# Hammer

***Release 0.1***

**Jan 20, 2023**

# Contents:


**1 Introduction to Hammer** **3**

1.1 Hammer Basics . . . 3
1.1.1 Hammer Overview . . . 3
1.1.2 Hammer Setup . . . 5
1.2 Technology Setup and Use . . . 6
1.2.1 Hammer Tech JSON . . . 6
1.2.2 Hammer Tech defaults.yml . . . 10
1.3 Hammer CAD Tool Plugins . . . 11
1.3.1 Hammer CAD Tools . . . 11
1.3.2 Setting up a Hammer CAD Tool Plugin . . . 11
1.4 Hammer Flow Steps . . . 12
1.4.1 Hammer Actions . . . 12
1.4.2 Synthesis . . . 13
1.4.3 Place-and-Route . . . 14
1.4.4 DRC . . . 16
1.4.5 LVS . . . 17
1.4.6 Simulation . . . 18
1.4.7 Power . . . 21
1.4.8 Formal Verification . . . 24
1.4.9 Static Timing Analysis . . . 25
1.5 Hammer Use . . . 27
1.5.1 Hammer IR and Meta Variables . . . 27
1.5.2 Hammer APIs . . . 32
1.5.3 Flow Control . . . 34
1.5.4 Extending Hammer with Hooks . . . 35
1.5.5 Hammer Buildfile . . . 36
1.5.6 Hierarchical Hammer Flow . . . 37
1.6 Hammer Examples . . . 39
1.6.1 OpenROAD and Sky130 . . . 39

**2 Indices and tables** **41**

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.

CHAPTER 1

# Introduction to Hammer

Hammer (Higly Agile Masks Made Efforlessly 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.

## 1.1 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.

### 1.1.1 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.

### 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 form of YML and JSON files. The designer sets a variety of settings in the form of keys in different namespaces that are designated in Hammer to control its functionality. These keys are contained in `hammer/src/hammer-vlsi/defaults.yml`. This file shows all of the keys that are a part of main Hammer and 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: "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.

### 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/src/hammer-vlsi/technology/`, such as `asap7`, and how to setup a technology plugin is documented in more detail in the *Technology Setup and Use* 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 technology plugin repositories.

---

### 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` is publicly available; however, users must request access to `hammer-synopsys-plugins` and `hammer-mentor-plugins`:

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

---

**Note:** If you are not a UCB BAR or BWRC affiliate and have access to tools from a specific vendor, please email hammer-plugins-access@lists.berkeley.edu with a request for which plugin(s) you would like access to. MAKE SURE

TO INCLUDE YOUR GITHUB ID IN YOUR EMAIL AND YOUR ASSOCIATION TO SHOW YOU HAVE LICENSED ACCESS TO THOSE TOOLS. There will be no support guarantee for the plugin repositories, but users are encouraged to file issues and contribute patches where needed.

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.

#### Calling Hammer

To use Hammer on the command line, the designer will invoke the `hammer-vlsi` utility included in the core Hammer repo. 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
```

In both of these commands, an environment configuration is passed to Hammer using a `-e` flag, which in this case is `env.yml`. `env.yml` contains pointers to the required tool licenses and environment variables. These environment settings will not be propagated to the output configuration files after each action.

Any number of other YML or JSON files can then be 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.

`--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 `src/hammer-vlsi/hammer_vlsi/cli_driver.py` file.

### 1.1.2 Hammer Setup

Hammer has a few requirements and there are several environment variables to setup.

#### System Requirements

- Python 3.6+ required
- The `ruamel.yaml` package is recommended for key history (`pip install ruamel.yaml`)
- python3 in the $PATH
- hammer-shell in the $PATH
- hammer_config, python-jsonschema-objects, hammer-tech, hammer-vlsi in $PYTHONPATH

- HAMMER_PYYAML_PATH set to pyyaml/lib3 or pyyaml in $PYTHONPATH
- HAMMER_HOME set to hammer repo root
- HAMMER_VLSI path set to $HAMMER_HOME/src.hammer-vlsi

Sourcing `hammer/sourceme.sh` will setup the environment described above.

To check your environment you may run the following:

```
git submodule update --init --recursive
export HAMMER_HOME=$PWD
source sourceme.sh
cd src/test
./unittests.sh
echo $?
```

If the last line above returns 0, then the environment is set up and ready to go.

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.

## 1.2 Technology Setup and Use

These guides will walk you through how to set up a technology 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.

### 1.2.1 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 a full schema that the tech JSON supports, please see `src/hammer-tech/schema.json`.

#### 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`. `install-example` shows an example of the installs and tarballs from the ASAP7 plugin.

