  - `hammer-vlsi -e env.yml -p {OBJ_DIR}/par-input.json --obj_dir OBJ_DIR par`
- P&R to DRC
  - `hammer-vlsi -e env.yml -p config.yml -p {OBJ_DIR}/par-rundir/par-output.json --obj_dir OBJ_DIR par-to-drc`
- P&R to LVS
  - `hammer-vlsi -e env.yml -p config.yml -p {OBJ_DIR}/par-rundir/par-output.json --obj_dir OBJ_DIR par-to-lvs`
- P&R to Simulation
  - `hammer-vlsi -e env.yml -p config.yml -p {OBJ_DIR}/par-rundir/par-output.json --obj_dir OBJ_DIR par-to-sim`

4.4 DRC

Hammer has an action for running design rules check (DRC) on a post-place-and-route GDS. This action requires a tool plugin to implement `HammerDRCTool`.

4.4.1 DRC Setup Keys

- Namespace: `vlsi.core`
  - `drc_tool`
    - Module of the DRC tool e.g. `hammer.drc.calibre`

4.4.2 DRC Input Keys

- Namespace: `drc`
  - `inputs.top_module` (str)
    - Name of top RTL module to run DRC on. Auto-populated after par-to-drc.
  - `inputs.layout_file` (str)
    - GDSII file from place-and-route. Auto-populated after par-to-drc.
  - `inputs.additional_drc_text_mode` (str)
    - Chooses what custom DRC commands to add to the run file. `auto` selects the one provided in the *Hammer Tech JSON*.
  - `inputs.drc_rules_to_run` ([str])
    - This selects a subset of the rules given in the technology's Design Rule Manual (DRM). The format of these rules will be technology- and tool-specific.
  - `submit` (dict)
    - Can override global settings for submitting jobs to a workload management platform.

4.4.3 DRC Inputs

There are no other prerequisites to running DRC other than setting the keys described above.

4.4.4 DRC Outputs

- DRC results report and database in `{OBJ_DIR}/drc-rundir`
- A run file: `{OBJ_DIR}/drc-rundir/drc_run_file`
- A script to interactively view the DRC results: `{OBJ_DIR}/drc-rundir/generated_scripts/view_drc`

4.4.5 DRC Commands

- DRC Command (after par-to-drc is run)
  - `hammer-vlsi -e env.yml -p {OBJ_DIR}/drc-input.json --obj_dir OBJ_DIR drc`

4.5 LVS

Hammer has an action for running layout-versus-schematic (LVS) on a post-place-and-route GDS and gate-levl netlist. This action requires a tool plugin to implement `HammerLVSTool`.

4.5.1 LVS Setup Keys

- Namespace: `vlsi.core`
  - `lvs_tool`
    - Module of the lvs tool, e.g. `hammer.lvs.calibre`

4.5.2 LVS Input Keys

- Namespace: `lvs`
  - `inputs.top_module` (str)
    - Name of top RTL module to run LVS on. Auto-populated after par-to-lvs.
  - `inputs.layout_file` (str)
    - GDSII file from P&R. Auto-populated after par-to-lvs.
  - `inputs.schematic_files` ([str])
    - Netlists from P&R'd design and libraries. Auto-populated after par-to-lvs.
  - `inputs.additional_lvs_text_mode` (str)
    - Chooses what custom LVS commands to add to the run file. `auto` selects the one provided in the *Hammer Tech JSON*.
  - `submit` (dict)
    - Can override global settings for submitting jobs to a workload management platform.

4.5.3 LVS Inputs

There are no other prerequisites to running LVS other than setting the keys described above.

4.5.4 LVS Outputs

- LVS results report and database in `{OBJ_DIR}/lvs-rundir`
- A run file: `{OBJ_DIR}/lvs-rundir/lvs_run_file`
- A script to interactively view the LVS results: `{OBJ_DIR}/lvs-rundir/generated_scripts/view_lvs`

4.5.5 LVS Commands

- LVS Command (after par-to-lvs is run)
  - `hammer-vlsi -e env.yml -p {OBJ_DIR}/lvs-input.json --obj_dir OBJ_DIR lvs`

4.6 Simulation

Hammer supports RTL, post-synthesis, and post-P&R simulation. It provides a simple API to add flags to the simulator call and automatically passes in collateral to the simulation tool from the synthesis and place-and-route outputs. This action requires a tool plugin to implement `HammerSimTool`.

4.6.1 Simulation Setup Keys

- Namespace: `vlsi.core`
  - **`sim_tool`**
    - Module of the simulation tool, e.g. `hammer.sim.vcs`

4.6.2 Simulation Input Keys

- Namespace: `sim.inputs`
  - `input_files` ([])
    - A list of file paths to source files (verilog sources, testharness/testbench, etc.) to be passed to the synthesis tool (both verilog and any other source files needed). The paths may be relative to the directory in which `hammer-vlsi` is called.
  - `top_module` (str)
    - Name of the top level module of the design.
  - `options` ([str])
    - Any options that are passed into this key will appear as plain text flags in the simulator call.
  - `defines` ([str])
    - Specifies define options that are passed to the simulator. e.g. when using VCS, this will be added as `+define+{DEFINE}`.
  - `compiler_cc_opts` ([str])
    - Specifies C compiler options when generating the simulation executable. e.g. when using VCS, each compiler_cc_opt will be added as `-CFLAGS {compiler_cc_opt}`.
  - `compiler_ld_opts` ([str])
    - Specifies C linker options when generating the simulation executable. e.g. when using VCS, each compiler_ld_opt will be added as `-LDFLAGS {compiler_ld_opt}`.
  - `timescale` (str)
    - Plain string that specifies the simulation timescale. e.g. when using VCS, `sim.inputs.timescale: '1ns/10ps'` would be passed as `-timescale=1ns/10ps`
  - `benchmarks` ([str])

    - A list of benchmark binaries that will be run with the simulator to test the design if the testbench requires it. For example, this may be RISC-V binaries. If unspecified or left empty, the simulation will execute as normal.

- `parallel_runs` (int)
    - Maximum number of simulations to run in parallel. -1 denotes all in parallel. 0 or 1 denotes serial execution.
- `tb_name` (str)
    - The name of the testbench/test driver in the simulation.
- `tb_dut` (str)
    - Hierarchical path to the to top level instance of the "dut" from the testbench.
- `level` (`"rtl"` or `"gl"`)
    - This defines whether the simulation being run is at the RTL level or at the gate level.
- `all_regs` (str)
    - Path to a list of all registers in the design, typically generated from the synthesis or P&R tool. This is used in gate level simulation to initialize register values.
- `seq_cells` (str)
    - Path to a list of all sequential standard cells in the design, typically generated from the synthesis or P&R tool. This is used in gate level simulation.
- `sdf_file` (str)
    - Path to Standard Delay Format file used in timing annotated simulations.
- `gl_register_force_value` (0 or 1)
    - Defines what value all registers will be initialized to for gate level simulations.
- `timing_annotated` (false or true)
    - Setting to false means that the simulation will be entirely functional. Setting to true means that the simulation will be time annotated based on synthesis or P&R results.
- `saif.mode` (`"time"`, `"trigger"`, `"trigger_raw"`, or `"full"`)
    - `"time"`: pair with `saif.mode.start_time` and `saif.mode.end_time` (TimeValue) to dump between 2 timestamps
    - `"trigger"`: inserts a trigger into the simulator run script
    - `"trigger_raw"`: inserts a given start/end trigger tcl script into the simulator run script. Specify scripts with `saif.mode.start_trigger_raw` and `saif.mode.end_trigger_raw` (str)
    - `"full"`: dump the full simulation
- `execution_flags` ([str])
    - Each string in this list will be passed as an option when actually executing the simulation executable generated from the previous arguments.
    - Can also use `execution_flags_prepend` and `execution_flags_append` for additional execution flags
- `execute_sim` (true or false)
    - Determines whether or not the simulation executable that is generated with the above inputs with the given flags or if the executable will just be generated.

4.6.3 Simulation Inputs

There are no prerequisites to running an RTL simulation other than setting the keys that are described above. Running the `syn-to-sim` action after running synthesis will automatically generate the Hammer IR required to pipe the synthesis outputs to the Hammer simulation tool, and should be included in the Hammer call, as demonstrated in the "Post-Synthesis Gate Level Sim" command below. The same goes for post-place-and-route simulations. The required files for these simulations (SDF, SPEF, etc.) are generated and piped to the simulation tool in the corresponding action's outputs.

The Hammer simulation tool will initialize register values in the simulation, as that is of particular need when simulating Chisel-based designs, to deal with issues around x-pessimism.

4.6.4 Simulation Outputs

The simulation tool is able to output waveforms for the simulation. All of the relevant outputs of the simulation can be found in `OBJ_DIR/sim-rundir/`.

4.6.5 Simulation Commands

- RTL Simulation Command
  - `hammer-vlsi -e env.yml -p config.yml --obj_dir OBJ_DIR sim`
- Synthesis to Sim
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json -o OBJ_DIR/syn-to-sim_input.json --obj_dir OBJ_DIR syn-to-sim`
- Post-Synthesis Gate Level Sim
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-to-sim_input.json --obj_dir OBJ_DIR sim`
- P&R to Simulation
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-rundir/par-output.json -o OBJ_DIR/par-to-sim_input.json --obj_dir OBJ_DIR par-to-sim`
- Post-P&R Gate Level Sim
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-to-sim_input.json --obj_dir OBJ_DIR sim`

4.7 Power

Hammer supports RTL, post-synthesis, and post-P&R power analysis. It provides a simple API to add flags to the power tool call and automatically passes in collateral to the power tool from the other tool steps. This action requires a tool plugin to implement `HammerPowerTool`.

