OpenROAD

OpenROAD Team

Aug 19, 2021

CONTENTS

The OpenROAD ("Foundations and Realization of Open, Accessible Design") project was launched in June 2018 within the DARPA IDEA program. OpenROAD aims to bring down the barriers of cost, expertise and unpredictability that currently block designers' access to hardware implementation in advanced technologies. The project team (Qualcomm, Arm and multiple universities and partners, led by UC San Diego) is developing a fully autonomous, open-source tool chain for digital SoC layout generation, focusing on the RTL-to-GDSII phase of system-on-chip design. Thus, OpenROAD holistically attacks the multiple facets of today's design cost crisis: engineering resources, design tool licenses, project schedule, and risk.

The IDEA program targets no-human-in-loop (NHIL) design, with 24-hour turnaround time and zero loss of power-performance-area (PPA) design quality.

The NHIL target requires tools to adapt and auto-tune successfully to flow completion, without (or, with minimal) human intervention. Machine intelligence augments human expertise through efficient modeling and prediction of flow and optimization outcomes throughout the synthesis, placement and routing process. This is complemented by development of metrics and machine learning infrastructure.

The 24-hour runtime target implies that problems must be strategically decomposed throughout the design process, with clustered and partitioned subproblems being solved and recomposed through intelligent distribution and management of computational resources. This ensures that the NHIL design optimization is performed within its available `[threads * hours]` "box" of resources. Decomposition that enables parallel and distributed search over cloud resources incurs a quality-of-results loss, but this is subsequently recovered through improved flow predictability and enhanced optimization.

For a technical description of the OpenROAD flow, please refer to our DAC-2019 paper: Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. The paper is also available from ACM Digital Library. Other publications and presentations are linked here.

CHAPTER

ONE

CODE OF CONDUCT

Please read our code of conduct *here*.

CHAPTER

TWO

DOCUMENTATION

The OpenROAD Project has two releases:

2.1 Application

The application is a standalone binary capable of performing RTL-to-GDSII SoC design, from logic synthesis and floorplanning through detailed routing with metal fill insertion, signoff parasitic extraction and timing analysis.

See documentation for the *application here*.

2.2 Flow

The flow is a set of integrated scripts that allow for RTL-to-GDSII flow using open-source tools.

See documentation for the *flow here*.

CHAPTER

THREE

HOW TO CONTRIBUTE

If you are willing to **contribute**, see the *Getting Involved* section.

If you are a **developer** with EDA background, learn more about how you can use OpenROAD as the infrastructure for your tools in the *Developer Guide* section.

CHAPTER

FOUR

HOW TO GET IN TOUCH

We maintain the following channels for communication:

- Project homepage and news: https://theopenroadproject.org
- Twitter: https://twitter.com/OpenROAD_EDA
- Issues and bugs:
  - OpenROAD: https://github.com/The-OpenROAD-Project/OpenROAD/issues
  - OpenROAD Flow: https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/issues
- Discussions:
  - OpenROAD: https://github.com/The-OpenROAD-Project/OpenROAD/discussions
  - OpenROAD Flow: https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/discussions
- Inquiries: openroad@eng.ucsd.edu

See also our *FAQs*.

CHAPTER

FIVE

SITE MAP

5.1 Contributor Covenant Code of Conduct

5.1.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

5.1.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.1.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

5.1.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

5.1.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at complaints@openroad.tools. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

5.1.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

5.1.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

5.2 Getting Involved

Thank you for taking the time to read this document and to contribute. The OpenROAD project will not reach all of its objectives without help!

Possible ways to contribute:

- Open-source PDK information
- Open-source Designs
- Useful scripts
- Tool improvements
- New tools
- Improvements to documentation, including this document
- Star our project and repos so we can see the number of people who are interested

5.2.1 Licensing Contributions

As much as possible, all contributions should be licensed using the BSD3 license. You can propose another license if you must, but contributions made with BSD3 fit best with the spirit of OpenROAD's permissive open-source philosophy. We do have exceptions in the project, but over time we hope that all contributions will be BSD3, or some other permissive license such as MIT or Apache2.0.

5.2.2 Contributing Open Source PDK information and Designs

If you have new design or PDK information to contribute, please add this to the repo OpenROAD-flow-scripts. In the flow directory you will see a directory for designs with Makefiles to run them, and one for PDK platforms used by the designs. If you add a new PDK platform, be sure to add at least one design that uses it.

5.2.3 Contributing Scripts and Code

We follow the Google C++ style guide. If you find code in our project that does *not* follow this guide, then within each file that you edit, follow the style in that file.

Please pay careful attention to the [tool checklist](DeveloperGuide.md#Tool Checklist) for all code. If you want to add or improve functionality in OpenROAD, please start with the top-level app repo. You can see in the `src` directory that submodules exist pointing to tested versions of the other relevant repos in the project. Please look at the tool workflow in the developer guide *document* to work with the app and its submodule repos in an efficient way.

Please run clang-format on all the C++ source files that you change, before committing. In the root directory of the OpenROAD repository there is the file `.clang-format` that defines all coding formatting rules.

Please pay attention to the test directory and be sure to add tests for any code changes that you make, using open-source PDK and design information. We provide the `nangate45` PDK in the OpenROAD-flow-scripts repo to help with this. Pull requests with code changes are unlikely to be accepted without accompanying test cases. There are many examples tests. Each repo has a test directory as well with tests you should run and add to if you modify something in one of the submodules.

For changes that claim to improve QoR or PPA, please run many tests and ensure that the improvement is not design-specific. There are designs in the OpenROAD-flow-scripts repo which can be used unless the improvement is technology-specific.

Do not add runtime or build dependencies without serious thought. For a project like OpenROAD with many application subcomponents, the software architecture can quickly get out of control. Changes with lots of new dependencies which are not necessary are less likely to be integrated.

If you want to add Tcl code to define a new tool command, look at pdngen as an example of how to do so. Take a look at the cmake file which automatically sources the Tcl code and the Tcl file itself.

To accept contributions, we require each commit to be made with a DCO (Developer Certificate of Origin) attached. When you commit you add the `-s` flag to your commit. For example:

```
git commit -s -m "test dco with -s"
```

This will append a statement to your commit comment that attests to the DCO. GitHub has built in the `-s` option to its command line since use of this is so pervasive. The promise is very basic, certifying that you know that you have the right to commit the code. Please read the full statement here.

5.2.4 Questions

Please refer to our *FAQs*.

5.3 Developer Guide

5.3.1 Tool Philosophy

OpenROAD is a tool to build a chip from synthesizable RTL (Verilog) to completed physical layout (manufacturable, tapeout-clean GDSII).

The unifying principle behind the design of OpenROAD is for all of the tools to reside in one tool, with one process, and one database. All tools in the flow should use Tcl commands exclusively to control them instead of external "configuration files". File-based communication between tools and forking processes is strongly discouraged. This architecture streamlines the construction of a flexible tool flow and minimizes the overhead of invoking each tool in the flow.

5.3.2 Tool File Organization

Every tool follows the following file structure, grouping sources, tests and headers together.

```
src/CMakelists.txt - add_subdirectory's src/CMakelists.txt
src/tool/src/ - sources and private headers
src/tool/src/CMakelists.txt
src/tool/include/tool/ - exported headers
src/tool/test/
src/tool/test/regression
```

OpenROAD repository:

```
CMakeLists.txt - top-level CMake file
src/Main.cc
src/OpenRoad.cc - OpenROAD class functions
src/OpenRoad.i - top-level swig, %includes tool swig files
src/OpenRoad.tcl - basic read/write lef/def/db commands
include/ord/OpenRoad.hh - OpenROAD top-level class, has instances of tools
```

Some tools such as OpenSTA are submodules, which are simply subdirectories in `src/` that are pointers to the git submodule. They are intentionally not segregated into a separate /module.

The use of submodules for new code integrated into OpenROAD is strongly discouraged. Submodules make changes to the underlying infrastructure (e.g., OpenSTA) difficult to propagate across the dependent submodule repositories. Submodules: just say no.

Where external/third-party code that a tool depends on should be placed depends on the nature of the dependency.

- Libraries - code packaged as a linkable library. Examples are `tcl`, `boost`, `zlib`, `eigen`, `lemon`, `spdlog`.

These should be installed in the build environment and linked by OpenROAD. Document these dependencies in the top-level `README.md` file. The `Dockerfile` should be updated to illustrate where to find the library and how to install it. Adding libraries to the build environment requires coordination with system administrators, so that continuous integration hosts ensure that environments include the dependency. Advance notification should also be given to the development team so that their private build environments can be updated.

Each tool CMake file builds a library that is linked by the OpenROAD application. The tools should not define a `main()` function. If the tool is Tcl only and has no C++ code, it does not need to have a CMake file. Tool CMake files should **not** include the following:

- `cmake_minimum_required`
- `GCC_COVERAGE_COMPILE_FLAGS`
- `GCC_COVERAGE_LINK_FLAGS`
- `CMAKE_CXX_FLAGS`
- `CMAKE_EXE_LINKER_FLAGS`

None of the tools have commands to read or write LEF, DEF, Verilog or database files. For consistency, these functions are all provided by the OpenROAD framework.

Tools should package all state in a single class. An instance of each tool class resides in the top-level OpenROAD object. This allows multiple tools to exist at the same time. If any tool keeps state in global variables (even static), then only one tool can exist at a time. Many of the tools being integrated were not built with this goal in mind and will only work on one design at a time.

Each tool should use a unique namespace for all of its code. The same namespace should be used for Tcl functions, including those defined by a swig interface file. Internal Tcl commands stay inside the namespace, and user visible Tcl commands should be defined in the global namespace. User commands should be simple Tcl commands such as 'global_placement' that do not create tool instances that must be based to the commands. Defining Tcl commands for a tool class is fine for internal commands, but not for user visible commands. Commands have an implicit argument of the current OpenROAD class object. Functions to get individual tools from the OpenROAD object can be defined.

5.3.3 Initialization (C++ tools only)

The OpenROAD class has pointers to each tool, with functions to get each tool. Each tool has (at a minimum) a function to make an instance of the tool class, an initialization function that is called after all of the tools have been made, and a function to delete the tool. This small header does **not** include the class definition for the tool so that the OpenROAD framework does not have to know anything about the tool internals or include a gigantic header file.

`MakeTool.hh` defines the following:

```
Tool *makeTool();
void initTool(OpenRoad *openroad);
void deleteTool(Tool *tool);
```

The `OpenRoad::init()` function calls all of the `makeTool` functions and then all of the `initTool()` functions. The `init` functions are called from the bottom of the tool dependencies. Each `init` function grabs the state it needs out of the `OpenRoad` instance.

5.3.4 Commands

Tools should provide Tcl commands to control them. Tcl object based tool interfaces are not user-friendly. Define Tcl procedures that take keyword arguments that reference the `OpenRoad` object to get tool state. OpenSTA has Tcl utilities to parse keyword arguments (`sta::parse_keyword_args`). See `OpenSTA/tcl/*.tcl` for examples. Use swig to define internal functions to C++ functionality.