```
"name": "ASAP7 Library",
"grid_unit": "0.001",
"time_unit": "1 ps",
"installs": [
  {
    "path": "$PDK",
    "base var": "technology.asap7.pdk_install_dir"
  },
  {
    "path": "$STDCELLS",
    "base var": "technology.asap7.stdcell_install_dir"
  },
```

(continues on next page)

(continued from previous page)

```
  {
    "path": "tech-asap7-cache",
    "base var": ""
  }
],
"tarballs": [
  {
    "path": "ASAP7_PDK_CalibreDeck.tar",
    "homepage": "http://asap.asu.edu/asap/",
    "base var": "technology.asap7.tarball_dir"
  }
],
```

Notice how in the installs, there are two directories holding the PDK files and standard cell files. The `tech-asap7-cache` with an empty `base var` denotes files that exist in the tech cache, which are placed there by a post-installation PDK hacking script (see ASAP7's `post_install_script` method). Finally, the encrypted Calibre decks are provided in a tarball.

### 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. `deck-example` shows 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.

### 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. `library-example` shows 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"
      }
```

(continues on next page)

(continued from previous page)

```
    ]
  },
  {
    "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"
    },
    "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.

TODO: ADD INFO ABOUT LIBRARY FILTERS

### Stackup

The `stackups` sets up the important metal layer information for Hammer to use. `stackups-example` shows 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.

### 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.

### Special Cells

The `special cells` field specifies a set of cells in the technology that have special functions. `special-cells-example` 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"]},
```

There are 8 `cell_type` s supported: `tiehicell`, `tielocell`, `tiehilocell`, `endcap`, `iofiller`, `stdfiller`, `decap`, and `tapcell`. 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"]},
```

### 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"
],
```

### 1.2.2 Hammer Tech defaults.yml

The `defaults.yml` for a technology specifies some technology-specific Hammer IR 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/src/hammer-vlsi/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 should be set in this file. 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.

## 1.3 Hammer CAD 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.

### 1.3.1 Hammer CAD Tools

Hammer currently has open-source CAD tool plugins in the `hammer/src/hammer-vlsi/` folder and three repos for CAD tools from commercial vendors: `hammer-cadence-plugins`, `hammer-synopsys-plugins`, and `hammer-mentor-plugins`. `hammer-cadence-plugins` is a public repo but the others are private since they contain 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:

- ACTION
  - TOOL_NAME
    - `__init__.py` contains the methods needed to implement the tool
    - `defaults.yml` contains the default Hammer IR 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, having `vlsi.core.par_tool_path: par_tool_foo` in your configuration would expect a TOOL_NAME of `par_tool_foo`.

### 1.3.2 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/src/hammer-vlsi/`.

#### 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`.

#### 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,
```

(continues on next page)

(continued from previous page)

```
        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.

#### 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.

#### 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.

#### 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.

#### 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.

## 1.4 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.

### 1.4.1 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/src/hammer-vlsi/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.

### 1.4.2 Synthesis

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

#### Synthesis Setup Keys

- Namespace: `vlsi.core`
  - `synthesis_tool_path`
    - Set to the directory containing the tool plugin directory for the DRC tool, typically `/path/to/tool_plugin/synthesis`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
  - `synthesis_tool`
    - Actual name of the synthesis tool that is setup in the directory `synthesis_tool_path`, e.g. `genus`

#### 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

#### Synthesis Inputs

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

#### 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.

#### 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`

### 1.4.3 Place-and-Route

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

#### P&R Setup Keys

- Namespace: `vlsi.core`
  - `par_tool_path`
    - Set to the directory containing the tool plugin directory for the place-and-route tool, typically `/path/to/tool_plugin/par`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
  - `par_tool`
    - Actual name of the P&R tool that is setup in the directory, `par_tool_path`, e.g. `innovus`

#### 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.

### P&R Inputs

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

### 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.

#### 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`

### 1.4.4 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`.

#### DRC Setup Keys

- Namespace: `vlsi.core`
  - `drc_tool_path`
    - Set to the directory containing the tool plugin directory for the DRC tool, typically `/path/to/tool_plugin/drc`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
  - `drc_tool`
    - Actual name of the DRC tool that is setup in the directory, `drc_tool_path`, e.g. `calibre`

#### 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.

#### DRC Inputs

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

#### 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`

#### 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`

### 1.4.5 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`.

#### LVS Setup Keys

- Namespace: `vlsi.core`
    - `lvs_tool_path`
        * Set to the directory containing the tool plugin directory for the LVS tool, typically `/path/to/tool_plugin/lvs`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
    - `lvs_tool`
        * Actual name of the lvs tool that is setup in the directory, `lvs_tool_path`, e.g. `calibre`

#### 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.

#### LVS Inputs

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

#### 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`

#### 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`

### 1.4.6 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`.

#### Simulation Setup Keys

- Namespace: `vlsi.core`
  - **`sim_tool_path`**
    - Set to the directory containing the tool plugin directory for the sim tool, typically `/path/to/tool_plugin/sim`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
  - **`sim_tool`**
    - Actual name of the simulation tool that is setup in the directory `sim_tool_path`, e.g. `vcs`

### 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.

### 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.

### 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/`.

#### 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`

### 1.4.7 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`.

#### Power Setup Keys

- Namespace: `vlsi.core`
  - **`power_tool_path`**
    - Set to the directory containing the tool plugin directory for the power tool, typically `/path/to/tool_plugin/power`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
  - **`power_tool`**
    - Actual name of the power tool that is setup in the directory `power_tool_path`, e.g. `voltus`