4.7.1 Power Setup Keys

- Namespace: `vlsi.core`
  - **`power_tool`**
    - Module of the power tool, e.g. `hammer.power.voltus`

4.7.2 Simulation Input Keys

- Namespace: `power.inputs`
  - `database` (str)
    - Path to the place and route database of the design to be analyzed. This path may be relative to the directory in which `hammer-vlsi` is called.
  - `tb_name` (str)
    - The name of the testbench/test driver in the simulation.
  - `tb_dut` (str)
    - Hierarchical path to the to top level instance of the "dut" from the testbench.
  - `spefs` ([str])
    - List of paths to all spef (parasitic extraction) files for the design. This list may include a spef file per MMMC corner. Paths may be relative to the directory in which `hammer-vlsi` is called.
  - `waveforms` ([str])
    - List of paths to waveforms to be used for dynamic power analysis. Paths may be relative to the directory in which `hammer-vlsi` is called.
  - `start_times` ([TimeValue])
    - List of analysis start times corresponding to each of the `waveforms` used for dynamic power analysis.
  - `end_times` ([TimeValue])
    - List of analysis end times corresponding to each of the `waveforms` used for dynamic power analysis.
  - `saifs` ([str])
    - List of paths to SAIF (activity files) for dynamic power analysis. Generally generated by a gate-level simulation. Paths may be relative to the directory in which `hammer-vlsi` is called.
  - `extra_corners_only` (bool)
    - If overridden to `true`, the power tool will report for only the extra MMMC corners, saving runtime. The typical use case is to only report power and rail analysis for a typical/nominal corner.
  - **`input_files` ([str])**
    - A list of the paths to the design inputs files (HDL or netlist) for power analysis.
  - **`sdc` (str)**
    - Path to SDC input file.
  - **`report_configs` ([dict])**
    - List of report configs that specify `PowerReport` structs.
  - **`level` (FlowLevel)**

* Power analysis mode for different levels of the VLSI flow. The available options are `rtl`, `syn`, and `par`.

- **`top_module` (str)**

  * Top RTL module for power analysis.

4.7.3 Power Inputs

Hammer's power analysis can be run with an RTL input, or post-synthesis or post-place-and-route (and with corresponding simulations). Auto-translation of of Hammer IR to the power tool from those outputs are accomplished using the `sim-rtl-to-power`, `syn-to-power`, `sim-syn-to-power`, `par-to-power`, and `sim-par-to-power` actions, as demonstrated below. The required files for power analysis (database, SAIF, SPEF, etc.) are generated and piped to the power tool from the pre-requisite action's outputs.

4.7.4 Power Outputs

The power tool outputs static and active power estimations into the `OBJ_DIR/power-rundir/` directory. Exact report format may vary by tool used.

4.7.5 Power Commands

RTL Power Analysis:

- RTL Sim
  - `hammer-vlsi -e env.yml -p config.yml --obj_dir OBJ_DIR sim-rtl`
- Simulation to Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/sim-rundir/sim-rtl-output.json -o OBJ_DIR/sim-rtl-to-power_input.json --obj_dir OBJ_DIR sim-rtl-to-power`
- Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/sim-rtl-to-power_input.json --obj_dir OBJ_DIR power-rtl`

Post-synthesis Power Analysis:

- Syn to Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json -o OBJ_DIR/syn-to-power_input.json --obj_dir OBJ_DIR syn-to-power`
- Syn to Simulation
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json -o OBJ_DIR/syn-to-sim_input.json --obj_dir OBJ_DIR syn-to-sim`
- Post-Syn Gate Level Sim
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-to-sim_input.json --obj_dir OBJ_DIR sim-syn`
- Simulation to Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/sim-rundir/sim-syn-output.json -o OBJ_DIR/sim-syn-to-power_input.json --obj_dir OBJ_DIR sim-syn-to-power`

- Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-to-power_input.json -p OBJ_DIR/sim-syn-to-power_input.json --obj_dir OBJ_DIR power-syn`

Post-P&R Power Analysis:

- P&R to Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-rundir/par-output.json -o OBJ_DIR/par-to-power_input.json --obj_dir OBJ_DIR par-to-power`
- P&R to Simulation
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-rundir/par-output.json -o OBJ_DIR/par-to-sim_input.json --obj_dir OBJ_DIR par-to-sim`
- Post-P&R Gate Level Sim
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-to-sim_input.json --obj_dir OBJ_DIR sim-par`
- Simulation to Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/sim-rundir/sim-par-output.json -o OBJ_DIR/sim-par-to-power_input.json --obj_dir OBJ_DIR sim-par-to-power`
- Power
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-to-power_input.json -p OBJ_DIR/sim-par-to-power_input.json --obj_dir OBJ_DIR power-par`

4.8 Formal Verification

Hammer supports post-synthesis and post-P&R formal verification. It provides a simple API to provide a set of reference and implementation inputs (e.g. Verilog netlists) to perform formal verification checks such as logical equivalence checking (LEC). This action requires a tool plugin to implement `HammerFormalTool`.

4.8.1 Formal Verification Setup Keys

- Namespace: `vlsi.core`
  - **`formal_tool`**
    - Python module of the formal verification tool e.g. `hammer.formal.conformal`

4.8.2 Formal Verification Input Keys

- Namespace: `formal.inputs`
  - `check` (str)
    - Name of the formal verification check that is to be performed. Support varies based on the specific tool plugin. Potential check types/algorithms could be “lec”, “power”, “constraint”, “cdc”, “property”, “eco”, and more. At the moment, only “lec” is supported.
  - `input_files` ([])

    * A list of file paths to implementation source files (verilog, vhdl, spice, liberty, etc.) to be passed to the formal verification tool. For a LEC tool, this would be the sources for the "revised" design. The paths may be relative to the directory in which `hammer-vlsi` is called.

  - `reference_files` ([])

    * A list of file paths to reference source files (verilog, vhdl, spice, liberty, etc.) to be passed to the formal verification tool. For a LEC tool, this would be the sources for the "golden" design. The paths may be relative to the directory in which `hammer-vlsi` is called.

  - `top_module` (str)

    * Name of the top level module of the design to be verified.

4.8.3 Formal Verification Inputs

Running the `syn-to-formal` action after running synthesis will automatically generate the Hammer IR required to pipe the synthesis outputs to the Hammer formal verification tool, and should be included in the Hammer call, as demonstrated in the "Post-Synthesis Formal Verification" command below. The same goes for post-place-and-route formal verification.

At this time, only netlists are passed to the formal verification tool. Additional files (SDCs, CPFs, etc.) are needed for more advanced formal verification checks but are not yet currently supported. Similarly, formal verification on the behavioral RTL is also not yet supported.

4.8.4 Formal Verification Outputs

The formal verification tool produces reports in `OBJ_DIR/formal-rundir/`. Outputs from formal verification flows such as engineering change order (ECO) patches are not yet supported.

4.8.5 Formal Verification Commands

- Synthesis to Formal Verification
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json -o OBJ_DIR/syn-to-formal_input.json --obj_dir OBJ_DIR syn-to-formal`
- Post-Synthesis Formal Verification
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-to-formal_input.json --obj_dir OBJ_DIR formal`
- P&R to Formal Verification
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-rundir/par-output.json -o OBJ_DIR/par-to-formal_input.json --obj_dir OBJ_DIR par-to-formal`
- Post-P&R Formal Verification
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-to-formal_input.json --obj_dir OBJ_DIR formal`

4.9 Static Timing Analysis

Hammer supports post-synthesis and post-P&R static timing analysis. It provides a simple API to provide a set of design inputs (e.g. Verilog netlists, delay files, parasitics files) to perform static timing analysis (STA) for design signoff. This action requires a tool plugin to implement `HammerTimingTool`.

4.9.1 STA Setup Keys

- Namespace: `vlsi.core`
  - **`timing_tool`**
    - Module of the STA tool, e.g. `hammer.timing.tempus`

4.9.2 STA Input Keys

- Namespace: `timing.inputs`
  - `input_files` ([])
    - A list of file paths to the Verilog gate-level netlist to be passed to the STA tool. The paths may be relative to the directory in which `hammer-vlsi` is called.
  - `top_module` (str)
    - Name of the top level module of the design to be timed.
  - `post_synth_sdc` (str)
    - Post-synthesis generated SDC. Auto-populated after syn-to-timing.
  - `spefs` ([str])
    - List of paths to all spef (parasitic extraction) files for the design. This list may include a spef file per MMMC corner. Paths may be relative to the directory in which hammer-vlsi is called.
  - `sdf_file` (str)
    - Path to Standard Delay Format file. Auto-populated after syn-to-timing and par-to-timing.
  - `max_paths` (int)
    - Maximum number of timing paths to report from the STA tool. Large limits may hurt tool runtime.

4.9.3 STA Inputs

Running the `syn-to-timing` action after running synthesis will automatically generate the Hammer IR required to pipe the synthesis outputs to the Hammer STA tool, and should be included in the Hammer call, as demonstrated in the "Post-Synthesis STA" command below. The same goes for post-place-and-route STA.

4.9.4 STA Outputs

The STA tool produces reports in `OBJ_DIR/timing-rundir/`. Outputs from advanced STA flows such as engineering change order (ECO) patches are not yet supported.

4.9.5 STA Commands

- Synthesis to STA
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-rundir/syn-output.json -o OBJ_DIR/syn-to-timing_input.json --obj_dir OBJ_DIR syn-to-timing`
- Post-Synthesis STA
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/syn-to-timing_input.json --obj_dir OBJ_DIR timing`
- P&R to STA
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-rundir/par-output.json -o OBJ_DIR/par-to-timing_input.json --obj_dir OBJ_DIR par-to-timing`
- Post-P&R STA
  - `hammer-vlsi -e env.yml -p config.yml -p OBJ_DIR/par-to-timing_input.json --obj_dir OBJ_DIR timing`

CHAPTER

FIVE

HAMMER USE

This documentation will walk through more advanced features of the Hammer infrastructure. You will learn about Hammer's APIs, flow control, how to write hooks, the supported build infrastructure, and how to set up hierarchical flows.

5.1 Hammer IR and Meta Variables

5.1.1 Hammer IR

Hammer IR is the primary standardized data exchange format of Hammer. Hammer IR standardizes physical design constraints such as placement constraints and clock constraints. In addition, the Hammer IR also standardizes communication among and to Hammer plugins, including tool control (e.g. loading tools, etc) and configuration options (e.g. number of CPUs).

5.1.2 The hammer-config library

The hammer-config library is the part of Hammer responsible for parsing Hammer YAML/JSON configuration files into Hammer IR. Hammer IR is used for the standardization and interchange of data between the different parts of Hammer and Hammer plugins.

Note: There is a built-in order of precedence, from lowest to highest:

1) Hammer's `defaults.yml`
2) Tool plugin's `defaults.yml`
3) Tech plugin's `defaults.yml`
4) User's Hammer IR (using `-p` on the command line)

JSON/YAML files specified with `-p` later on the command line have higher precedence and keys appearing later if duplicated in a given file also take precedence. In the examples below, "Level #" will be used to denote the level of precedence of the configuration snippet.