Tcl files can be included by encoding them in CMake into a string that is evaluated at run time (See `Resizer::init()`).

5.3.5 Errors

Tools should report errors to the user using the `ord::error` function defined in `include/openroad/Error.hh`. `ord::error` throws `ord::Exception`. The variables `ord::exit_on_error` and `ord::file_continue_on_error` control how the error is handled. If `ord::exit_on_error` is `true` then OpenROAD reports the error and exits. If the error is encountered while reading a file with the `source` or `read_sdc` commands and `ord::file_continue_on_error` is `false` then no other commands are read from the file. The default value is `false` for both variables.

5.3.6 Test

Each "tool" has a /test directory containing a script named `regression` to run "unit" tests. With no arguments it should run default unit tests.

No database files should be in tests. Read LEF/DEF/Verilog to make a database.

The regression script should not depend on the current working directory. It should be able to be run from any directory. Use filenames relative to the script name rather the current working directory.

Regression scripts should print a concise summary of test failures. The regression script should return an exit code of 0 if there are no errors and 1 if there are errors. The script should **not** print thousands of lines of internal tool information.

Regression scripts should pass the `-no_init` option to `openroad` so that a user's `init` file is not sourced before the tests runs.

Regression scripts should add output files or directories to `.gitignore` so that running does not leave the source repository "dirty".

The Nangate45 open-source library data used by many tests is in `test/Nangate45`. Use the following command to add a link in the tool command

```
cd tool/test
ln -s ../../../test/Nangate45
```

After the link is installed, the test script can read the Liberty file with the command shown below.

```
read_liberty Nangate45/Nangate45_typ.lib
```

5.3.7 Building

Instructions for building are available *here*.

5.3.8 Example of Adding a Tool to OpenROAD

The patch file "add_tool.patch" illustrates how to add a tool to OpenROAD. Use

```
patch -p < docs/misc/AddTool.patch
cd src/tool/test
ln -s ../../../test/regression.tcl regression.tcl
```

to add the sample tool. This adds a directory `OpenRoad/src/tool` that illustrates a tool named "Tool" that uses the file structure described above and defines a command to run the tool with keyword and flag arguments as illustrated below:

```
% toolize foo
Helping 23/6
Gotta positional_argument1 foo
Gotta param1 0.000000
Gotta flag1 false

% toolize -flag1 -key1 2.0 bar
Helping 23/6
Gotta positional_argument2 bar
Gotta param1 2.000000
Gotta flag1 true

% help toolize
toolize [-key1 key1] [-flag1] positional_argument1
```

5.3.9 Documentation

Tool commands should be documented in the top-level OpenROAD `README.md` file. Detailed documentation should be the `tool/README.md` file.

5.3.10 Tool Flow

- Verilog to DB (dbSTA)
- Floorplan initialization (OpenROAD)
- I/O placement (ioPlacer)
- PDN generation (pdngen)
- I/O placement (ioPlacer)
- Tapcell and welltie insertion (tapcell)
- Macro placement (TritonMacroPlace)
- Global placement (RePlAce)
- Gate resizing and buffering (Resizer)
- Detailed placement (OpenDP)
- Clock tree synthesis (TritonCTS)
- Repair hold violations (Resizer)
- Global route (FastRoute)
- Antenna check (OpenROAD)
- Detailed route (TritonRoute)
- Metal fill insertion (OpenROAD)
- Final timing/power report (OpenSTA)

5.3.11 Tool Checklist

Tools should make every attempt to minimize external dependencies. Linking libraries other than those currently in use complicates the builds and sacrifices the portability of OpenROAD. OpenROAD should be portable to many different compiler/operating system versions and dependencies make this vastly more complicated.

1. OpenROAD submodules reference tool `openroad` branch head. No git `develop`, `openroad_app`, or `openroad_build` branches.
2. Submodules used by more than one tool belong in `src/`, not duplicated in each tool repo.
3. `CMakeLists.txt` does not use add_compile_options include_directories link_directories link_libraries. Use target_ versions instead. See https://gist.github.com/mbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1
4. `CMakeLists.txt` does not use glob. Use explicit lists of source files and headers instead.
5. `CMakeLists.txt` does not define `CFLAGS` `CMAKE_CXX_FLAGS` `CMAKE_CXX_FLAGS_DEBUG` `CMAKE_CXX_FLAGS_RELEASE`. Let the top level and defaults control these.
6. No `main.cpp` or main procedure.
7. No compiler warnings for GCC or Clang with optimization enabled.
8. Does not call `flute::readLUT` (called once by `openroad`).
9. Tcl command(s) documented in top level `README.md` in flow order.
10. Command line tool documentation in tool README.
11. Conforms to Tcl command naming standards (no camel case).
12. Does not read configuration files. Use command arguments or support commands.
13. `.clang-format` at tool root directory to aid foreign programmers.
14. No `jenkins/`, `Jenkinsfile`, `Dockerfile` in tool directory.
15. `regression` script named `test/regression` with no arguments that runs tests. Not `tests/regression-tcl.sh`, not `test/run_tests.py` etc.
16. `regression` script should run independent of current directory. For example, `../test/regression` should work.
17. `regression` should only print test results or summary, not belch 1000s of lines of output.
18. Test scripts use OpenROAD tcl commands (not `itcl`, not internal accessors).
19. `regression` script should only write files in a directory that is in the tool's `.gitignore` so the hierarchy does not have modified files in it as a result or running the regressions.
20. Regressions report no memory errors with `valgrind` (stretch goal).
21. Regressions report no memory leaks with `valgrind` (difficult).

James Cherry, Dec 2019

5.3.12 Coding Practices

List of coding practices.

NOTE

This is a compilation of many idioms in OpenROAD code that I consider undesirable. Obviously other programmers have different opinions or they would not be so pervasive. James Cherry 04/2020

C++

Practice #1

Don't comment out code. Remove it. `git` provides a complete history of the code if you want to look backwards. Huge chunks of commented-out code that are stunningly common in student code make it nearly impossible to read.

`FlexTa.cpp` has 220 lines of code and 600 lines of commented-out code.

Practice #2

Don't use prefixes on function names or variables. That's what namespaces are for.

```
namespace fr {
  class frConstraint
  class frLef58CutClassConstraint
  class frShortConstraint
  class frNonSufficientMetalConstraint
  class frOffGridConstraint
  class frMinEnclosedAreaConstraint
  class frMinStepConstraint
  class frMinimumcutConstraint
  class frAreaConstraint
  class frMinWidthConstraint
  class frLef58SpacingEndOfLineWithinEndToEndConstraint
  class frLef58SpacingEndOfLineWithinParallelEdgeConstraint
  class frLef58SpacingEndOfLineWithinMaxMinLengthConstraint
  class frLef58SpacingEndOfLineWithinConstraint
  class frLef58SpacingEndOfLineConstraint
}
```

Practice #3

Namespaces should be all lower case and short. This is an example of a poor choice: `namespace TritonCTS`

Practice #4

Don't use `extern` on function definitions. It is pointless in a world with prototypes.

```
namespace fr {
  extern frCoord getGCELLGRIDX();
  extern frCoord
  getGCELLGRIDY();
  extern frCoord getGCELLOFFSETX();
  extern frCoord
  getGCELLOFFSETY();
}
```

Practice #5

Don't use prefixes on file names. That's what directories are for.

```
frDRC.h frDRC_init.cpp frDRC_main.cpp frDRC_setup.cpp frDRC_util.cpp
```

Practice #6

Don't name variables theThingy, curThingy or myThingy. It is just distracting extraneous verbiage. Just use thingy.

```
float currXSize;
float currYSize;
float currArea;
float currWS;
float currWL;
float currWLnoWts;
```

Practice #7

Do not use global variables. All state should be inside of classes. Global variables make multi-threading next to impossible and preclude having multiple copies of a tool running in the same process. The only global variable in `openroad` should be the singleton that Tcl commands reference.

```
extern std::string DEF_FILE;
extern std::string GUIDE_FILE;
extern std::string OUTGUIDE_FILE;
extern std::string LEF_FILE;
extern std::string OUTTA_FILE;
extern std::string OUT_FILE;
extern std::string DBPROCESSNODE;
extern std::string OUT_MAZE_FILE;
extern std::string DRC_RPT_FILE;
extern int MAX_THREADS ;
extern int VERBOSE ;
extern int BOTTOM_ROUTING_LAYER;
extern bool ALLOW_PIN_AS_FEEDTHROUGH;
```

(continues on next page)

(continued from previous page)

```
extern bool USENONPREFTRACKS;
extern bool USEMINSPACING_OBS;
extern bool RESERVE_VIA_ACCESS;
extern bool ENABLE_BOUNDARY_MAR_FIX;
```

Practice #8

Do not use strings (names) to refer to database or sta objects except in user interface code. DEF, SDC, and Verilog all use different names for netlist instances and nets, so the names will not always match.

Practice #9

Do not use continue. Wrap the body in an if instead.

```
// instead of
for(dbInst* inst : block->getInsts() ) {
  // Skip for standard cells
  if (inst->getBBox()->getDY() <= cellHeight) { continue; }
  // code
}
// use
for(dbInst* inst : block->getInsts() ){
  // Skip for standard cells
  if (inst->getBBox()->getDY() > cellHeight) {
    // code
  }
}
```

Practice #10

Don't put magic numbers in the code. Use a variable with a name that captures the intent. Document the units if they exist.

Examples of unnamed magic numbers:

```
referenceHpwl_= 446000000;
coeffV = 1.36;
coeffV = 1.2;
double nearest_dist = 99999999999;
if (dist < rowHeight * 2) {}
for(int i = 9; i > -1; i--) {}
if(design_util > 0.6 || num_fixed_nodes > 0) div = 1;
avail_region_area += (theRect->xUR - theRect->xLL - (int)theRect->xUR % 200 + (int)t␣
↪heRect->xLL % 200 - 200) * (theRect->yUR - theRect->yLL - (int)theRect->yUR % 2000 +␣
↪(int)theRect->yLL % 2000 - 2000);
```

Practice #11

Don't copy code fragments. Write functions.

```
// 10x
int x_pos = (int)floor(theCell->x_coord / wsite + 0.5);
// 15x
int y_pos = (int)floor(y_coord / rowHeight + 0.5);

// This
nets[newnetID]->netIDorg = netID;
nets[newnetID]->numPins = numPins;
nets[newnetID]->deg = pinInd;
nets[newnetID]->pinX = (short *)malloc(pinInd* sizeof(short));
nets[newnetID]->pinY = (short *)malloc(pinInd* sizeof(short));
nets[newnetID]->pinL = (short *)malloc(pinInd* sizeof(short));
nets[newnetID]->alpha = alpha;

// Should factor out the array lookup.
Net *net = nets[newnetID];
net->netIDorg = netID;
net->numPins = numPins;
net->deg = pinInd;
net->pinX = (short*)malloc(pinInd* sizeof(short));
net->pinY = (short *)malloc(pinInd* sizeof(short));
net->pinL = (short *)malloc(pinInd* sizeof(short));
net->alpha = alpha;

// Same here:
if (grid[j][k].group != UINT_MAX) {
  if (grid[j][k].isValid) {
    if (groups[grid[j][k].group].name == theGroup->name)
      area += wsite * rowHeight;
  }
}
```

Practice #12

Don't use logical operators to test for null pointers.

```
if (!net) {
  // code
}

// should be
if (net != nullptr) {
  // code
}
```

Practice #13

Don't use malloc. Use new. We are writting C++, not C.