#### 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.

### 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.

### 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.

### 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`

### 1.4.8 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`.

#### Formal Verification Setup Keys

- Namespace: `vlsi.core`
  - **`formal_tool_path`**
    - Set to the directory containing the tool plugin directory for the formal tool, typically `/path/to/tool_plugin/formal`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.
  - **`formal_tool`**
    - Actual name of the formal verification tool that is setup in the directory `formal_tool_path`, e.g. `conformal`

#### 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.

#### 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.

#### 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.

#### 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`

### 1.4.9 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`.

#### STA Setup Keys

- Namespace: `vlsi.core`
  - **`timing_tool_path`**
    - Set to the directory containing the tool plugin directory for the STA tool, typically `/path/to/tool_plugin/timing`. This will be the parent directory of the directory containing `__init__.py` and `defaults.yml`.

  - **timing_tool**
    - Actual name of the STA tool that is setup in the directory `timing_tool_path`, e.g. `tempus`

### 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.

### 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.

### 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.

### 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`

## 1.5 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.

### 1.5.1 Hammer IR and Meta Variables

#### 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).

#### 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.

There is a built-in order of precedence, which from lowest to highest: 1) Hammer default, 2) tool plugin, 3) tech plugin, 4) User's Hammer IR. In 4), subsequent JSON/YAML files specified with `-p` in the command line have higher precedence, and keys appearing later if duplicated in a 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*).

#### 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`.

### 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 the next snippet contains `foo: 54321`, then `get_setting("foo")` would return `54321`.

### 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-config` 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.

### 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`.

### Common meta actions

- `append`: append the elements provided to the base list. (See the above `vlsi.tech.foobar65.bad_cells` example.)
- `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`

- `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`

- `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 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`

### 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.

### Key History

When the `ruamel.yaml` package is installed, HAMMER can emit what files have modified any configuration keys in YAML format. 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: "nop"
        technology_path: ["src/hammer-vlsi/technology"]

        synthesis_tool: "nop"
        synthesis_tool_path: ["src/hammer-vlsi/synthesis"]
```

`test/syn-rundir/syn-output-history.yml` after executing the command `hammer-vlsi -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.technology_path: # Modified by: test-config.yml
  - src/hammer-vlsi/technology
vlsi.core.synthesis_tool: nop # Modified by: test-config.yml
vlsi.core.synthesis_tool_path: # Modified by: test-config.yml
  - src/hammer-vlsi/synthesis
```

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: "nop"
        technology_path: ["src/hammer-vlsi/technology"]

        synthesis_tool: "nop"
        synthesis_tool_path: ["src/hammer-vlsi/synthesis"]
```

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

vlsi:
    core:
        technology: "${foo.subst}"
        technology_path: ["/dev/null"]
        technology_path_meta: subst

        par_tool: "nop"
        par_tool_path: ["src/hammer-vlsi/par"]

foo.subst: "nop2"
```

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

```
foo.subst: 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: nop2 # Modified by: test-config.yml, test-config2.yml
vlsi.core.technology_path: # Modified by: test-config.yml, test-config2.yml
  - /dev/null
vlsi.core.synthesis_tool: nop # Modified by: test-config.yml
vlsi.core.synthesis_tool_path: # Modified by: test-config.yml
  - src/hammer-vlsi/synthesis
vlsi.core.par_tool: nop # Modified by: test-config2.yml
vlsi.core.par_tool_path: # Modified by: test-config2.yml
  - src/hammer-vlsi/par
```

#### 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 propertly:

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`):

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

#### 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/src/hammer_config_test/test.py
- https://github.com/ucb-bar/hammer/blob/master/src/hammer_config/config_src.py

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

### 1.5.2 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.

### 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.

### 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.

### 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`.

### 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.

### 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.

#### Power Straps

Power strap constraints are specified using multiple Hammer IR keys in the `par` namespace. 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.

#### 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.

#### 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.

### 1.5.3 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.

#### 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).

### 1.5.4 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.

#### 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 `HammerToolHooksTest` class of test.py.

#### 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.

#### Technology, Tool, and User-Provided Hooks

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 technology 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 technology 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 technology 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 techonolgy's persistent hook.

### 1.5.5 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
```

`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 will also create a set of "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. This allows the user to create "patches" to the configuration, which can be easily passed to Hammer by setting, for example, `HAMMER_EXTRA_ARGS="-p patch.yml"`. Other potential uses for `HAMMER_EXTRA_ARGS` include 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.

### 1.5.6 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.

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

### 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. Syntheis of `ModuleA` will then depend on the place-and-route output of `ModuleAA`, and so forth. These are enumerated in the auto-generated Makefile.

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`.

### 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.

### 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.

#### 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.

## 1.6 Hammer Examples

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

### 1.6.1 OpenROAD and Sky130

The OpenROAD-flow is an open-source, technology-independent VLSI toolchain. As of this writing, it has the ability to run synthesis, place-and-route, and drc through various open-source tools.

Hammer has the ability to target the OpenROAD toolchain and the Skywater 130nm PDK. So you can now simply push a button in Chipyard to go from your Chisel design to a somewhat reasonable gds at an example 130-nm node.

### Instructions

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

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search