The `get_setting()` method is available to all Hammer technology/tool plugins and hooks (see *Extending Hammer with Hooks*).

5.1.3 Basics

```
foo:
    bar:
        adc: "yes"
        dac: "no"
```

The basic idea of Hammer IR in YAML/JSON format is centered around a hierarchically nested tree of YAML/JSON dictionaries. For example, the above YAML snippet is translated to two variables which can be queried in code - `foo.bar.adc` would have `yes` and `foo.bar.dac` would have `no`.

5.1.4 Overriding

Hammer IR snippets frequently "override" each other. For example, a technology plugin might provide some defaults which a specific project can override with a project YAML snippet.

For example, if the base snippet contains `foo: 12345` and a snippet in a file of higher precedence contains `foo: 54321`, then `get_setting("foo")` would return `54321`.

5.1.5 Meta actions

Sometimes it is desirable that variables are not completely overwritten, but instead modified.

For example, say that the technology plugin provides:

```
vlsi.tech.foobar65.bad_cells: ["NAND4X", "NOR4X"]
```

And let's say that in our particular project, we find it undesirable to use the `NAND2X` and `NOR2X` cells. However, if we simply put the following in our project YAML, the references to `NAND4X` and `NOR4X` disappear and we don't want to have to copy the information from the base plugin, which may change, or which may be proprietary, etc.

```
vlsi.tech.foobar65.bad_cells: ["NAND2X", "NOR2X"]
```

The solution is **meta variables**. This lets Hammer's config parser know that instead of simply replacing the base variable, it should do a particular special action. Any config variable can have `_meta` suffixed into a new variable with the desired meta action.

In this example, we can use the `append` meta action:

```
vlsi.tech.foobar65.bad_cells: ["NAND2X", "NOR2X"]
vlsi.tech.foobar65.bad_cells_meta: append
```

This will yield the desired result of `["NAND4X", "NOR4X", "NAND2X", "NOR2X"]` when `get_setting("vlsi.tech.foobar65.bad_cells")` is called in the end.

5.1.6 Applying multiple meta actions

Multiple meta actions can be applied sequentially if the `_meta` variable is an array. Example:

In Level 1:

```
foo.flash: yes
```

In Level 2 (located at /opt/foo):

```
foo.pipeline: "CELL_${foo.flash}.lef"
foo.pipeline_meta: ['subst', 'prependlocal']
```

Result: `get_setting("foo.pipeline")` returns `/opt/foo/CELL_yes.lef`.

5.1.7 Common meta actions

- `append`: append the elements provided to the base list. (See the above `vlsi.tech.foobar65.bad_cells` example.)
- `prepend`: prepend the elements provided to the base list. Useful where list order matters, but make sure the file containing this directive is at the end of the list of files passed to `hammer-vlsi` with `-p`.
- `subst`: substitute variables into a string.

  Base:

  ```
  foo.flash: yes
  ```

  Meta:

  ```
  foo.pipeline: "${foo.flash}man"
  foo.pipeline_meta: subst
  ```

  Result: `get_setting("foo.flash")` returns `yesman`
- `lazysubst`: by default, variables are only substituted from previous configs. Using `lazysubst` allows us to defer the substitution until the very end.

  Example without `lazysubst`:

  Level 1:

  ```
  foo.flash: yes
  ```

  Level 2:

  ```
  foo.pipeline: "${foo.flash}man"
  foo.pipeline_meta: subst
  ```

  Level 3:

  ```
  foo.flash: no
  ```

  Result: `get_setting("foo.flash")` returns `yesman`

  Example with `lazysubst`:

  Level 1:

```
foo.flash: yes
```

Level 2:

```
foo.pipeline: "${foo.flash}man"
foo.pipeline_meta: lazysubst
```

Level 3:

```
foo.flash: no
```

Result: `get_setting("foo.flash")` returns `noman`

In general, putting `lazy` in front of all other directives will defer the meta processing until the very end.

- `crossref` - directly reference another setting. Example:

  Level 1:

  ```
  foo.flash: yes
  ```

  Level 2:

  ```
  foo.mob: "foo.flash"
  foo.mob_meta: crossref
  ```

  Result: `get_setting("foo.mob")` returns `yes`

  You can also append/prepend instead of substitute using `crossappendref` and `crossprependref`.

- `transclude` - transclude the given path. Example:

  Level 1:

  ```
  foo.bar: "/opt/foo/myfile.txt"
  foo.bar_meta: transclude
  ```

  Result: `get_setting("foo.bar")` returns `<contents of /opt/foo/myfile.txt>`

- `prependlocal` - prepend the local path or package resource directory of this config file. Example:

  Level 1 (located at /opt/foo):

  ```
  foo.bar: "myfile.txt"
  foo.bar_meta: prependlocal
  ```

  Result: `get_setting("foo.mob")` returns `/opt/foo/myfile.txt`

- `deepsubst` - like `subst` but descends into sub-elements. Example:

  Level 1:

  ```
  foo.bar: "123"
  ```

  Level 2:

  ```
  foo.bar:
    baz: "${foo.bar}45"
    quux: "32${foo.bar}"
  foo.bar_meta: deepsubst
  ```

Result: `get_setting("foo.bar.baz")` returns `12345` and `get_setting("foo.bar.baz")` returns `32123`

5.1.8 Type Checking

Any existing configuration file can and should be accompanied with a corresponding configuration types file. This allows for static type checking of any key when calling `get_setting`. The file should contain the same keys as the corresponding configuration file, but can contain the following as values:

- primitive types (`int`, `str`, etc.)
- collection types (`list`)
- collections of key-value pairs (`list[dict[str, str]]`, `list[dict[str, list]]`, etc.) These values are turned into custom constraints (e.g. `PlacementConstraint`, `PinAssignment`) later in the Hammer workflow, but the key value pairs are not type-checked any deeper.
- optional forms of the above (`Optional[str]`)
- the wildcard `Any` type

Hammer will perform the same without a types file, but it is highly recommended to ensure type safety of any future plugins.

5.1.9 Key History

With the `ruamel.yaml` package, Hammer can emit what files have modified any configuration keys in YAML format. Turning on key history is accomplished with the `--dump-history` command-line flag. The file is named `{action}-output-history.yml` and is located in the output folder of the given action.

Example with the file `test-config.yml`:

```
synthesis.inputs:
    input_files: ["foo", "bar"]
    top_module: "z1top.xdc"

vlsi:
    core:
        technology: "hammer.technology.nop"

        synthesis_tool: "hammer.synthesis.nop"
```

`test/syn-rundir/syn-output-history.yml` after executing the command `hammer-vlsi --dump-history -p test-config.yml --obj_dir test syn`:

```
synthesis.inputs.input_files:  # Modified by: test-config.yml
  - LICENSE
  - README.md
synthesis.inputs.top_module: z1top.xdc # Modified by: test-config.yml

vlsi.core.technology: nop # Modified by: test-config.yml
vlsi.core.synthesis_tool: hammer.synthesis.nop # Modified by: test-config.yml
```

Example with the files `test-config.yml` and `test-config2.yml`, respectively:

```
synthesis.inputs:
    input_files: ["foo", "bar"]
    top_module: "z1top.xdc"

vlsi:
    core:
        technology: "hammer.technology.nop"

        synthesis_tool: "hammer.synthesis.nop"
```

```
par.inputs:
    input_files: ["foo", "bar"]
    top_module: "z1top.xdc"

vlsi:
    core:
        technology: "${foo.subst}"

        par_tool: "hammer.par.nop"

foo.subst: "hammer.technology.nop2"
```

`test/syn-rundir/par-output-history.yml` after executing the command `hammer-vlsi --dump-history -p test-config.yml -p test-config2.yml --obj_dir test syn-par`:

```
foo.subst: hammer.technology.nop2 # Modified by: test-config2.yml
par.inputs.input_files:  # Modified by: test-config2.yml
  - foo
  - bar
par.inputs.top_module: z1top.xdc # Modified by: test-config2.yml
synthesis.inputs.input_files:  # Modified by: test-config.yml
  - foo
  - bar
synthesis.inputs.top_module: z1top.xdc # Modified by: test-config.yml
vlsi.core.technology: hammer.technology.nop2 # Modified by: test-config.yml,␣
↪test-config2.yml
vlsi.core.synthesis_tool: hammer.synthesis.nop # Modified by: test-config.yml
vlsi.core.par_tool: hammer.par.nop # Modified by: test-config2.yml
```

5.1.10 Key Description Lookup

With the `ruamel.yaml` package, Hammer can execute the `info` action, allowing users to look up the description of most keys. The comments must be structured like so in order to be read properly:

```
foo: bar # this is a comment
# this is another comment for the key "foo"
```

Hammer will take the descriptions from any `defaults.yml` files.

Running `hammer-vlsi -p test-config.yml info` (assuming the above configuration is in `defaults.yml`):

```
foo.bar
Select from the current level of keys: foo.bar

apple
price
Select from the current level of keys: apple
----------------------------------------
Key: foo.bar.apple
Value: banana
Description: # type of fruit
History: ["defaults.yml"]
----------------------------------------

Continue querying keys? [y/n]: y

foo.bar
Select from the current level of keys: foo.bar

apple
price
Select from the current level of keys: price
----------------------------------------
Key: foo.bar.price
Value: 2
Description: # price of fruit
History: ["defaults.yml"]
----------------------------------------
```

Keys are queried post-resolution of all meta actions, so their values correspond to the project configuration after other actions like `syn` or `par`.

5.1.11 Reference

For a more comprehensive view, please consult the `hammer.config` API documentation in its implementation here:

- https://github.com/ucb-bar/hammer/blob/master/hammer/config/config_src.py
- https://github.com/ucb-bar/hammer/blob/master/tests/test_config.py

In `config_src.py`, most supported meta actions are contained in the `directives` list of the `get_meta_directives` method.

5.2 Hammer APIs

Hammer has a growing collection of APIs that use objects defined by the technology plugin, such as stackups and special cells. They expose useful extracted information from Hammer IR to other methods, such as in tool plugins that will implement this information in a tool-compatible manner.

For syntax details about the Hammer IR needed to use these APIs, refer to the defaults.yml.