Practice #14

Don't use C style arrays. There is no bounds checks for them so they invite subtle memory errors to unwitting programmers who fail to use valgrind. Use std::vector or std::array.

Practice #15

Break long functions into smaller ones, preferably that fit on one screen.

- 162 lines void DBWrapper::initNetlist()
- 246 lines static vector<pair<Partition, Partition>> GetPart()
- 263 lines void MacroCircuit::FillVertexEdge()

Practice #16

Don't reinvent functions like round, floor, abs, min, max. Use the std versions.

```
int size_x = (int)floor(theCell->width / wsite + 0.5);
```

Practice #17

Don't use C stdlib.h abs, fabs or fabsf. They fail miserably if the wrong arg type is passed to them. Use std::abs.

Practice #18

Fold code common to multiple loops into the same loop. Each of these functions loops over every instance like this:

```
legal &= row_check(log);
legal &= site_check(log);
for(int i = 0; i < cells.size(); i++) {
  cell* theCell = &cells[i];
  legal &= power_line_check(log);
  legal &= edge_check(log);
  legal &= placed_check(log);
  legal &= overlap_check(log);
}
// with this loop
for(int i = 0; i < cells.size(); i++) {
  cell* theCell = &cells[i];
}
```

Instead make one pass over the instances doing each check.

Practice #19

Don't use == true, or == false. Boolean expressions already have a value of true or false.

```
if(found.first == true) {
  // code
}
// is simply
if(found.first) {
  // code
}
// and
if(found.first == false) {
  // code
}
// is simply
if(!found.first) {
 // code
}
```

Practice #20

Don't nest if statements. Use && on the clauses instead.

```
if(grid[j][k].group != UINT_MAX)
  if(grid[j][k].isValid == true)
    if(groups[grid[j][k].group].name == theGroup->name)
```

is simply

```
if(grid[j][k].group != UINT_MAX
   && grid[j][k].isValid
   && groups[grid[j][k].group].name == theGroup->name)
```

Practice #21

Don't call return at the end of a function that does not return a value.

Practice #22

Don't use <>'s to include anything but system headers. Your project's headers should NEVER be in <>'s. - https://gcc.gnu.org/onlinedocs/cpp/Include-Syntax.html - https://stackoverflow.com/questions/21593/what-is-the-difference-between-include-filename-and-include-filename

These are all wrong:

```
#include <odb/db.h>
#include <sta/liberty/Liberty.hh>
#include <odb/db.h>
#include <odb/dbTypes.h>
```

(continues on next page)

(continued from previous page)

```
#include <odb/defin.h>
#include <odb/defout.h>
#include <odb/lefin.h>
```

Practice #23

Don't make "include the kitchen sink" headers and include them in every source file. This is convenient (lazy) but slows the builds down for everyone. Make each source file include just the headers it actually needs.

```
// Types.hpp
#include <sta/liberty/Liberty.hh>
#include <odb/db.h>
#include <odb/dbTypes.h>
// It should be obvious that every source file is not reading def.
#include <odb/defin.h>
// or writing it.
#include <odb/defout.h>
#include <odb/lefin.h>
#include "db_sta/dbNetwork.hh"
#include "db_sta/dbSta.hh"
```

Note this example also incorrectly uses <>'s around OpenROAD headers.

Header files should only include files to support the header. Include files necessary for code in the code file, not the header.

In the example below NONE of the system files listed are necessary for the header file.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

unsigned num_nets = 1000;
unsigned num_terminals = 64;
unsigned verbose = 0;
float alpha1 = 1;
float alpha2 = 0.45;
float alpha3 = 0;
float alpha4 = 0;
float margin = 1.1;
unsigned seed = 0;
unsigned root_idx = 0;
unsigned dist = 2;
float beta = 1.4;
bool runOneNet = false;
unsigned net_num = 0;
```

Practice #24

Use class declarations if you are only referring to objects by pointer instead of including their complete class definition. This can vastly reduce the code the compiler has to process.

```
class Network;
// instead of
#include "Network.hh"
```

Practice #25

Use pragma once instead of #define to protect headers from being read more than once. The #define symbol has to be unique, which is difficult to guarantee.

```
// Instead of:
#ifndef __MACRO_PLACER_HASH_UTIL__
#define __MACRO_PLACER_HASH_UTIL__
#endif
// use
#pragma once
```

Practice #26

Don't put "using namespace" inside a function. It makes no sense whatsoever but I have seen some very confused programmers do this far too many times.

Practice #27

Don't nest namespaces. We don't have enough code to justify that complication.

Practice #28

Don't use `using namespace` It is just asking for conflicts and doesn't explicity declare what in the namespace is being used. Use `using namespace::symbol;` instead. And especially NEVER EVER EVER `using namespace std`. It is HUGE.

The following is especially confused because it is trying to "use" the symbols in code that are already in the MacroPlace namespace.

```
using namespace MacroPlace;

namespace MacroPlace { }
```

Practice #29

Use `nullptr` instead of `NULL`. This is the C++ approved version of the ancient C `#define`.

Practice #30

Use range iteration. C++ iterators are ugly and verbose.

```
// Instead of
odb::dbSet::iterator nIter;
for (nIter = nets.begin(); nIter != nets.end(); ++nIter) {
  odb::dbNet* currNet = *nIter;
  // code
}
// use
for (odb::dbNet* currNet : nets) {
  // code
}
```

Practice #34

Don't use end of line comments unless they are very short. Don't assume that the person reading your code has a 60" monitor.

```
for (int x = firstTile._x; x <= lastTile._x; x++) { // Setting capacities of edges␣
↪completely inside the adjust region according the percentage of reduction
  // code
}
```

Practice #35

Don't std::pow for powers of 2 or for decimal constants.

```
// This
double newCapPerSqr = (_options->getCapPerSqr() * std::pow(10.0, -12));
// Should be
double newCapPerSqr = _options->getCapPerSqr() * 1E-12;

// This
unsigned numberOfTopologies = std::pow(2, numberOfNodes);
// Should be
unsigned numberOfTopologies = 1 << numberOfNodes;
```

Git

Practice #31

Don't put /'s in `.gitignore` directory names. `test/`

Practice #32

Don't put file names in `.gitignore` ignored directories. `test/results` `test/results/diffs`

Practice #33

Don't list compile artifacts in `.gitignore`. They all end up in the build directory so each file type does not have to appear in `.gitignore`.

All of the following is nonsense that has propagated faster than COVID in student code:

Compiled Object files

```
*.slo *.lo *.o *.obj
```

Precompiled Headers

```
*.gch *.pch
```

Compiled Dynamic libraries

```
*.so *.dylib *.dll
```

Fortran module files

```
*.mod *.smod
```

Compiled Static libraries

```
*.lai *.la *.a *.lib
```

CMake

Practice #35

Don't change compile flags in `cmake` files. These are set at the top level and should not be overridden.

```
set(CMAKE_CXX_FLAGS "-O3")
set(CMAKE_CXX_FLAGS_DEBUG "-g -ggdb")
set(CMAKE_CXX_FLAGS_RELEASE "-O3")
```

Practice #36

Don't put /'s in CMake directory names. CMake knows they are directories.

```
target_include_directories( ABKCommon PUBLIC ${ABKCOMMON_HOME} src/ )
```

Practice #37

Don't use glob. Explictly list the files in a group.

```
# Instead of
file(GLOB_RECURSE SRC_FILES ${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp)
# should be
list(REMOVE_ITEM SRC_FILES ${CMAKE_CURRENT_SOURCE_DIR}/src/Main.cpp)
list(REMOVE_ITEM SRC_FILES ${CMAKE_CURRENT_SOURCE_DIR}/src/Parameters.h)
list(REMOVE_ITEM SRC_FILES ${CMAKE_CURRENT_SOURCE_DIR}/src/Parameters.cpp)
```

5.3.13 Database Math 101

DEF defines the units it uses with the units command.

```
    UNITS DISTANCE MICRONS 1000 ;
```

Typically the units are 1000 or 2000 database units (DBU) per micron. DBUs are integers, so the distance resolution is typically 1/1000u or 1nm.

OpenDB uses an `int` to represent a DBU, which on most hardware is 4 bytes. This means a database coordinate can be +/-2147483647, which is about 2 billion units, corresponding to 2 million microns or 2 meters.

Since chip coordinates cannot be negative, it would make sense to use an `unsigned int` to represent a distance. This conveys the fact that it can never be negative and doubles the maximum possible distance that can be represented. The problem, however, is that doing subtraction with unsigned numbers is dangerous because the differences can be negative. An unsigned negative number looks like a very very big number. So this is a very bad idea and leads to bugs.

Note that calculating an area with `int` values is problematic. An `int * int` does not fit in an `int`. My suggestion is to use `int64_t` in this situation. Although `long` "works", its size is implementation-dependent.

Unfortunately, I have seen multiple instances of programs using a `double` for distance calculations. A double is 8 bytes, with 52 bits used for the mantissa. So, the largest possible integer value that can be represented without loss is 5e+15, 12 bits less than using an `int64_t`. Doing an area calculation on a large chip that is more than `sqrt(5e+15) = 7e+7 DBU` on a side will overflow the mantissa and truncate the result.

Not only is a `double` less capable than an `int64_t`, but using it tells any reader of the code that the value can be a real number, such as 104.23. So it is extremely misleading.

Circling back to LEF, we see that unlike DEF the distances are real numbers like 1.3 even though LEF also has a distance unit statement. I suspect this is a historical artifact of a mistake made in the early definition of the LEF file format. The reason it is a mistake is because decimal fractions cannot be represented exactly in binary floating-point. For example, 1.1 = 1.00011001100110011..., a continued fracion.

OpenDB uses `int` to represent LEF distances, just as with DEF. This solves the problem by multiplying distances by a decimal constant (distance units) to convert the distance to an integer. In the future I would like to see OpenDB use a `dbu` typedef instead of `int` everywhere.

Unfortunately, I see RePlAce, OpenDP, TritonMacroPlace and OpenNPDN all using `double` or `float` to represent distances and converting back and forth between DBUs and microns everywhere. This means they also need to `round` or `floor` the results of every calculation because the floating-point representation of the LEF distances is a fraction that cannot be exactly represented in binary. Even worse is the practice of reinventing round in the following idiom.

`(int) x_coord + 0.5`

Even worse than using a `double` is using `float` because the mantissa is only 23 bits, so the maximum exactly representable integer is 8e+6. This makes it even less capable than an `int`.

When a value has to be snapped to a grid such as the pitch of a layer, the calculation can be done with a simple divide using `int`s, which `floor`s the result. For example, to snap a coordinate to the pitch of a layer the following can be used:

```
int x, y;
inst->getOrigin(x, y);
int pitch = layer->getPitch();
int x_snap = (x / pitch) * pitch;
```

The use of rounding in existing code that uses floating-point representations is to compensate for the inability to represent floating-point fractions exactly. Results like 5.99999999992 need to be "fixed". This problem does not exist if fixed-point arithmetic is used.

The **only** place that the database distance units should appear in any program should be in the user interface, because humans like microns more than DBUs. Internally, code should use `int` for all database units and `int64_t` for all area calculations.

James Cherry, 2019

5.3.14 Using the Logging Infrastructure

OpenROAD uses spdlog as part of logging infrastructure in order to ensure a clear, consistent messaging and complete messaging interface. A wrapper formats the prefix in the recommended messaging style and limit. A message format is as follows:

```
<tool id>-<message id>  <Message body>.
```

For example,

```
[INFO ODB-0127] Reading DEF file: ./results/asap7/aes/base/4_cts.def
```

All output from OpenROAD tools should be directed through the logging API to ensure that redirection, file logging and execution control flow are handled consistently. This also includes messages from any third-party tool. Use the 'ord' message ID for third-party tools.

The logging infrastructure also supports generating a JSON file containing design metrics (e.g., area or slack). This output is directed to a user-specified file. The OpenROAD application has a `-metrics` command line argument to specify the file.

Handling Messages

OpenROAD supports multiple levels of severity for message outputs: critical, error, warning, information and debug. These are supported by automatic calls to the logger which will then prefix the appropriate severity type to the message.

Messaging Guidelines

In addition to the proper use of message types, follow the guidelines below to compose messages for clarity, consistency and other guidelines:

Grammar

Start with a capital letter and end with a period, besides well-known exceptions. Use capital letters for file formats and tool proper names, e.g., LEF, DEF, SPICE, FLUTE.

After the first word's capitalization, do not use capital letters (aside from obvious exceptions, such as RSMT, hCut, etc.).

Do not use exclamations. Severity must be communicated by message severity and clear implied or explicit action.

Avoid long, verbose messages. Use commas to list and separate clauses in messages.

Spellcheck all messages using American English spellings.

Use ellipsis `...` only to indicate a pause, as when some tool is running or being initialized.

Abbreviations and Shortcuts

Use single-word versions when well-accepted / well-understood by users and developers. Examples: `stdcell`, `cutline`, `wirelength`, `flipchip`, `padring`, `bondpad`, `wirebond`, `libcell`, `viarule`.

Do not abbreviate or truncate English words; expand for the sake of clarity.

```
Incorrect:   Num, #;  Tot.
```

```
Correct:     Number;  Total
```

Use acceptable, well-understood abbreviations for brevity. Examples: `db`, `tech`, `lib`, `inst`, `term`, `params`, `etc`.

Avoid contractions of action words:

```
Incorrect:   Can't, Can not;  Don't
```

```
Correct:     Cannot;  Do not
```

Actionability

Messages should communicate a clear, implied or explicit action that is necessary for flow continuation or improved quality of results.

```
Example:
A value for core_area must be specified in the footprint specification, or in the␣
↪environment variable CORE_AREA.
```

Clarity

Messages must be clear and complete, so as to communicate necessary and sufficient information and actions. Elaborate specific variables, options, and/or parameters to avoid any ambiguity.

```
Example:
Unrecognized argument $arg, should be one of -pitch, -bump_pin_name, -spacing_to_edge, -
↪cell_name, -bumps_per_tile, -rdl_layer, -rdl_width, -rdl_spacing.
```

Specify objects clearly in the local context:

```
Example:
cutWithin is smaller than cutSpacing for ADJACENTCUTS on layer {}. Please check your␣
↪rule definition.

Incomplete:
Warning: {} does not have viaDef aligned with layer.
```

Make any assumptions or use of default values explicit:

```
Example:
No net slacks found.
Timing-driven mode disabled.

Incomplete, missing default:
Utilization exceeds 100%.
```

Use simple language, and avoid repetitions:

```
Example:
Missing orientation for cell $cell_ref.

Incorrect:
No orientation available for orientation of $cell_ref.
```

Message Types

OpenROAD supports the following levels of severity through the logger: report, debug, information, warning, error and critical.

Report

Report messages are output by the tool in the form of a report to the user. Examples include timing paths or power analysis results.

Example report message:

```
Path startpoint: $startpoint
```

Debug

Debug messages are only of use to tool developers and not to end users. These messages are not shown unless explicitly enabled.

Information

Information messages may be used to report metrics, quality of results, or program status to the user. Any message which indicates runtime problems, such as potential faulty input or other internal program issues, should be issued at a higher status level.

Example information messages:

```
Number of input ports: 47

Running optimization iteration 2

Current cell site utilization: 57.1567%
```

Warning

Warnings should be used to indicate atypical runtime conditions that may affect quality, but not correctness, of the output. Any conditions that affect correctness should be issued at a higher status level.

Example warning messages:

```
Core area utilization is greater than 90%. The generated cell placement may not be␣
↪routable.

14 outputs are not constrained for max capacitance.

Pin 'A[0]' on instance 'mem01' does not contain antenna information and will not be␣
↪checked for antenna violations.
```

Error

Error messages should be used to indicate correctness problems. Problems with command arguments are a good example of where error messages are appropriate. Errors exit the current command by throwing an exception that is converted to an error in Tcl. Errors that occur while reading a command file stop execution of the script commands.

Example error messages:

```
Invalid selection: net 'test0' does not exist in the design.

Cell placement cannot be run before floorplanning.

Argument 'max_routing_layer' expects an integer value from 1 to 10.
```

Critical

Critical messages should be used to indicate correctness problems that the program is not able to work around or ignore, and that require immediate exiting of the program (abort).

Example critical messages:

```
Database 'chip' has been corrupted and is not recoverable.

Unable to allocate heap memory for array 'vertexIndices'. The required memory size may␣
↪exceed host machine limits.

Assertion failed: 'nodeVisited == false' on line 122 of example.cpp. Please file a␣
↪Github issue and attach a testcase.
```

Coding

Each status message requires:

- The three letter tool ID
- The message ID
- The message string
- Optionally, additional arguments to fill in placeholders in the message string

Reporting is simply printing and does not require a tool or message ID. The tool ID comes from a fixed enumeration of all the tools in the system. This enumeration is in `Logger.h`. New abbreviations should be added after discussion with the OpenROAD system architects. The abbreviation matches the C++ namespace for the tool.

Message IDs are integers. They are expected to be unique for each tool. This has the benefit that a message can be mapped to the source code unambiguously even if the text is not unique. Maintaining this invariant is the tool owner's responsibility. To ensure that the IDs are unique, each tool should maintain a file named 'messages.txt' in the top-level tool directory, listing the message IDs along with the format string. When code that uses a message ID is removed, the ID should be retired by removing it from 'messages.txt'. See the utility `etc/find_messages.py` to scan a tool directory and write a `messages.txt` file.

Spdlog comes with the `fmt` library which supports message formatting in a python or C++20 like style.

The message string should not include the tool ID or message ID which will automatically be prepended. A trailing newline will automatically be added, and hence messages should not end with one. Messages should be written as complete sentences and end in a period. Multi-line messages may contain embedded new lines.

Some examples:

```
logger->report("Path startpoint: {}", startpoint);

logger->error(ODB, 25, "Unable to open LEF file {}.", file_name);

logger->info(DRT, 42, "Routed {} nets in {:3.2f}s.", net_count, elapsed_time);
```

Tcl functions for reporting messages are defined in the OpenROAD swig file `OpenRoad.i`. The message is simply a Tcl string (no C++20 formatting).

```
utl::report "Path startpoint: $startpoint"

utl::error ODB 25 "Unable to open LEF file $file_name."

utl::info DRT 42 "Routed $net_count nets in [format %3.2f $elapsed_time]."
```

`utl::report` should be used instead of 'puts' so that all output is logged.

Calls to the Tcl functions `utl::warn` and `utl::error` with a single message argument report with tool `ID UKN` and message `ID 0000`.

Tools use `#include utl/Logger.h` that defines the logger API. The Logger instance is owned by the OpenROAD instance. Each tool should retrieve the logger instance in the tool init function called after the tool make function by the OpenROAD application.

Every tool swig file must include src/Exception.i so that errors thrown by `utl::error` are caught at the Tcl command level. Use the following swig command before `%inline`.

```
%include "../../Exception.i"
```

The logger functions are shown below.

```
Logger::report(const std::string& message,
               const Args&... args)
Logger::info(ToolId tool,
             int id,
             const std::string& message,
             const Args&... args)
Logger::warn(ToolId tool,
             int id,
             const std::string& message,
             const Args&... args)
Logger::error(ToolId tool,
              int id,
              const std::string& message,
              const Args&... args)
Logger::critical(ToolId tool,
                 int id,
                 const std::string& message,
                 const Args&... args)
```

The corresponding Tcl functions are shown below.

```
utl::report message
utl::info tool id message
```

(continues on next page)

(continued from previous page)

```
utl::warn tool id message
utl::error tool id message
utl::critical tool id message
```

Although there is a `utl::critical` function, it is really difficult to imagine any circumstances that would justify aborting execution of the application in a tcl function.

Debug Messages

Debug messages have a different programming model. As they are most often *not* issued the concern is to avoid slowing down normal execution. For this reason such messages are issued by using the debugPrint macro. This macro will avoid evaluating its arguments if they are not going to be printed. The API is:

```
debugPrint(logger, tool, group, level, message, ...);
```

The `debug()` method of the Logger class should not be called directly. No message id is used as these messages are not intended for end users. The level is printed as the message id in the output.

The argument types are as for the info/warn/error/critical messages. The one additional argument is group which is a `const char*`. Its purpose is to allow the enabling of subsets of messages within one tool.

Debug messages are enabled with the tcl command: `set_debug_level <tool> <group> <level>`

Metrics

The metrics logging uses a more restricted API since JSON only supports specific types. There are a set of overloaded methods of the form:

```
metric(ToolId tool,
       const std::string_view metric,
       <type> value)
```

where `<type>` can be `int, double, string, or bool`. This will result in the generated JSON:

```
"<tool>-<metric>" : value
```

String values will be enclosed in double-quotes automatically.

Converting to Logger

The error functions in `include/openroad/Error.hh` should no longer be included or used. Use the corresponding logger functions.

All uses of the tcl functions ord::error and ord::warn should be updated call the `utl::error/warn` with a tool ID and message ID. For compatibility these are defaulted to `UKN` and `0000` until they are updated.

Regression tests should not have any `UKN-0000` messages in their ok files. A simple grep should indicate that you still have pending calls to pre-logger error/warn functions.

The `cmake` file for the tool must also be updated to include spdlog in the link libraries so it can find the header files if they are not in the normal system directories.

NOTE

At UCSD, dfm.ucsd.edu is an example of this problem; it has an ancient version of spdlog in '/usr/include/spdlog'. Use `module` to install spdlog 1.8.1 on dfm.ucsd.edu and check your build there.