5.2.1 Power Specification

Simple power specs are specified using the Hammer IR key `vlsi.inputs.supplies`, which is then translated into a a Supply object. `hammer_vlsi_impl` exposes the Supply objects to other APIs (e.g. power straps) and can generate the CPF/UPF files depending on which specification the tools support. Multi-mode multi-corner (MMMC) setups are also available by setting `vlsi.inputs.mmmc_corners` and manual power spec definitions are supported by setting the relevant `vlsi.inputs.power_spec...` keys.

5.2.2 Timing Constraints

Clock and pin timing constraints are specified using the Hammer IR keys `vlsi.inputs.clocks/output_loads/delays`. These objects can be turned into SDC-style constraints by `hammer_vlsi_impl` for consumption by supported tools.

5.2.3 Floorplan & Placement

Placement constraints are specified using the Hammer IR key `vlsi.inputs.placement_constraints`. These constraints are very flexible and have varying inputs based on the type of object the constraint applies to, such as hierarchical modules, hard macros, or obstructions. At minimum, an `(x, y)` coordinate corresponding to the lower left corner must be given, and additional parameters such as width/height, margins, layers, or orientation are needed depending on the type of constraint. Place-and-route tool plugins will take this information and emit the appropriate commands during floorplanning. Additional work is planned to ensure that floorplans are always legal (i.e. on grid, non-overlapping, etc.).

All Hammer tool instances have access to a method that can produce graphical visualization of the floorplan as an SVG file, viewable in a web browser. To use it, call the `generate_visualization()` method from any custom hook (see *Extending Hammer with Hooks*). The options for the visualization tool are in the Hammer IR key `vlsi.inputs.visualization`.

5.2.4 Bumps

Bump constraints are specified using the Hammer IR key `vlsi.inputs.bumps`. Rectangular-gridded bumps are supported, although bumps at fractional coordinates in the grid and deleted bumps are allowed. The place-and-route tool plugin translates the list of bump assignments into the appropriate commands to place them in the floorplan and enable flip-chip routing. The bumps API is also used by the PCB plugin to emit the collateral needed by PCB layout tools such as Altium Designer. This API ensures that the bumps are always in correspondence between the chip and PCB.

The visualization tool mentioned above can also display bump placement and assignments. There are options to view the bumps from the perspective of the ASIC designer or the PCB designer. The views are distinguishable by a reference dot displayed in the left and right corners for the ASIC and PCB perspectives, respectively.

5.2.5 Pins

Pin constraints are specified using the Hammer IR key `vlsi.inputs.pin`. PinAssignments objects are generated and passed to the place-and-route tool to place pins along specified block edges on specified metal layers. Preplaced (e.g. hard macros in hierarchical blocks) pins are also supported so that they are not routed. Additional work is planned to use this API in conjunction with the placement constraints API to allow for abutment of hierarchical blocks, which requires pins to be aligned on abutting edges.

5.2.6 Power Straps

Power strap constraints are specified using multiple Hammer IR keys in the `par` namespace.You can find the keys in <tech>/defaults.yml under the tech plugin directory. An example from asap7 is as follows:

Listing 1: ASAP7 default power straps setting

```
par.power_straps_mode: generate # Power straps (most DRC-clean)
par.generate_power_straps_method: by_tracks
par.generate_power_straps_options:
  by_tracks:
    strap_layers:
      - M3
      - M4
      - M5
      - M6
      - M7
      - M8
      - M9
    pin_layers:
      - M9
    track_width: 7 # minimum allowed for M2 & M3
    track_spacing: 0
    track_spacing_M3: 28 # space straps apart evenly, in conjunction w/ track_
↪utilization_M3
    track_start: 10
    power_utilization: 0.25
    power_utilization_M3: 0.6 # together with track_spacing_M3 results in approx. 0.25␣
↪eff. utilization
    power_utilization_M8: 1.0
    power_utilization_M9: 1.0
```

The default keys for all hammer configs are defined in the defaults.yml, which contains detailed comments on what each key does. Here is the default setting and parameter descriptions for power strap generation.

Listing 2: Hammer global default power straps setting

```
  # Used by floorplanning.
  # Overridden by individual placement constraints on top_layer.
  # type: Optional[str]

  generate_power_straps_method: "by_tracks" # If power_straps_mode is 'generate', which␣
↪method to use.
  # Currently, the valid options are:
  # - by_tracks - Specify the power strap plan per layer in terms of tracks

  generate_power_straps_options:
  # Default settings for power strap generation modes
  # Keys are the valid values of the 'generate_power_straps_method' key above
    by_tracks:
      blockage_spacing: "par.blockage_spacing" # Spacing from end of strap to a block or␣
↪blockage
      # Overrideable by appending _<layer name>
      # type: Decimal
```

(continues on next page)

(continued from previous page)

```
    blockage_spacing_meta: lazycrossref

    track_width: 5 # Number of routing tracks to be consumed by an individual strap
    # Overrideable by appending _<layer name>
    # type: int

    track_start: 0 # The first track to contain a power stripe
    # Overrideable by appending _<layer name>
    # type: int
    # TODO(johnwright): include an auto-center option

    track_offset: 0.0 # Absolute offset for straps relative to the design bounding box␣
↪origin
    # Overrideable by appending _<layer name>
    # type: Decimal

    track_spacing: 0 # Number of routing tracks between sets of straps (0 is␣
↪recommended)
    # Overrideable by appending _<layer name>
    # type: int

    power_utilization: 0.1 # Ratio of total routing tracks to dedicate to power straps,
↪ which is used to calculate set pitch
    # Overrideable by appending _<layer name>
    # type: float

    antenna_trim_shape: stripe # "none" or "stripe", specifies antenna trimming␣
↪strategy
```

The currently supported API supports power strap generation by tracks, which auto-calculates power strap width, spacing, set-to-set distance, and offsets based on basic DRC rules specified in the technology Stackup object.

The technology Stackup information ("stackups") can be found in the <tech>.tech.json file under the tech plugin directory. The "stackups" usually are located near the end of the <tech> `.tech.json` file. An example from asap7 is as follows:

Listing 3: ASAP7 stackup object

```
  {
    "name": "asap7_3Ma_2Mb_2Mc_2Md",
    "grid_unit": 0.001,
    "metals": [
      {"name": "M1", "index": 1, "direction": "horizontal", "min_width": 0.072, "pitch
↪": 0.144, "offset": 0.0, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.072}], "power_strap_width_table": [], "grid_unit": 0.001},
      {"name": "M2", "index": 2, "direction": "horizontal", "min_width": 0.072, "pitch
↪": 0.144, "offset": -1.08, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.072}], "power_strap_width_table": [0.072, 0.36, 0.648, 0.936, 1.224,␣
↪1.512], "grid_unit": 0.001},
      {"name": "M3", "index": 3, "direction": "vertical", "min_width": 0.072, "pitch":␣
↪0.144, "offset": 0.0, "power_strap_widths_and_spacings": [{"width_at_least": 0.0, "min_
↪spacing": 0.072}], "power_strap_width_table": [0.072, 0.36, 0.648, 0.936, 1.224, 1.
↪512], "grid_unit": 0.001},
```

(continues on next page)

(continued from previous page)

```
        {"name": "M4", "index": 4, "direction": "horizontal", "min_width": 0.096, "pitch
↪": 0.192, "offset": 0.048, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.096}, {"width_at_least": 0.1, "min_spacing": 0.288}], "power_strap_
↪width_table": [0.096, 0.48, 0.864, 1.248, 1.632], "grid_unit": 0.001},
        {"name": "M5", "index": 5, "direction": "vertical", "min_width": 0.096, "pitch":␣
↪0.192, "offset": 0.048, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.096}, {"width_at_least": 0.1, "min_spacing": 0.288}], "power_strap_
↪width_table": [0.096, 0.48, 0.864, 1.248, 1.632, 2.016, 2.4, 2.784, 3.168, 3.552, 3.
↪936], "grid_unit": 0.001},
        {"name": "M6", "index": 6, "direction": "horizontal", "min_width": 0.128, "pitch
↪": 0.256, "offset": 0.064, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.128}, {"width_at_least": 0.1, "min_spacing": 0.288}], "power_strap_
↪width_table": [0.128, 0.64, 1.152, 1.664, 2.176], "grid_unit": 0.001},
        {"name": "M7", "index": 7, "direction": "vertical", "min_width": 0.128, "pitch":␣
↪0.256, "offset": 0.064, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.128}, {"width_at_least": 0.1, "min_spacing": 0.288}], "power_strap_
↪width_table": [0.128, 0.64, 1.152, 1.664, 2.176], "grid_unit": 0.001},
        {"name": "M8", "index": 8, "direction": "horizontal", "min_width": 0.16, "pitch
↪": 0.32, "offset": 0.16, "power_strap_widths_and_spacings": [{"width_at_least": 0.0,
↪"min_spacing": 0.16}, {"width_at_least": 0.239, "min_spacing": 0.24}, {"width_at_least
↪": 0.319, "min_spacing": 0.32}, {"width_at_least": 0.479, "min_spacing": 0.48}, {
↪"width_at_least": 1.999, "min_spacing": 2.0}, {"width_at_least": 3.999, "min_spacing":␣
↪4.0}], "power_strap_width_table": [], "grid_unit": 0.001},
        {"name": "M9", "index": 9, "direction": "vertical", "min_width": 0.16, "pitch":␣
↪0.32, "offset": 0.16, "power_strap_widths_and_spacings": [{"width_at_least": 0.0, "min_
↪spacing": 0.16}, {"width_at_least": 0.239, "min_spacing": 0.24}, {"width_at_least": 0.
↪319, "min_spacing": 0.32}, {"width_at_least": 0.479, "min_spacing": 0.48}, {"width_at_
↪least": 1.999, "min_spacing": 2.0}, {"width_at_least": 3.999, "min_spacing": 4.0}],
↪"power_strap_width_table": [], "grid_unit": 0.001},
        {"name": "Pad", "index": 10, "direction": "redistribution", "min_width": 0.16,
↪"pitch": 8.16, "offset": 0.0, "power_strap_widths_and_spacings": [{"width_at_least": 0.
↪0, "min_spacing": 8.0}, {"width_at_least": 47.999, "min_spacing": 12.0}], "power_strap_
↪width_table": [], "grid_unit": 0.001}
      ]
    }
  ],
  "sites": [
```

The keys in the Stackup object are defined in stackup.py as follows.

Listing 4: Description for a metal layer/stackup

```
    name: Metal layer name (e.g. M1, M2).
    index: The order in the stackup (lower is closer to the substrate).
    direction: The preferred routing direction of this metal layer, or
               RoutingDirection.Redistribution for non-routing top-level
               redistribution metals like Aluminium.
    min_width: The minimum wire width for this layer.
    max_width: The maximum wire width for this layer.
    pitch: The minimum cross-mask pitch for this layer (NOT same-mask pitch
           for multiple-patterned layers). Width of routing grid for a given layer.
           To route denser wires on chip, multiple masks are required.
```

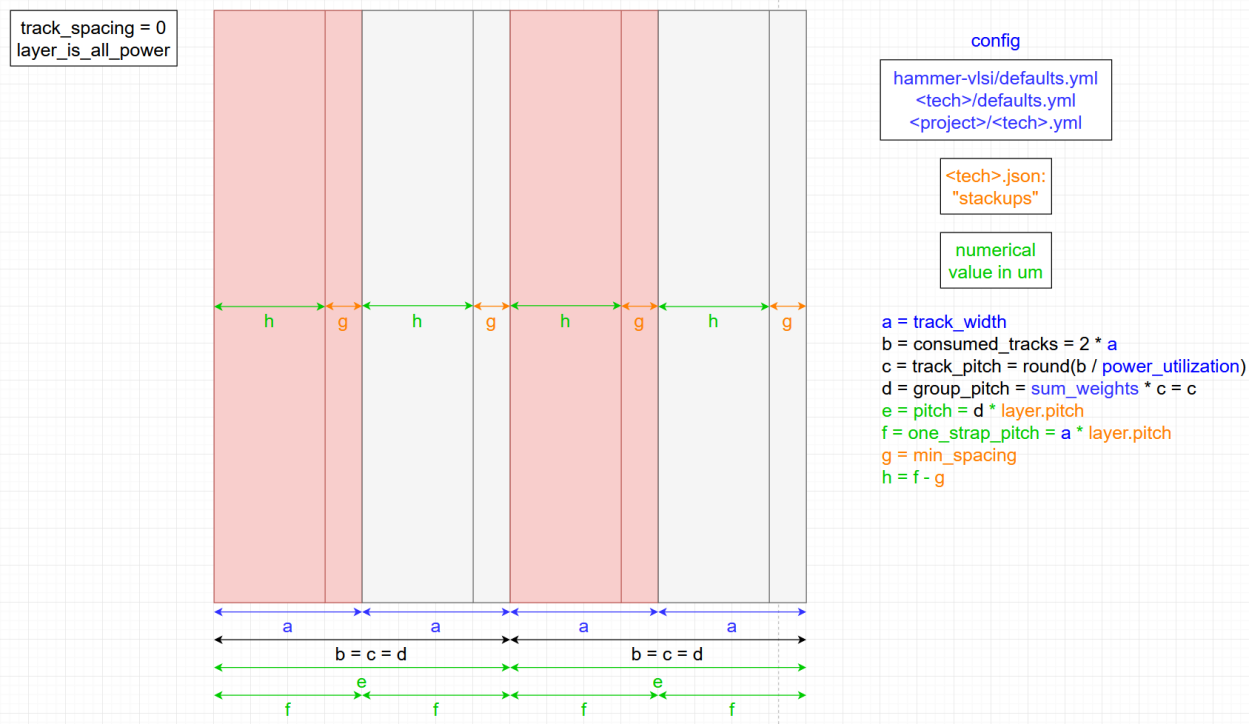
(continues on next page)

(continued from previous page)

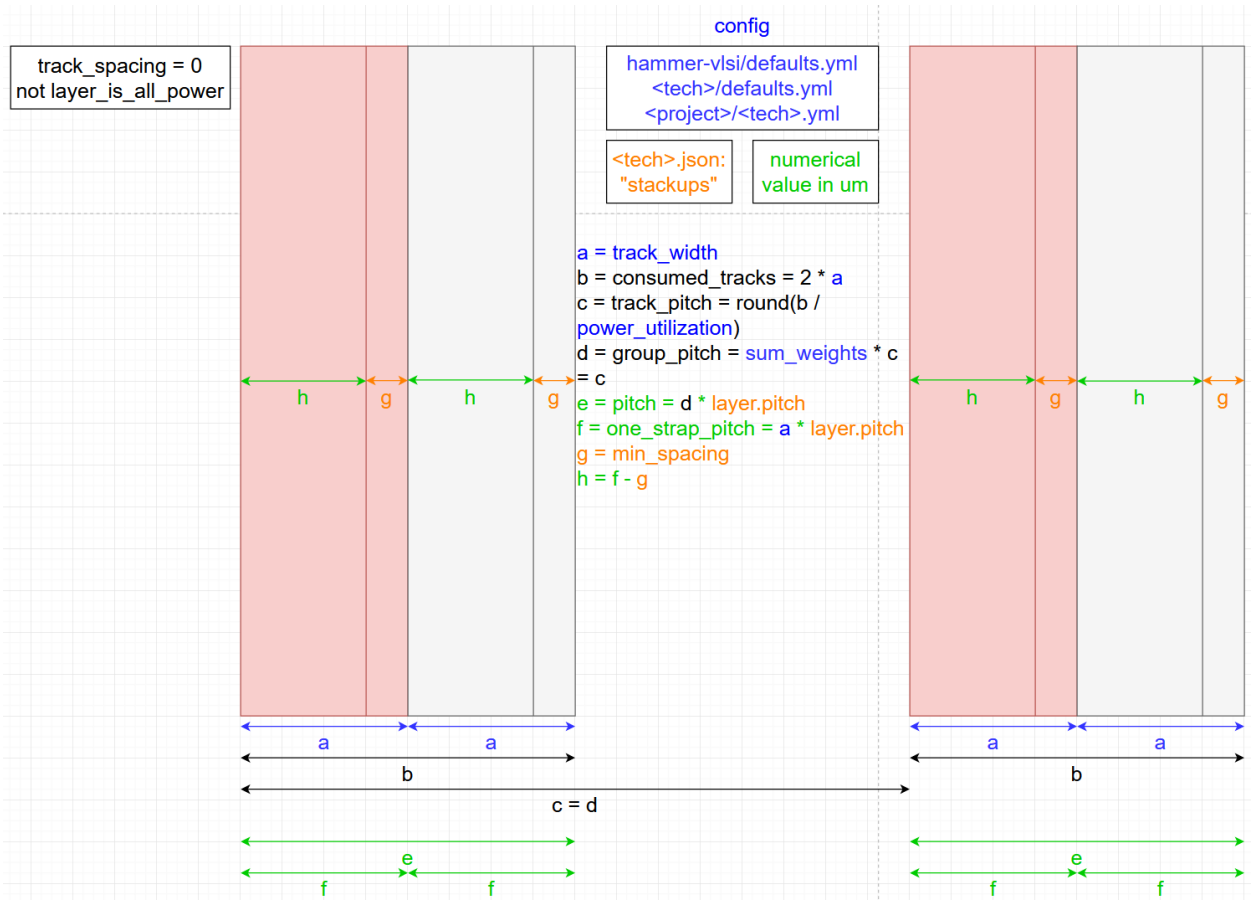
```
          During fabrication, the masks are applied separately with some spatial offsets
          to achieve denser line patterning. For more information on multiple-patterning,
          check https://en.wikipedia.org/wiki/Multiple_patterning
   offset: The routing track offset from the origin for the first track in this layer.
           (0 = first track is on an axis).
   power_strap_widths_and_spacings: A list of WidthSpacingTuples that specify the minimum
                                    spacing rules for an infinitely long wire of variying width.
   power_strap_width_table: A list of allowed metal widths in the technology.
                            Widths smaller than the last number must be quantized to a value in the table.
   grid_unit: The fixed-point decimal value of a minimum grid unit (e.g. 1nm = 0.001).
```

The basic pieces of information needed are the desired track utilization per strap and overall power strap density. Powerstraps are routed in pairs of Vdd and Vss. Based on the effective power utilization and track spacing, there are three ways to route powerstraps.

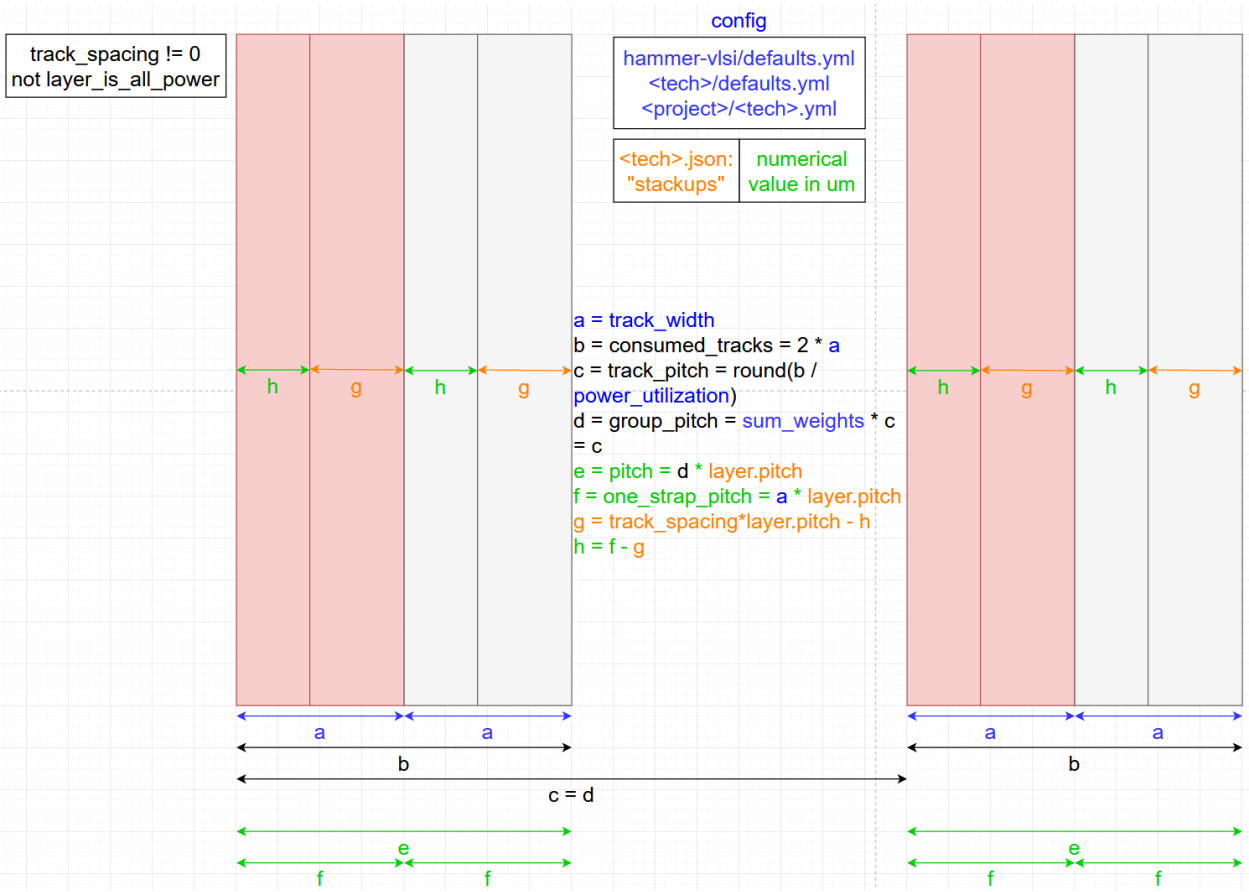
For track spacing = 0 and effective power utilization = 100%, powerstraps are routed as follows.