```
target_link_libraries(<library_target>
  PUBLIC
    utl
)
```

±————————±————————+ | Tool | message/namespace | +==================+==================+
| antenna_checker | ant | ±————————±————————+ | dbSta | sta | ±————————±————————+
| FastRoute | grt | ±————————±————————+ | finale | fin | ±————————±————————+ |
flute3 | stt | ±————————±————————+ | gui | gui | ±————————±————————+ | ICeWall |
pad | ±————————±————————+ | init_fp | ifp | ±————————±————————+ | ioPlacer | ppl
| ±————————±————————+ | OpenDB | odb | ±————————±————————+ | opendp | dpl |
±————————±————————+ | OpenRCX | rcx | ±————————±————————+ | *OpenROAD* | >
ord | ±————————±————————+ | OpenSTA | sta | ±————————±————————+ | PartMgr |
par | ±————————±————————+ | pdngen | pdn | ±————————±————————+ | PDNSim |
psm | ±————————±————————+ | replace | gpl | ±————————±————————+ | resizer | rsz
| ±————————±————————+ | tapcell | tap | ±————————±————————+ | TritonCTS | cts |
±————————±————————+ | TritonMacroPlace | mpl | ±————————±————————+ | TritonRoute |
drt | ±————————±————————+ | utility | utl | ±————————±————————+

5.4 OpenROAD

The documentation is also available here.

5.4.1 Install dependencies

For a limited number of configurations the following script can be used to install dependencies. The script `etc/DependencyInstaller.sh` supports Centos7 and Ubuntu 20.04. You need root access to correctly install the dependencies with the script.

```
./etc/DependencyInstaller.sh -help

Usage: ./etc/DependencyInstaller.sh -run[time]      # installs dependencies to run a pre-
↪compiled binary
       ./etc/DependencyInstaller.sh -dev[elopment]  # installs dependencies to compile␣
↪the openroad binary
```

5.4.2 Build

The first step, independent of the build method, is to download the repository:

```
git clone --recursive https://github.com/The-OpenROAD-Project/OpenROAD.git
cd OpenROAD
```

OpenROAD git submodules (cloned by the `--recursive` flag) are located in `src/`.

The default build type is RELEASE to compile optimized code. The resulting executable is in `build/src/openroad`.

Optional CMake variables passed as `-D<var>=<value>` arguments to CMake are show below.

Argument	Value
CMAKE_BUILD_TYPE	DEBUG, RELEASE
CMAKE_CXX_FLAGS	Additional compiler flags
TCL_LIB	Path to tcl library
TCL_HEADER	Path to tcl.h
ZLIB_ROOT	Path to zlib
CMAKE_INSTALL_PREFIX	Path to install binary

Build by hand

```
mkdir build
cd build
cmake ..
make
```

The default install directory is `/usr/local`. To install in a different directory with CMake use:

```
cmake .. -DCMAKE_INSTALL_PREFIX=<prefix_path>
```

Alternatively, you can use the `DESTDIR` variable with make.

```
make DESTDIR=<prefix_path> install
```

Build using support script

```
./etc/Build.sh
# To build with debug option enabled and if the Tcl library is not on the default path
./etc/Build.sh --cmake="-DCMAKE_BUILD_TYPE=DEBUG -DTCL_LIB=/path/to/tcl/lib"
```

The default install directory is `/usr/local`. To install in a different directory use:

```
./etc/Build.sh --cmake="-DCMAKE_INSTALL_PREFIX=<prefix_path>"
```

5.4.3 Regression Tests

There are a set of regression tests in `test/`.

```
# run all tool unit tests
test/regression
# run all flow tests
test/regression flow
# run <tool> tests
test/regression <tool>
# run <tool> tool tests
src/<tool>/test/regression
```

The flow tests check results such as worst slack against reference values. Use `report_flow_metrics [test]...` to see the all of the metrics. Use `save_flow_metrics [test]...` to add margins to the metrics and save them to .metrics_limits.

```
% report_flow_metrics gcd_nangate45
                      insts    area util slack_min slack_max  tns_max clk_skew max_slew␣
↪max_cap max_fanout DPL ANT drv
gcd_nangate45           368     564  8.8     0.112    -0.015     -0.1    0.004        0␣
↪      0          0   0   0   0
```

5.4.4 Run

```
openroad
  -help              show help and exit
  -version           show version and exit
  -no_init           do not read ~/.openroad init file
  -no_splash         do not show the license splash at startup
  -threads count     max number of threads to use
  -exit              exit after reading cmd_file
  cmd_file           source cmd_file
```

OpenROAD sources the Tcl command file `~/.openroad` unless the command line option `-no_init` is specified.

OpenROAD then sources the command file cmd_file if it is specified on the command line. Unless the `-exit` command line flag is specified it enters and interactive Tcl command interpreter.

Below is a list of the available tools/modules included in the OpenROAD app.

OpenROAD (global commands)

- *OpenROAD*

Database

- OpenDB

Parasitics Extraction

- *OpenRCX*

Synthesis

- *Restructure*

Initialize Floorplan

- *Floorplan*

Pin placement

- *ioPlacer*

Chip level connections

- *ICeWall*

Macro Placement

- *TritonMacroPlacer*

Tapcell

- *Tapcell*

PDN analysis

- *PDN*
- *PDNSim*

Global Placement

- RePlAce

Timing Analysis

- *OpenSTA*

Gate Resizer

- *Resizer*

Detailed Placement

- *OpenDP*

Clock Tree Synthesis

- *TritonCTS 2.0*

Global Routing

- *FastRoute*
- *Antenna Checker*

Detailed Router

- *TritonRoute*

Metal Fill

- *Metal Fill*

5.4.5 License

5.4.6 OpenROAD

OpenROAD is run using Tcl scripts. The following commands are used to read and write design data.

```
read_lef [-tech] [-library] filename
read_def filename
write_def [-version 5.8|5.6|5.5|5.4|5.3] filename
read_verilog filename
write_verilog filename
read_db filename
write_db filename
write_abstract_lef filename
```

Use the Tcl `source` command to read commands from a file.

```
source [-echo] file
```

If an error is encountered in a command while reading the command file, then the error is printed and no more commands are read from the file. If `file_continue_on_error` is 1 then OpenROAD will continue reading commands after the error.

If `exit_on_error` is 1 then OpenROAD will exit when it encounters an error.

OpenROAD can be used to make a OpenDB database from LEF/DEF, or Verilog (flat or hierarchical). Once the database is made it can be saved as a file with the `write_db` command. OpenROAD can then read the database with the `read_db` command without reading LEF/DEF or Verilog.

The `read_lef` and `read_def` commands can be used to build an OpenDB database as shown below. The `read_lef -tech` flag reads the technology portion of a LEF file. The `read_lef -library` flag reads the MACROs in the LEF file. If neither of the `-tech` and `-library` flags are specified they default to `-tech -library` if no technology has been read and `-library` if a technology exists in the database.

```
read_lef liberty1.lef
read_def reg1.def
# Write the db for future runs.
write_db reg1.db
```

The `read_verilog` command is used to build an OpenDB database as shown below. Multiple Verilog files for a hierarchical design can be read. The `link_design` command is used to flatten the design and make a database.

```
read_lef liberty1.lef
read_verilog reg1.v
link_design top
# Write the db for future runs.
write_db reg1.db
```

Example scripts

Example scripts demonstrating how to run OpenROAD on sample designs can be found in /test. Flow tests taking sample designs from synthesizable RTL Verilog to detail-routed final layout in the open-source technologies Nangate45 and Sky130HD are shown below.

```
gcd_nangate45.tcl
aes_nangate45.tcl
tinyRocket_nangate45.tcl
gcd_sky130hd.tcl
aes_sky130hd.tcl
ibex_sky130hd.tcl
```

Each of these designs use the common script `flow.tcl`.

Abstract LEF Support

OpenROAD contains an abstract LEF writer that can take your current design and emit an abstract LEF representing the external pins of your design and metal obstructions.

```
read reg1.db
write_abstract_lef reg1_abstract.lef
```

Limitations of the Abstract LEF Writer

Currently the writer will place an obstruction over the entire block area on any metal layer if there is any object on that metal layer.

TCL functions

Get the die and core areas as a list in microns: `llx lly urx ury`

```
ord::get_die_area
ord::get_core_area
```

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License.

5.4.7 OpenRCX

OpenRCX is a Parasitic Extraction (PEX, or RCX) tool that works on OpenDB design APIs. It extracts routed designs based on the LEF/DEF layout model.

OpenRCX extracts both Resistance and Capacitance for wires, based on coupling distance to the nearest wire and the track density context over and/or under the wire of interest, as well as cell abstracts. The capacitance and resistance measurements are based on equations of coupling distance interpolated on exact measurements from a calibration file, called the Extraction Rules file. The Extraction Rules file (RC technology file) is generated once for every process node and corner, using a provided utility for DEF wire pattern generation and regression modeling.

OpenRCX stores resistance, coupling capacitance and ground (i.e., grounded) capacitance on OpenDB objects with direct pointers to the associated wire and via db objects. Optionally, OpenRCX can generate a `.spef` file.

Commands

Extract Parasitics

```
extract_parasitics
  [-ext_model_file filename]      pointer to the Extraction Rules file
  [-corner_cnt count]             process corner count
  [-max_res ohms]                 combine resistors in series up to
                                  <max_res> value in OHMS
  [-coupling_threshold fF]        coupling below the threshold is grounded
  [-lef_res]                      use per-unit RC defined in LEF
  [-cc_model track]               calculate coupling within
                                  <cc_model> distance
  [-context_depth depth]          calculate upper/lower coupling from
                                  <depth> level away
  [-no_merge_via_res]             separate via resistance
```

The `extract_parasitics` command performs parasitic extraction based on the routed design. If there is no routed design information, then no parasitics are returned. Use `ext_model_file` to specify the Extraction Rules file used for the extraction.

The `cc_model` option is used to specify the maximum number of tracks of lateral context that the tool considers on the same routing level. The default value is 10. The `context_depth` option is used to specify the number of levels of vertical context that OpenRCX needs to consider for the over/under context overlap for capacitance calculation. The default value is 5. The `max_res` option combines resistors in series up to the threshold value. The `no_merge_via_res` option separates the via resistance from the wire resistance.

The `corner_cnt` defines the number of corners used during the parasitic extraction.

Write SPEF