For track spacing = 0 and effective power utilization < 100%, powerstraps are routed as follows.



For track spacing > 0 and effective power utilization < 100%, powerstraps are routed as follows.



The currently supported API supports power strap generation by tracks, which auto-calculates power strap width, spacing, set-to-set distance, and offsets based on basic DRC rules specified in the technology Stackup object. The basic pieces of information needed are the desired track utilization per strap and overall power strap density. Different values can be specified on a layer-by-layer basis by appending `_<layer name>` to the end of the desired option.

5.2.7 Special Cells

Special cells are specified in the technology's JSON, but are exposed to provide lists of cells needed for certain steps, such as for fill, well taps, and more. Synthesis and place-and-route tool plugins can grab the appropriate type of special cell for the relevant steps.

5.2.8 Submission

Each tool has run submission options given by the Hammer IR key `<tool type>.submit`. Using the `command` and `settings` keys, a setup for LSF or similar workload management platforms can be standardized.

5.3 Flow Control

Physical design is necessarily an iterative process, and designers will often require fine control of the flow within any given action. This allows for rapid testing of new changes in the Hammer IR to improve the quality of results (QoR) after certain steps.

Given the flow defined by tool steps and hooks (described in the next section), users can select ranges of these to run. The tool script will then be generated with commands corresponding to only the steps that are to be run.

5.3.1 Command-line Interface

Flow control is specified with the following optional Hammer command-line flags, which must always target a valid step/hook:

- `--start_before_step <target>`: this starts the tool from (inclusive) the target step. Alternate flag: `--from_step`
- `--start_after_step <target>`: this starts the tool from (exclusive) the target step. Alternate flag: `--after_step`
- `--stop_after_step`: this stops the tool at (inclusive) the target step. Alternate flag: `--to_step`
- `--stop_before_step`: this stops the tool at (exclusive) the target step. Alternate flag: `--until_step`
- `--only_step`: this only runs the target step

As `hammer-vlsi` is parsing through the steps for a given tool, it will print debugging information indicating which steps are skipped and which are run.

Certain combinations are not allowed:

- `--only_step` is not compatible with any of the other flags
- `--start_before_step` and `start_after_step` may not be specified together
- `--stop_after_step` and `--stop_before_step` may not be specified together
- `--start_before_step` and `--stop_before_step` may not be the same step
- `--start_after_step` and `--stop_after_step` may not be the same step
- `--start_after_step` and `--stop_before_step` may not be the same or adjacent steps

Logically, a target of `stop_{before|after}_step` before `start_{before|after}_step` will also not run anything (though it is not explicitly checked).

5.4 Extending Hammer with Hooks

It is unlikely that using the default Hammer APIs alone will produce DRC- and LVS-clean designs with good QoR in advanced technology nodes if the design is sufficiently complex. To solve that, Hammer is extensible using *hooks*. These hooks also afford power users additional flexibility to experiment with CAD tool commands to tweak aspects of their designs. The hook framework is inherently designed to enable reusability, because successful hook methods that solve a technology-specific problem may be upstreamed into the technology plugin for future designs.

5.4.1 Hook Methods

Hooks are fundamentally Python methods that extend a given tool's set of available steps and can inject additional TCL commands into the flow. Hook methods need to take in an instance of a particular `HammerTool`, which provides them with the full set of Hammer IR available to the tool. Hooks (depending on how they are included, see below) get turned into step objects that can be targeted with `--from/after/to/until_step` and other hooks.

Hooks can live in a Python file inside the design root so that it is available to the class that needs to extend the default `CLIDriver`. An example of some skeletons of hooks are found in Chipyard. For more comprehensive examples, refer to the hooks unit tests in the `TestHammerToolHooks` class of test_hooks.py.

5.4.2 Including Hooks

Hooks modify the flow using a few `HammerTool` methods, such as:

- `make_replacement_hook(<target>, <hook_method>)`: this swaps out an existing target step/hook with the hook method
- `make_pre_insertion_hook(<target>, <hook_method>)`: this inserts the hook method before the target step/hook
- `make_post_insertion_hook(<target>, <hook_method>)`: this inserts the hook method after the target step/hook
- `make_removal_hook(<target>)`: this removes the target step/hook from the flow

Note: `<target>` should be a string (name of step/hook), while `<hook_method>` is the hook method itself. All of the hook methods specified this way are targetable by other hooks.

Sometimes, CAD tools do not save certain settings into checkpoint databases. As a result, when for example a `--from_step` is called, the setting will not be applied when the database from which to continue the flow is read in. To get around this, a concept of "persistence" is implemented with the following methods:

- `make_persistent_hook(<hook_method>)`: this inserts the hook method at the beginning of any tool invocation, regardless of which steps/hooks are run
- `make_pre_persistent_hook(<target>, <hook_method>)`: this inserts the hook method at the beginning of any tool invocation, as long as the target step/hook is located at or after the first step to be run
- `make_post_persistent_hook(<target>, <hook_method>)`: this inserts the hook method at the beginning of any tool invocation if the target step/hook is before the first step to be run, or right after the target step/hook if that step/hook is within the steps to be run.

All persistent hooks are NOT targetable by flow control options, as their invocation location may vary. However, persistent hooks ARE targetable by `make_replacement/pre_insertion/post_insertion/removal_hook`. In this case, the hook that replaces or is inserted pre/post the target persistent hook takes on the persistence properties of the target persistence hook.

Some examples of these methods are found in the Chipyard example, linked above.

A list of these hooks must be provided in an implementation of method such as `get_extra_par_hooks` in the command-line driver. This new file becomes the entry point into Hammer, overriding the default `hammer-vlsi` executable.

5.4.3 Technology, Tool, and User-Provided Hooks

Note: Hooks may be provided by the technology plugin, the tool plugin, and/or the user. The order of step & hook priority is as follows, from lowest to highest:

1. Technology default steps
2. Technology plugin hooks
3. Tool plugin hooks
4. User hooks

A technology plugin specifies hooks in its `__init__.py` (as a method inside its subclass of `HammerTechnology`). It should implement a `get_tech_<action>_hooks(self, tool_name: str)` method. The tool name parameter may be checked by the hook implementation because multiple tools may implement the same action. Technology plugin hooks may only target tool default steps to insert/replace.

The included ASAP7 technology plugin provides an example of how to inject two different types of hooks: 1) a persistent hook invoked anytime after the `init_design` step to set top & bottom routing layers, and 2) two post-insertion hooks, one to modify the floorplan for DRCs and the other to scale down a GDS post-place-and-route using the `gdstk` or `gdspy` GDS manipulation utilities. Note that the persistent hook that is included does not necessarily need to be persistent (Innovus does retain this setting in snapshot databases), but it serves as an example for building your own tech plugin.

A tool plugin specifies hooks in its `__init__.py` (as a method inside its subclass of `HammerTool`). It should implement a `get_tool_hooks(self)` method. In contrast to the tech-supplied hooks, the action name and tool name are not specified because a tool instance can only correspond to a single action. Tool plugin hooks may target its own default steps and technology plugin hooks to insert/replace.

A user specifies hooks in the command-line driver and should implement a `get_extra_<action>_hooks(self)` method. User hooks may target tool default steps, technology plugin hooks, and tool plugin hooks to insert/replace. A good example is the `example-vlsi` file in the Chipyard example, which implements a `get_extra_par_hooks(self)` method that returns a list of hook inclusion methods.

The priority means that if both the technology and user specify persistent hooks, any duplicate commands in the user's persistent hook will override those from the technology's persistent hook.

5.5 Flowgraphs

Hammer has **experimental** support for flowgraph constructions, similar to tools like mflowgen. Their intention is to simplify the way flows are constructed and ran in Hammer. They can be imported via the `hammer.flowgraph` module.

5.5.1 Construction

Flowgraphs are nothing more than a collection of `Node` instances linked together via a `Graph` instance. Each `Node` "pulls" from a directory to feed in inputs and "pushes" output files to another directory to be used by other nodes. `Node` instances are roughly equivalent to a single call to the `hammer-vlsi` CLI, so they take in similar attributes:

- The action being called
- The tool used to perform the action
- The pull and push directories
- Any *required* input/output files
- Any *optional* input/output files
- A driver to run the node with; this enables backwards compatibility with *hooks*.
- Options to specify steps within an action; this enables backwards compatibility with *flow control*.
  - `start_before_step`
  - `start_after_step`
  - `stop_before_step`
  - `stop_after_step`
  - `only_step`

A minimal example of a `Node`:

```
from hammer.flowgraph import Node

test = Node(
    action="foo",
    tool="nop",
    pull_dir="foo_dir",
    push_dir="/dev/null",
    required_inputs=["foo.yml"],
    required_outputs=["foo-out.json"],
)
```

Each `Node` has the ability to be "privileged", meaning that a flow *must* start with this node. This only occurs when a flow is being controlled using any of the steps.

5.5.2 Running a Flowgraph

`Node` instances are linked together using an adjacency list. This list can be used to instantiate a `Graph`:

```
from hammer.flowgraph import Graph, Node

root = Node(
    "foo", "nop", "foo_dir", "",
    ["foo.yml"],
    ["foo-out.json"],
)
child1 = Node(
    "bar", "nop", "foo_dir", "bar_dir",
```

(continues on next page)

(continued from previous page)

```
    ["foo-out.json"],
    ["bar-out.json"],
)
graph = Graph({root: [child1]})
```

Using the Hammer CLI tool, separate actions are manually linked via an *auxiliary* action, such as `syn-to-par`. By using a flowgraph, `Graph` instances by default *automatically* insert auxiliary actions. This means that actions no longer need to be specified in a flow; the necessary nodes are inserted by the flowgraph tool. This feature can be disabled by setting `auto_auxiliary=False`.

A `Graph` can be run by calling the `run` method and passing in a starting node. When running a flow, each `Node` keeps an internal status based on the status of the action being run:

- `NOT_RUN`: The action has yet to be run.
- `RUNNING`: The action is currently running.
- `COMPLETE`: The action has finished.
- `INCOMPLETE`: The action ran into an error while being run.
- `INVALID`: The action's outputs have been invalidated (e.g. inputs or attributes have changed).

The interactions between the statuses are described in the diagram:

Regardless of whether a flow completes with or without errors, the graph at the time of completion or error is returned, allowing for a graph to be "resumed" once any errors have been fixed.

5.5.3 Visualization

A flowgraph can be visualized in Markdown files via the Mermaid tool. Calling a `Graph` instance's `to_mermaid` method outputs a file named `graph-viz.md`. The file can be viewed in a site like Mermaid's live editor or using Github's native support.

The flowgraph below would appear like this:

```
from hammer.flowgraph import Graph, Node

syn = Node(
    "syn", "nop",
    os.path.join(td, "syn_dir"), os.path.join(td, "s2p_dir"),
    ["syn-in.yml"],
    ["syn-out.json"],
)
s2p = Node(
    "syn-to-par", "nop",
    os.path.join(td, "s2p_dir"), os.path.join(td, "par_dir"),
    ["syn-out.json"],
    ["s2p-out.json"],
)
par = Node(
    "par", "nop",
    os.path.join(td, "par_dir"), os.path.join(td, "out_dir"),
    ["s2p-out.json"],
    ["par-out.json"],
)
```

(continues on next page)

(continued from previous page)

```
g = Graph({
    syn: [s2p],
    s2p: [par],
    par: []
})
```

Here are the contents of `graph-viz.md` after calling `g.to_mermaid()`:

````
```mermaid

stateDiagram-v2
    syn --> syn_to_par
    syn_to_par --> par
```
````

Which would render like this:

Note that the separators have been changed to comply with Mermaid syntax.

5.6 Hammer Buildfile

Hammer natively supports a GNU Make-based build system to manage build dependencies. To use this flow, `vlsi.core.build_system` must be set to `make`. Hammer will generate a Makefile include in the object directory named `hammer.d` after calling the `build` action:

```
hammer-vlsi -e env.yml -p config.yml --obj_dir build build
```

`build/hammer.d` will contain environment variables needed by Hammer and a target for each major Hammer action (e.g. `par`, `synthesis`, etc. but not `syn-to-par`, which is run automatically when calling `make par`). For a flat design, the dependencies are created betwen the major Hammer actions. For hierarchical designs, Hammer will use the hierarchy to build a dependency graph and construct the Make target dependencies appropriately.

`hammer.d` should be included in a higher-level Makefile. While `hammer.d` defines all of the variables that it needs, there are often reasons to set these elsewhere. Because `hammer.d` uses `?=` assignment, the settings created in the top-level Makefile will persist. An example of this setup is found in Chipyard.

To enable interactive usage, `hammer.d` also contains a set of companion "redo" targets (e.g. `redo-par` and `redo-syn`). These targets intentionally have no dependency information; they are for advanced users to make changes to the input config and/or edit the design manually, then continue the flow. Additional arguments can be passed to the "redo" targets with the `HAMMER_EXTRA_ARGS` environment variable, for the following example uses:

- "Patching" the configuration for an action using `HAMMER_EXTRA_ARGS="-p patch.yml"`.
- Flow control using `--to_step`/`--until_step` and `--from_step`/`after_step` to stop a run at a particular step or resume one from a previous iteration.

5.7 Hierarchical Hammer Flow

Hammer supports a bottom-up hierarchical flow. This is beneficial for very large designs to reduce the computing power by partitioning it into submodules, especially if there are many repeated instances of those modules.

5.7.1 Hierarchical Hammer Config

The hierarchal flow is controlled in the `vlsi.inputs.hierarchal` namespace. To specify hierarchical mode, you must specify the following keys. In this example, we have our top module set as `ChipTop`, with a submodule `ModuleA` and another submdule `ModuleAA` below that (these are names of Verilog modules).

```
vlsi.inputs.hierarchical:
  mode: hierarchical
  top_module: ChipTop
  config_source: manual
  manual_modules:
  - ChipTop:
    - ModuleA
  - ModuleA:
    - ModuleAA
  constraints:
  - ChipTop:
    - vlsi.core...
    - vlsi.inputs...
  - ModuleA:
    - vlsi.core...
    - vlsi.inputs...
  - ModuleAA:
    - vlsi.core...
    - vlsi.inputs...
```

Note how the configuration specific to each module in `vlsi.inputs.hierarchical.constraints` are list items, whereas in a flat flow, they would be at the root level.

Placement constraints for each module, however, are not specified here. Instead, they should be specified in `vlsi.inputs.hierarchical.manual_placement_constraints`. The parameters such as `x`, `y`, `width`, `height`, etc. are omitted from each constraint for clarity. In the bottom-up hierarchal flow, instances of submodules are of `type: hardmacro` because they are hardened from below.

```
vlsi.inputs.hierarchical:
  manual_placement_constraints_meta: append
  manual_placement_constraints:
  - ChipTop:
    - path: "ChipTop"
      type: toplevel
    - path: "ChipTop/path/to/instance/of/ModuleA"
      type: hardmacro
  - ModuleA:
    - path: "ModuleA"
      type: toplevel
    - path: "ModuleA/path/to/instance/of/ModuleAA"
      type: hardmacro
  - ModuleAA:
```

(continues on next page)

(continued from previous page)

```
  - path: "moduleAA"
    type: toplevel
```

5.7.2 Flow Management and Actions

Based on the structure in `vlsi.inputs.hierarchical.manual_modules`, Hammer constructs a hierarchical flow graph of dependencies. In this particular example, synthesis and place-and-route of `ModuleAA` will happen first. Synthesis of `ModuleA` will then depend on the place-and-route output of `ModuleAA`, and so forth.

These are enumerated in the auto-generated Makefile, `hammer.d`, which is placed in the directory pointed to by the `--obj_dir` command line flag when the `buildfile` action is run. This action must be run BEFORE executing your flow. If you adjust the hierarchy, you must re-run this action.

To perform a flow action (syn, par, etc.) for a module using the auto-generated Makefile, the name of the action is appended with the module name. For example, the generated Make target for synthesis of `ModuleA` is `syn-ModuleA`. A graphical example of the auto-generated dependency graph with `RocketTile` under a `ChipTop` is shown below:

The auto-generated Makefile also has `redo-` targets corresponding to each generated action, e.g. `redo-syn-ModuleA`. These targets break the dependency graph and allow you to re-run any action with new input configuration without forcing the flow to re-run from the beginning of the graph.

5.7.3 Cadence Implementation

Currently, the hierarchical flow is implemented with the Cadence plugin using its Interface Logic Model (ILM) methodology. At the end of each submodule's place-and-route, an ILM is written as the hardened macro, which contains an abstracted view of its design and timing models only up to the first sequential element.

ILMs are similar to LEFs and LIBs for traditional hard macros, except that the interface logic is included in all views. This means that at higher levels of hierarchy, the ILM instances can be flattened at certain steps, such as those that perform timing analysis on the entire design, resulting in a more accurate picture of timing than a normal LIB would afford.

5.7.4 Tips for Constraining Hierarchical Modules

In a bottom-up hierarchical flow, is is important to remember that submodules do not know the environment in which they will placed. This means:

- At minimum, the pins must be placed on the correct edges of the submodule on metal layers that are accessible in the parent level. Depending on the technology, this may interfere with things like power straps near the edge, so custom obstructions may be necessary. If fixed IOs are placed in submodules (e.g. bumps), then in the parent level, those pins must be promoted up using the `preplaced: true` option in the pin assignment.
- Clocks should be constrained more tightly for a submodule compared to its parent to account for extra clock insertion delay, jitter, and skew at increasingly higher levels of hierarchy. Otherwise, you may run into surprise timing violations in submodule instances even if those passed timing in isolation.
- You may need to specify pin delays `vlsi.inputs.delays` for many pins to optimize the partitioning of sequential signals that cross the submodule boundary. By default, without pin delay constraints, the input and output delay are constrained to be coincident with its related clock arrival at the module boundary.
- Custom SDC constraints that originate from a higher level (e.g. false paths from async inputs) need to be specified in submodules as well.

- Custom CPFs will need to be written if differently-named power nets need to globally connected between submodules. Similarly, hierarchical flow with custom CPFs can also be used to fake a multi-power domain topology until Hammer properly supports this feature.

5.7.5 Special Notes & Limitations

1. Hammer IR keys propagate up through the hierarchical tree. For example, if `vlsi.inputs.clocks` was specified in the constraints for `ModuleAA` but not for `ModuleA`, `ModuleA` will inherit `ModuleAA`'s constraints. Take special care of where your constraints come from, especially for a parent module with more than one submodule. To avoid confusion, it is recommended to specify the same set of keys for every module.
2. Hammer IR keys specified at the root level (i.e. outside of `vlsi.inputs.hierarchical.constraints`) do not override the corresponding submodule constraints. However, if you add a Hammer IR file using `-p` on the command line (after the file containing `vlsi.inputs.hierarchical.constraints`), those keys are global and override submodule constraints unless a meta action is specified. To avoid confusion, it is recommended to specify all constraints with `vlsi.inputs.hierarchical.constraints`.
3. Due to the structure of `vlsi.inputs.hierarchical.constraints` as a list structure, currently, there are the following limitations:
    - You must include all of the constraints in a single file. The config parser is unable to combine constraints from differnt files because most meta actions do not work on list items (advanced users will need to use `deepsubst`). This will make it harder for collaboration, and unfortunately, changes to module constraints at a higher level of hierarchy after submodules are hardened will trigger the Make dependencies, so you will need to modify the generated Makefile or use redo-targets.
    - Other issues have been observed, such as the bump API failing (see this issue at the top module level. This is caused by similar mechanisms as above. The workaround is to ensure that bumps are specified at the root level for only the top module and the bumps step is removed from submodule par actions.
4. Most Hammer APIs are not yet intelligent enough to constrain across hierarchical boundaries. For example:
    - The power straps API is unable to pitch match power straps based on legalized placement of submodule instances or vice versa.
    - The pin placement API does not match the placement of pins that may face each other in two adjacent submodule instances. You will need to either manually place the pins yourself or ensure a sufficient routing channel between the instances at the parent level.
5. Hammer does not support running separate decks for submodule DRC and LVS. Technology plugins may need to be written with Makefiles and/or technology-specific options that will implement different checks for submodules vs. the top level.