```
write_spef
  [-net_id net_id]                output the parasitics info for specific nets
  [filename]                      the output filename
```

The `write_spef` command writes the .spef output of the parasitics stored in the database. Use `net_id` option to write out .spef for specific nets.

Scale RC

```
adjust_rc
  [-res_factor res]               scale the resistance value by this factor
  [-cc_factor cc]                 scale the coupling capacitance value by this factor
  [-gndc_factor gndc]             scale the ground capacitance value by this factor
```

Use the `adjust_rc` command to scale the resistance, ground, and coupling capacitance. The `res_factor` specifies the scale factor for resistance. The `cc_factor` specifies the scale factor for coupling capacitance. The `gndc_factor` specifies the scale factor for ground capacitance.

Comparing SPEF files

```
diff_spef
  [-file filename]                specifies the input .spef filename
```

The `diff_spef` command compares the parasitics in the database with the parasitic information from <file>.spef. The output of this command is `diff_spef.out` and contains the RC numbers from the parasitics in the database and the <file>.spef, and the percentage RC difference of the two data.

Extraction Rules File Generation

```
bench_wires
  [-cnt count]                    specify the metal count
                                  per pattern
  [-len wire_len]                 specify the wire length
                                  in the pattern
  [-s_list space_multiplier_list] list of wire spacing multipliers
  [-all]                          generate all patterns
```

The `bench_wires` command produces a layout which contains various patterns that are used to characterize per-unit length R and C values. The generated patterns model the lateral, vertical, and diagonal coupling capacitances, as well as ground capacitance effects. This command generates a .def file that contains a number of wire patterns.

The `cnt` option specifies the number of wires in each pattern; the default value of cnt is 5. Use the `len` option to change the wire length in the pattern. The `all` option is used to specify all different pattern geometries (over, under, over_under, and diagonal). The option `all` is required.

The `s_list` option specifies the lists of wire spacing multipliers from the minimum spacing defined in the LEF. The list will be the input index on the OpenRCX RC table (Extraction Rules file).

This command is specifically intended for the Extraction Rules file generation only.

```
bench_verilog
    [filename]                    the output verilog filename
```

`bench_verilog` is used after the `bench_wires` command. This command generates a Verilog netlist of the generated pattern layout by the `bench_wires` command.

This command is optional when running the Extraction Rules generation flow. This step is required if the favorite extraction tool (i.e., reference extractor) requires a Verilog netlist to extract parasitics of the pattern layout.

```
bench_read_spef
    [filename]                    the input .spef filename
```

The `bench_read_spef` command reads a <filename>.spef file and stores the parasitics into the database.

```
write_rules
  [-file filename]                output file name
  [-db]                           read parasitics from the database
```

The `write_rules` command writes the Extraction Rules file (RC technology file) for OpenRCX. It processes the parasitics data from the layout patterns that are generated using the `bench_wires` command, and writes the Extraction Rules file with <file> as the output file.

The db option instructs OpenRCX to write the Extraction Rules file from the parasitics stored in the database. This option is required.

This command is specifically intended for the purpose of Extraction Rules file generation.

Example scripts

Example scripts demonstrating how to run OpenRCX in the OpenROAD environment on sample designs can be found in /test. An example flow test taking a sample design from synthesizable RTL Verilog to final-routed layout in an open-source SKY130 technology is shown below.

```
gcd.tcl
```

Example scripts demonstrating how to run the Extraction Rules file generation can be found in this directory.

```
generate_patterns.tcl     # generate patterns
generate_rules.tcl        # generate the Extraction Rules file
ext_patterns.tcl          # check the accuracy of OpenRCX
```

Regression tests

There is a set of regression tests in /test.

```
./test/regression
```

Extraction Rules File Generation

This flow generates an Extraction Rules file (RC tech file, or RC table) for OpenRCX. This file provides resistance and capacitance tables used for RC extraction for a specific process corner.

The Extraction Rules file (RC technology file) is generated once for every process node and corner automatically.

The detailed documentation can be found *here*

Limitations

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

Extraction Rules Generation Flow for OpenRCX

This flow generates the RC tech file for OpenRCX. The RC tech file provides resistance and capacitance tables used for RC extraction for a specific process corner.

The flow involves:

A. Running OpenRCX generate_patterns.tcl to generate layout patterns.

- Input: tech LEF
- Output: patterns.def, patterns.v
- Script: generate_patterns.tcl
- Desc: OpenRCX generates many pattern geometries to model various types of capacitance and resistance (i.e., multi-conductor) geometric configurations.

B. Running your favorite extraction tool (i.e., reference extractor) to extract parasitics of the layout patterns.

- Input: patterns.def, patterns.v (if required), and additional files required by the reference extractor.
- Output: patterns.spef
- Script: Not provided
- Desc: Extract parasitics of the patterns generated by OpenRCX using a reference extractor. This one-time step provides the parasitics of various types of pattern geometries as reference for fitted per-unit length R, C calculation.

C. Running OpenRCX to convert patterns.spef to RC tech file.

- Input: patterns.spef
- Output: RC tech file
- Script: generate_rules.tcl
- Desc: OpenRCX takes the .spef from the reference extractor and performs calculations to produce capacitance and resistance tables for a wide range of wire geometries. The output of this flow is a custom RC tech file for OpenRCX.

D. Benchmarking - test the accuracy of OpenRCX on the patterns layout.

- Input: patterns.def and RC tech file
- Output: rcx.spef, diff_spef.out
- Script: ext_patterns.tcl
- Desc: Perform parasitic extraction on pattern layout for the calibration using the generated RC tech file. OpenRCX then compares the extracted parasitics with the golden parasitics that had been extracted by the reference extractor in Step (B) above.

How to run:

1. Go to OpenRCX home directory (./OpenROAD/src/rcx).
2. cd calibration.
3. Modify the user_env.tcl script in the script directory.

- TECH_LEF: points to the directory of the tech LEF
- PROCESS_NODE: the technology node
- extRules: the name and the location of the OpenRCX tech file

1. Run the executable script run.sh –> run Steps (A) through (D) of the flow above.

- source run.sh or
- ./run.sh

1. The OpenRCX RC tech file can be found in the directory that is specified in the extRules variable.

5.4.8 Restructure

Restructure is an interface to ABC for local resynthesis. The package allows logic restructuring that targets area or timing. It extracts a cloud of logic using the OpenSTA timing engine, and passes it to ABC through `blif` interface. Multiple recipes for area or timing are run to obtain multiple structures from ABC; the most desirable among these is used to improve the netlist. The ABC output is read back by a `blif` reader which is integrated to OpenDB. `blif` writer and reader also support constants from and to OpenDB. Reading back of constants requires insertion of tie cells which should be provided by the user as per the interface described below.

Commands

Restructuring can be done in two modes: area or delay.

Area Mode

```
restructure -liberty_file <liberty_file>
            -target "area"
            -tielo_pin  <tielo_pin_name>
            -tiehi_pin  <tiehi_pin_name>
```

Timing Mode

```
restructure -liberty_file <liberty_file>
            -target "delay"
            -slack_threshold <slack_val>
            -depth_threshold <depth_threshold>
            -tielo_pin  <tielo_pin_name>
            -tiehi_pin  <tiehi_pin_name>
```

Argument Description:

- `liberty_file` Liberty file with description of cells used in design. This is passed to ABC.

- `target` could be area or delay. In area mode, the focus is area reduction and timing may degrade. In delay mode, delay is likely reduced but area may increase.
- `-slack_threshold` specifies a (setup) timing slack value below which timing paths need to be analyzed for restructuring.
- `-depth_threshold` specifies the path depth above which a timing path would be considered for restructuring.
- `tielo_pin` specifies the tie cell pin which can drive constant zero. Format is lib/cell/pin
- `tiehi_pin` specifies the tie cell pin which can drive constant one. Format is lib/cell/pin

Example scripts

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

Authors

- Sanjiv Mathur
- Ahmad El Rouby

License

5.4.9 Initialize Floorplan

Commands

Initialize Floorplan

```
initialize_floorplan
  [-site site_name]               LEF site name for ROWS
  -die_area "lx ly ux uy"         die area in microns
  [-core_area "lx ly ux uy"]      core area in microns
or
  -utilization util               utilization (0-100 percent)
  [-aspect_ratio ratio]           height / width, default 1.0
  [-core_space space
    or "bottom top left right"]   space around core, default 0.0 (microns).
                                  Should be either one value for all margins
                                  or 4 values, one for each margin.
```

The die area and core area used to write ROWs can be specified explicitly with the -die_area and -core_area arguments. Alternatively, the die and core areas can be computed from the design size and utilization as shown below:

```
core_area = design_area / (utilization / 100)
core_width = sqrt(core_area / aspect_ratio)
core_height = core_width * aspect_ratio
core = ( core_space_left, core_space_bottom )
      ( core_space_left + core_width, core_space_bottom + core_height )
die =  ( 0, 0 )
      ( core_width + core_space_left + core_space_right,
        core_height + core_space_bottom + core_space_top )
```

Make Tracks

The `initialize_floorplan` command removes existing tracks. Use the `make_tracks` command to add routing tracks to a floorplan.

```
make_tracks [layer]
            [-x_pitch x_pitch]
            [-y_pitch y_pitch]
            [-x_offset x_offset]
            [-y_offset y_offset]
```

With no arguments `make_tracks` adds X and Y tracks for each routing layer. With a `-layer` argument `make_tracks` adds X and Y tracks for layer with options to override the LEF technology X and Y pitch and offset.

Place pins around core boundary

```
auto_place_pins pin_layer_name
```

This command only allow for pins to be on the same layer. For a more complex pin placement strategy please see the pin placement documentation *here*.

Example scripts

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

5.4.10 ioPlacer

Place pins on the boundary of the die on the track grid to minimize net wirelengths. Pin placement also creates a metal shape for each pin using min-area rules.

For designs with unplaced cells, the net wirelength is computed considering the center of the die area as the unplaced cells' position.

Commands

Place All Pins

Use the following command to perform pin placement:

```
place_pins [-hor_layers <h_layers>]
           [-ver_layers <v_layers>]
           [-random_seed <seed>]
           [-exclude <interval>]
           [-random]
           [-group_pins <pins>]
           [-corner_avoidance <length>]
           [-min_distance <distance>]
           [-min_distance_in_tracks]
```

- `-hor_layers` (mandatory). Specify the layers to create the metal shapes of pins placed in horizontal tracks. Can be a single layer or a list of layer names.
- `-ver_layers` (mandatory). Specify the layers to create the metal shapes of pins placed in vertical tracks. Can be a single layer or a list of layer names.
- `-random_seed`. Specify the seed for random operations.
- `-exclude`. Specify an interval in one of the four edges of the die boundary where pins cannot be placed. Can be used multiple times.
- `-random`. When this flag is enabled, the pin placement is random.
- `-group_pins`. Specify a list of pins to be placed together on the die boundary.
- `-corner_avoidance distance`. Specify the distance (in microns) from each corner within which pin placement should be avoided.
- `-min_distance distance`. Specify the minimum distance between pins on the die boundary. This distance can be in microns (default) or in number of tracks between each pin.
- `-min_distance_in_tracks`. Flag that allows setting the min distance in number of tracks instead of microns.

The `exclude` option syntax is `-exclude edge:interval`. The `edge` values are (top|bottom|left|right). The `interval` can be the whole edge, with the * value, or a range of values. For example, in `place_pins -hor_layers metal2 -ver_layers metal3 -exclude top:* -exclude right:15-60.5 -exclude left:*-50` three intervals are excluded: the whole top edge, the right edge from 15 microns to 60.5 microns, and the left edge from its beginning to 50 microns.

Place Individual Pin

```
place_pin [-pin_name <pin_name>]
          [-layer <layer>]
          [-location <{x y}>]
          [-pin_size <{width height}>]
```

The `place_pin` command places a specific pin in the specified location, with the specified size.

- `-pin_name` option is the name of a pin of the design.
- `-layer` defines the routing layer where the pin is placed.
- `-location` defines the center of the pin.
- `-pin_size` option defines the width and height of the pin.

Define Pin Shape Pattern

The `define_pin_shape_pattern` command defines a pin placement grid on the specified layer. This grid has positions inside the die area, not only at the edges of the die boundary.

```
define_pin_shape_pattern [-layer <layer>]
                         [-x_step <x_step>]
                         [-y_step <y_step>]
                         [-region <{llx lly urx ury}>]
                         [-size <{width height}>]
                         [-pin_keepout <dist>]
```

- The `-layer` option defines a single top-most routing layer of the placement grid.
- The `-region` option defines the `{llx, lly, urx, ury}` region of the placement grid.
- The `-x_step` and `-y_step` options define the distance between each valid position on the grid, in the x- and y-directions, respectively.
- The `-size` option defines the width and height of the pins assigned to this grid. The centers of the pins are placed on the grid positions. Pins may have half of their shapes outside the defined region.
- The `-pin_keepout` option defines the boundary (microns) around existing routing obstructions that the pins should avoid; this defaults to the `layer` minimum spacing.

Set IO Pin Constraint

The `set_io_pin_constraint` command sets region constraints for pins according to the pin direction or the pin name. This command can be called multiple times with different constraints. Only one condition should be used for each command call.

The `-direction` argument is the pin direction (input, output, inout, or feedthrough). The `-pin_names` argument is a list of names. The `-region` syntax is the same as that of the `-exclude` syntax.

Note that if you call `define_pin_shape_pattern` before `set_io_pin_constraint`, the `edge` values are (up, top, bottom, left, right). Where `up` relates to the layer created by `define_pin_shape_pattern`. To restrict pins to the pin placement grid defined with `define_pin_shape_pattern` use:

- `-region up:{llx lly urx ury}` to restrict the pins into a specific region in the grid
- `-region up:*` to restrict the pins into the entire region of the grid.

```
set_io_pin_constraint -direction <direction>
                      -pin_names <names>
                      -region <edge:interval>
```

Clear IO Pin Constraints

The `clear_io_pin_constraints` command clears all the previously-defined constraints and pin shape patterns created with `define_pin_shape_pattern`.

```
clear_io_pin_constraints
```

Example scripts

Regression tests

Limitations

External references

- This code depends on Munkres.

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

5.4.11 ICeWall

At the top level of the chip, special padcells are use to connect signals to the external package. Additional commands are provided to *specify the placement of padcells, bondpads and bumps*

The definition of the padring is split into three separate parts:

- A library definition which contains additional required information about the IO cells being used
- A package description which details the location of IO cells around the periphery of the design
- A signal mapping file that associates signals in the design with the IO cells placed around the periphery

The separation of the package description from the signal mapping file allows the same IO padring to be reused for different designs, reducing the amount of rework needed.

For more details refer to *doc/README.md*

ICeWall

Introduction

ICeWall is a utility to place IO cells around the periphery of a design, and associate the IO cells with those present in the netlist of the design.

Additional data is required to define the IO cells in the library over and above that specified in the LEF description, e.g., to:

- Define the orientation of IO cells on each side.
- Allow different cell types to be used for the same type of signal depending upon which side of the die the IO cell is to be placed on. Technologies that require the POLY layer to be aligned in the same direction across the whole die require different cell variants for left/right edges and top/bottom edges.
- Define the pins that connect by abutment through the sides of the padcells.
- Define cells that are used as breaker cells around the padring, defining the signals that they break.
- Define the filler cells and corner cells to be used.

In order to allow for a given padring to be reused across multiple designs, the definition of the placement of padcells is done independently of the signals associated with the IO cells. It is expected that the incoming design will instantiate IO cells for all IO cells that include discrete devices, i.e., including signal IO as well as power/ground pads.

The package data file defines where padcells are to be placed, but the association between the placement and the netlist is not made until the signal mapping file is read in. This separation of placement and netlist information allows the same IO placement to be reused for multiple designs. A package data file can also be extracted from an existing DEF file to allow padrings of existing designs to be reused.

The signal mapping file associates signals in the netlist with locations in the padring.

Data definitions

ICeWall acts upon the information provided in the data files to produce a padring layout. The required data is split across 3 files to allow for the anticipated use models.

Library data

The library data is defined by calling the Footprint library function. This function takes a single argument, which is a TCL dictionary structure that defines the needed definitions of the cells.

The following keys need to be defined in the dictionary:

- types
- connect_by_abutment
- pad_pin_name
- pad_pin_layer
- breakers
- cells

The types entry associates types used in the package data file with cells defined in the cells section, and in addition requires the definition of the corner cell to be used and the list of pad filler cells to be used, ordered by cell width.

The connect_by_abutment key defines the list of pins on the edges of the IO cells that are connected by abutment in forming a ring of these signals around the die. ICeWall will connect these abutted signals together into SPECIAL nets, so that the router will not try to add wiring for these pins.

The pad_pin_name key defines the name of the pin of the IO cell that is connected to the chip package.

The pad_pin_layer key defines the layer that will be used to create a physical pin on the die that is coincident with the location of the external connection to the chip package.

The breakers key defines the list of cell types in the types definition that act as signal breakers for the signals that connect by abutment.

The cells key defines additional data for each IO cell. Each type defined in the types key must have a matching definition in the list of cells. Within a cell definition, the following keys may be used:

- cell_name
- orient
- physical_only
- breaks
- signal_type
- power_pad
- ground_pad
- pad_pin_name

Key	Description
cell_name	Can be either the name of the LEF macro to use, or else a dictionary allowing a different cell name to be specified for each side of the die. Sides are denoted by bottom, right, top, and left.
orient	A dictionary that specifies the orientation of the cell for each side of the die.
physical_only	Cells that are not associated with cells in the design, primarily used for physical connections through the padring.
breaks	For breaker cells, defines the abutment signals which are broken by the presence of this cell. Each signal broken requires a list of pins on the breaker cell for the left and right hand side of the cell. (An empty list may be provided if there are no pins for the broken signals.)
signal_type	Set to 1 if this cell is to be regarded as a signal type.
power_pad	Optional. Set to 1 to define this cell as a power pad.
ground_pad	Optional. Set to 1 to define this cell as a ground pad.
pad_pin_name	Optional. Define the name of the pin on the macro that connects to the chip package.

TCL Command Reference

Details on the *TCL command interface*

Package data

As an alternative to using TCL commands, the definition of the padring can be made using a padring strategy file, which allows for batch loading of the padcell data all at once.

For an example of what a padring strategy file looks like, please refer to …/test/soc_bsg_black_parrot_nangate45/bsg_black_parrot.package.strategy .

Signal mapping

Assignment of signals to padcells can be done via a signal mapping file. This file consists of multiple lines, with each line mapping a named padcell to a signal in the design.

For power and ground pads, the same power/ground signal will likely be associated with multiple padcells.

Example

```
sig132   ddr_dq_10_io
sig133   ddr_dq_9_io
sig134   ddr_dq_8_io
v18_0    DVDD_0
v18_1    DVDD_0
v18_2    DVDD_0
v18_3    DVDD_0
```

add_pad

Synopsis

```
  % add_pad [-name <name>] \
                    [-type <type>] \
                    [-cell <library_cell>] \
                    [-signal <signal_name>] \
                    [-edge (bottom|right|top|left)] \
                    [-location {(center|origin) {x <value> y <value>} [orient␣
↪(R0|R90|R180|R270|MX|MY|MXR90|MYR90)]}] \
                    [-bump {row <number> col <number>}] \
                    [-bondpad {(center|origin) {x <value> y <value>}}] \
                    [-inst_name <instance_name>]
```

Description

Use the add_pad command to add a padcell to the footprint definition. Use this command before calling the init_footprint command. Use this command as a replacement for the padcell strategy file, or to supplement definitions in an existing padcell strategy file.

The -name switch is optional, but should be used when defining signal names in a separate signal map file.

One of the -cell or -type options is required, which will directly or indirectly associate one of the cells in the library definition with this padcell.

The -signal option can be used to associate the signal name of a top-level port with a padcell.

Use the -inst_name option to associate the padcell with a particular instance in the design. For example, when there are several power/ground padcells in the design, using the -inst_name option will associate a particular instance in the netlist to the specified location on the padring. For padcells with no logical equivalent, e.g., ring breaker cells, an instance will be created of the specified name.

The -edge option is used to determine the orientation of the padcell, where the actual orientation of a padcell on a given edge is specified in the library definitions file. The coordinates specified for the location of the padcell are cross-checked against the orientation determined from the -edge option. Similarly, if orient is defined by the -location option, this too will be cross-checked against the value of the edge option. If the -edge option is not defined then the -location setting is used to determine the side on which the padcell appears, hence determining the orientation of the cell. Once again, the orient setting merely serves to act as a cross-check.

Options

Switch_Name	Description
-name	Specify the name of the padcell when using a separate signal assignment file. A name is automatically generated if not specified.
-type	The type of cell specified in the library data.
-cell	
-signal	The name of the signal in the design that is connected to the bondpad/bump of the padcell. Except for nets which have been defined as power or ground nets, only padcell can be associated with a particular signal, and this signal must be a pin at the top level of the design.
-edge	Specify which edge of the design the padcell is to be placed on. Valid choices are bottom, right, top or left.
-location	Specify the location of the center or origin of the padcell.
-bump	For flipchip designs, declare that the padcell is associated with the bump at the specified row/col location on the die.
-bondpad	For wirebond designs where the padcells have separate bondpad instances, use this option to specify the location of the associated bondpad.
-inst_name	Specify the name of the padcell instance in the design. This takes precedence over the -pad_inst_pattern method described in the set_padring_options command.

Examples