CHAPTER

SIX

HAMMER EXAMPLES

The following are prebuilt example designs, toolchain and/or flows you can use

6.1 Introduction with Sky130 and OpenROAD

The following directions will get a simple `pass` design from RTL to GDS using the OpenROAD tools and the Skywater 130nm PDK. These directions are meant to provide the minimal set of steps to do so, please reference the next section, Hammer End-to-End Integration Tests, for more detailed descriptions of all files and commands.

6.1.1 Instructions

First, follow the Hammer Developer Setup to clone Hammer and install/activate the poetry virtual environment.

Next, run the setup script to install the OpenROAD tools using Conda, and Skywater 130nm PDK using the Open-PDKs tool. This step will take a long time due to the amount and size of the required installs. You should supply a `PREFIX` path to a directory that will serve as the root of all PDK files and supporting tools (total size of all files is ~42GB), otherwise the script will default to installing to your home directory (`~/`).

```
cd hammer/e2e
./scripts/setup-sky130-openroad.sh [PREFIX]
```

You should now have a file `configs-env/my-env.yml` containing all required tool and technology paths for this tutorial. To point to your custom environment setup, set the Make variable `env=my`. Additionally, we set the `design`, `pdk`, and `tools` Make variables to the appropriate RTL design, PDK, and tools flow, respecively. Now simply run the VLSI flow:

```
make design=pass pdk=sky130 tools=or env=my build
make design=pass pdk=sky130 tools=or env=my syn
make design=pass pdk=sky130 tools=or env=my par
make design=pass pdk=sky130 tools=or env=my drc
make design=pass pdk=sky130 tools=or env=my lvs
```

After Place-and-Route completes, the final database can be opened in an interactive OpenROAD session. Hammer generates a convenient script to launch these sessions:

```
cd ./build-sky130-or/pass/par-rundir
./generated-scripts/open_chip
```

After DRC and LVS complete, the final results may be viewed with the following scripts:

```
# View DRC results:
cd ./build-sky130-or/pass/drc-rundir
./generated-scripts/view_drc

# View LVS results:
cd ./build-sky130-or/pass/lvs-rundir
./generated-scripts/open_chip
```

Congrats, you've just run your `pass` design from RTL to GDS with Hammer!

6.1.2 Next Steps

At this point, you should go through the Hammer End-to-End Integration Tests documentation to learn how to configure Hammer further. We've also outlined some common next steps below.

Commercial flow

This flow may be run with commercial tools by setting the Make variable `tools=cm`, which selects the tool plugins via `configs-tools/cm.yml`. You will need to create a custom environment YAML file with your environment configuration, in `configs-env/<my_custom>-env.yml`. As a minimum, you must specify YAML keys below for the CAD tool licenses and versions/paths of the tools you are using, see examples in `configs-env/`.

```
    # Commercial tool licenses/paths
    mentor.mentor_home: "" # Base path to where Mentor tools are installed
    mentor.MGLS_LICENSE_FILE: "" # Mentor license server/file

    cadence.cadence_home: "" # Base path to where Cadence tools are installed
    cadence.CDS_LIC_FILE: "" # Cadence license server/file

    synopsys.synopsys_home: "" # Base path to where Synopsys tools are installed
    synopsys.SNPSLMD_LICENSE_FILE: "" # Synopsys license server/files
    synopsys.MGLS_LICENSE_FILE: ""

    # Commercial tool versions/paths
    sim.vcs.version: ""
    synthesis.genus.version: "" # NOTE: for genus/innovus/joules, must specify binary␣
↪path if version < 221
    par.innovus.version: ""
    power.joules.joules_bin: ""
```

Now re-run a similar flow as before, but pointing to your environment and the commercial tool plugins:

```
make design=pass pdk=sky130 tools=cm env=<my_custom> build
```

Running DRC/LVS with commercial tools requires access to an NDA-version of the Skywater PDK. We support running DRC/LVS with either Cadence Pegasus or Siemens Calibre. See the Sky130 documentation for how to point to the NDA PDKs and run DRC/LVS with their respective tools.

Chipyard flow

Follow these directions in the Chipyard docs to build your own Chisel SoC design with OpenROAD and Sky130.

6.2 Hammer End-to-End Integration Tests

This folder contains an end-to-end (RTL -> GDS) smoketest flow using Hammer, using the Cadence toolchain, and the ASAP7 or Skywater 130 PDKs.

6.2.1 Setup

The integration tests use Hammer as a source dependency, so create the e2e poetry environment.

```
poetry install
poetry shell
```

6.2.2 Overview

Flow Selection

The following variables in the Makefile select the target flow to run:

- `design` - RTL name
  - {`pass`, `gcd`}
- `pdk` - PDK name
  - {`sky130`, `asap7`}
- `tools` - CAD tool flow
  - {`cm` (commercial), `or` (OpenROAD)}
- `env` - compute environment
  - {`bwrc` (BWRC), `a` (Millenium), `inst` (instructional machines)}

The outputs of the flow by default reside in `OBJ_DIR=build-<pdk>-<tools>/<design>/`

Configs

The Hammer configuration files consist of environment (`ENV_YML`) and project (`PROJ_YMLS`) configurations. The environment configs take precedence over ALL project configs. The order of precedence for the project configs reads from right to left (i.e. each file overrides all files to its left). All configuration files are summarized below.

```
#         lowest precedence ----------------------------------------------> highest␣
↪precedence
CONFS ?= $(PDK_CONF) $(TOOLS_CONF) $(DESIGN_CONF) $(DESIGN_PDK_CONF) $(SIM_CONF) $(POWER_
↪CONF)
```

- `ENV_YML`- Environment configs that specify CAD tool license servers and paths are in `configs-env`. This will take precedence over any other config file
- `PDK_CONF` - PDK configs shared across all runs with this PDK are in `configs-pdk`

- TOOLS_CONF - Tool configs to select which CAD tool flow to use are in configs-tools
- Design-specific configs are located in configs-design/<design>, and are summarized below:
  - DESIGN_CONF - the common design config (design input files, anything else design-specific)
  - DESIGN_PDK_CONF - PDK-specific configs for this particular design (clock, placement, pin constraints)
  - SIM_CONF - Simulation configs for post-RTL, Synthesis, or PnR simulation
  - POWER_CONF - Power simulation configs for post-RTL, Synthesis, or PnR simulation (NOTE: The Makefile expects the power config filename for each simulation level + PDK to be in the format power-{rtl,syn,par}-<pdk>.yml, while the joules.yml and voltus.yml files serve as templates for the Cadence Joules/Voltus power tools)

6.2.3 Run the Flow

First, use Hammer to construct a Makefile fragment with targets for all parts of the RTL -> GDS flow. Specify the appropriate env/tools/pdk/design variables to select which configs will be used.

```
make build
# same as: `make env=bwrc tools=cm pdk=sky130 design=pass build`
```

Hammer will generate a Makefile fragment in OBJ_DIR/hammer.d.

Then run the rest of the flow, making sure to set the env/tools/pdk/design variables as needed:

```
make sim-rtl
make power-rtl

make syn
make sim-syn
make power-syn

make par
make sim-par
make power-par

make drc
make lvs
```

These actions are summarized in more detail:

- RTL simulation
  - make sim-rtl
  - Generated waveform in OBJ_DIR/sim-rtl-rundir/output.fsdb
- Post-RTL Power simulation
  - make sim-rtl-to-power
  - make power-rtl
  - Generated power reports in OBJ_DIR/power-rtl-rundir/reports
- Synthesis
  - make syn

  - Gate-level netlist in `OBJ_DIR/syn-rundir/<design>.mapped.v`
- Post-Synthesis simulation
  - `make syn-to-sim`
  - `make sim-syn`
  - Generated waveform and register forcing ucli script in `OBJ_DIR/sim-syn-rundir`
- Post-Synthesis Power simulation
  - `make syn-to-power`
  - `make sim-syn-to-power`
  - `make power-syn`
  - Generated power reports in `OBJ_DIR/power-syn-rundir/reports`
- PnR
  - `make syn-to-par`
  - `make par`
  - LVS netlist (`<design>.lvs.v`) and GDS (`<design>.gds`) in `OBJ_DIR/par-rundir`
- Post-PnR simulation
  - `make par-to-sim`
  - `make sim-par`
- Post-PnR Power simulation
  - `make par-to-power`
  - `make sim-par-to-power`
  - `make power-par`
  - Generated power reports in `OBJ_DIR/power-par-rundir`

Flow Customization

If at any point you would like to use custom config files (that will override any previous configs), assign the `extra` Make variable to a space-separated list of these files. For example, to run the `pass` design with `sky130` through the commercial flow, but run LVS with Cadence Pegasus instead of the default Siemens Calibre, simply run the following:

```
make extra="configs-tool/pegasus.yml" build
```

To use the Hammer step flow control, prepend `redo-` to any VLSI flow action, then assign the `args` Make variable to the appropriate Hammer command line args. For example, to only run the `report_power` step of the `power-rtl` action (i.e. bypass synthesis), run the following:

```
make args="--only_step report_power" redo-power-rtl
```

6.2.4 Custom Setups

If you're not using a Berkeley EECS compute node, you can create your own environment setup.

- Create an environment config similar to those in `configs-env` for your node to specify the CAD tool license servers and paths/versions of CAD tools and the PDK, and set the `ENV_YML` variable to this file.
- The rest of the flow should be identical

ASAP7 Install

Clone the asap7 repo somewhere and reference the path in your `ENV_YML` config.

Sky130 Install

Refer to the Hammer Sky130 plugin README to install the Sky130 PDK, then reference the path in your `ENV_YML` config (only the `technology.sky130.sky130A` key is required).

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search