```
add_pad -edge bottom                    -signal p_ddr_dm_2_o -type sig   -location
↪{center {x 2742.000 y  105.000}} -bondpad {center {x 2742.000 y   63.293}}

add_pad -edge bottom -inst_name u_vss_0 -signal VSS          -type vss   -location
↪{center {x  397.000 y  105.000}} -bondpad {center {x  397.000 y  149.893}}

add_pad -edge top    -inst_name u_brk0                       -type fbk   -location
↪{center {x 1587.500 y 2895.000}}
```

define_pad_cell

Synopsis

```
 % define_pad_cell \
       [-name name] \
       [-type cell_type|-fill|-corner|-bump|-bondpad] \
       [-cell_name cell_names_per_side] \
       [-orient orientation_per_side] \
       [-pad_pin_name pad_pin_name] \
       [-break_signals signal_list] \
       [-physical_only]
```

Description

In order to create a padring, additional information about the padcells must be provided in order to be able to proceed. The define_pad_cell command is used to specify this additional information.

When specifying the orientation of padcells on a per-side basis, use keys bottom, right, top and left. In the case of specifying corner cells, use keys ll, lr, ur and ul instead.

One or more libcells can be defined as type fill. When filling a gap between padcells, the largest fill cell that is has a width less than or equal to the gap width will be added. If the gap is too small to fit any fill cell, then the smallest fill cell will be added anyway, on the assumption that the library is designed such that the smallest fill cell is allowed to overlap with padcell instances. At least one libcell of type corner must be defined. For a wirebond design, if the library uses separate bondpad instances from the padcells, then a bondpad type must be defined. If the design is flipchip, then one libcell definition must be of type bump.

Options

Option	Description
-name	A name to use for the specification of the libcell. This does not need to match the name of the actual cell name in the library.
-type	Define a type for this cell, which can be used as a reference in the add_pad command.
-fill	Synonymous with -type fill
-corner	Synonymous with -type corner
-bondpad	Synonymous with -type bondpad
-bump	Synonymous with -type bump
-cell_name	Specifies the name of the cell in the library to be used for this padcell. This can either be a single cell name, in which case this cell will always be used, or else it can be a set of key-value pairs, with the key being the side of the die on which the pad is to be placed, and the value being the name of the cell to be used on that side. This allows different cells to be used depending upon the edge of the die on which the cell is to be placed.
-orient	Specifies the orientation of the padcell for each of the edges of the die. This is specified as a list of key-value pairs, with the key being the side of the die, and the value being the orientation to use for that side.
-pad_pin_name	Specifies the name of the pin on the padcell that is used to connect to the top-level pin of the design.
-breaks	For cells which break the signals which connect by abutment through the padring, specifies the names of the signals that are affected. This is specified as a list of key-value pairs, where the key is the name of the signal being broken, and the value is a list of the pins on the left and right hand sides of the breaker cell that break this signal. Specify this as an empty list if the breaker cell does not have pins for broken signals.
-physical_only	Defines the cell to be physical only.

Examples

```
define_pad_cell \
  -name PADCELL_SIG \
  -type sig \
  -cell_name {top PADCELL_SIG_V bottom PADCELL_SIG_V left PADCELL_SIG_H right PADCELL_
↪SIG_H} \
  -orient {bottom R0 right R90 top R180 left R270} \
  -pad_pin_name PAD

define_pad_cell \
  -name PADCELL_VDD \
  -type vdd \
  -cell_name {top PADCELL_VDD_V bottom PADCELL_VDD_V left PADCELL_VDD_H right PADCELL_
↪VDD_H} \
  -orient {bottom R0 right R90 top R180 left R270} \
  -pad_pin_name VDD

define_pad_cell \
```

(continues on next page)

(continued from previous page)

```
  -name PADCELL_CBRK \
  -type cbk \
  -cell_name {bottom PADCELL_CBRK_V right PADCELL_CBRK_H top PADCELL_CBRK_V left PADCELL_
↪CBRK_H} \
  -orient {bottom R0 right R90 top R180 left R270} \
  -break_signals {RETN {RETNA RETNB} SNS {SNSA SNSB}} \
  -physical_only 1

define_pad_cell \
  -name PAD_CORNER \
  -corner \
  -orient {ll R0 lr R90 ur R180 ul R270} \
  -physical_only 1
```

initialize_padring

Synopsis

```
  % initialize_padring [<signal_mapping_file>]
```

Description

Generate the padring placement based on the information supplied about the padcells, their locations (bondpads or bumps), and library cells. If the footprint has been specified without signal mapping, then signal mapping can be done at this stage using the optional signal_mapping_file argument.

Example

```
initialize_padring soc_bsg_black_parrot_nangate45/soc_bsg_black_parrot.sigmap
```

place_cell

Synopsis

```
  % place_cell -inst_name <inst_name> \
                [-cell <library_cell>] \
                -origin <point>] \
                -orient (R0|R90|R180|R270|MX|MY|MXR90|MYR90) \
                [-status (PLACED|FIRM)]
```

Description

Use the place_cell command to place an instance at a specific location within the design. This can be used to pre-place marker cells, ESD cells, etc. which have a known, fixed location in the design and should not be moved by the automatic macro placer.

One of the -inst_name, -origin and -orient options is required. If there is no instance <inst_name> in the netlist of the design, then the -cell option is required, and an instance of the specified <library_cell> will be added to the design. If an instance <inst_name> does exist in the design and the -cell option is specified, it is an error if the instance in the netlist does not match the specified <library_cell>.

Options

Switch_Name	Description
-inst_name	Specify the name of the padcell instance in the design.
-name	Specify the name of the padcell when using a separate signal assignment file. A name is automatically generated if not specified.
-cell	Specify the name of the cell in the library to be added as an instance in the design.
-origin	Specify the origin of the cell as a point, i.e., a list of two numbers.
-orient	Specify a valid orientation for the placement of the cell.

Example

```
place_cell -cell MARKER -inst_name u_marker_0 -origin {1197.5 1199.3} -orient R0 -status␣
↪FIRM
```

set_bump_options

Synopsis

```
% set_bump \
      -row row \
      -col col \
      [(-power|-ground|-net) net_name] \
      [-remove] \
```

Description

The set_bump command is used to provide additional information about specific bumps in the bump array.

The -row and -col options are required and identify the row and column of a specific bump location in the bump array. The bump in the top left corner of the array is row 1, column 1.

The -net, -power and -ground options are mutually exclusive and are used to specify the name of the net connected to a bump and whether it is of type signal, power or ground.

The -remove option is used to specify that no bump should be placed at the specified position in the bump array.

Options

Option	Description
-row	Specifies the row of a specific bump.
-col	Specifies the column of a specific bump.
-net	Specifies the name of the signal net connected to the specified bump.
-power	Specifies the name of the power net connected to the specified bump.
-ground	Specifies the name of the ground net connected to the specified bump.
-remove	Removes the specified bump from the bump array.

Examples

```
set_bump -row  8 -col  8 -power VDD1

set_bump -row  9 -col  8 -remove

set_bump -row 10 -col  8 -power VDD2
```

set_bump_options

Synopsis

```
% set_bump_options \
      [-pitch pitch] \
      [-spacing_to_edge spacing] \
      [-offset {x_offset y_offset}] \
      [-array_size {num_rows num_cols}] \
      [-bump_pin_name pin_name] \
      [-cell_name bump_cell_table] \
      [-num_pads_per_tile value] \
      [-rdl_layer name] \
      [-rdl_width value] \
      [-rdl_spacing value] \
      [-rdl_cover_file_name rdl_file_name]
```

Description

The set_bump_options command is used to provide detailed information about how to handle bumps and the redistribution layer (RDL) connections.

Use the -pitch option to set the center-to-center spacing of bumps.

If the -spacing_to_edge option is specified, then the number of rows and columns of bumps added will be the maximum that can fit in the die with a minimum spacing to the edge of the die as specified. Alternatively, specify the number of rows and columns of bumps using the -array_size option, and use the -offset option to specify the location of the lower left bump on the die.

The name of the cell in the library is specified with the -cell_name option. If the technology supports a number of different bump cells according to bump pitch, then the value for -cell_name can be specified as a list of key-value pairs,

where the key is pitch between bump centers, and the value is the name of the bump cell to be used. The actual cell selected will depend upon the value of the -pitch option.

The name of the pin on the bump is specified using -bump_pin_name.

The -num_pads_per_tile option specifies the number of padcells that can be placed within a single bump pitch. This can be specified as an integer, or as a list of key-value pairs where key and value are defined as for the -cell_name option.

Details about the redistribution layer, name, width and spacing are specified with the -rdl_layer, -rdl_width and -rdl_spacing commands (units: microns) respectively.

The -rdl_cover_file_name is used to specify the name of the file to contain the RDL routing.

Options

Option	Description
-pitch	Specifies the center-to-center spacing of bumps.
-spacing_to_edge	Specifies the spacing from the edge of the die to the edge of the bumps.
-array_size	Specifies the numbers of rows and columns of bumps as a 2-element list.
-offset	Specifies the location of the center of the lower left bump on the die.
-bump_pin_name	Specifies the name of the pin on the bump cell.
-cell_name	Specifies the name of the bump cell, or a list of key-value pairs giving different values of bump cell name with the value of -pitch used as a key.
-num_pads_per_tile	The maximum number of externally connected padcells placed within a bump pitc.h
-rdl_layer	Name of the redistribution layer.
-rdl_width	The width of the RDL layer to use when connecting bumps to padcells.
-rdl_spacing	The required spacing between RDL wires.
-rdl_cover_file_name	Specifies the name of the file to which the routing of the redistribution layer is to be written. If not specified, the default value is cover.def. In an earlier release, the OpenROAD database did not support 45-degree geometries used by RDL routing, and this cover.def allowed for the RDL to be added at the end of the flow, without being added to the database. Now that the database will allow 45-degree geometries, this command will be deprecated once ICeWall has been modified to write RDL layout directly into the database.

Examples

```
set_bump_options \
  -pitch 160 \
  -bump_pin_name PAD \
  -spacing_to_edge 165 \
  -cell_name {140 BUMP_small 150 BUMP_medium 180 BUMP_large} \
  -num_pads_per_tile 5 \
  -rdl_layer metal10 \
  -rdl_width 10 \
  -rdl_spacing 10


set_bump_options \
  -pitch 160 \
  -bump_pin_name PAD \
  -array_size {17 17} \
  -offset {210.0 215.0} \
  -cell_name DUMMY_BUMP \
  -num_pads_per_tile 5 \
  -rdl_layer metal10 \
  -rdl_width 10 \
  -rdl_spacing 10
```

set_padring_options

Synopsis

```
  % set_padring_options \
      -type (flipchip|wirebond) \
      [-power power_nets] \
      [-ground ground_nets] \
      [-offsets offsets] \
      [-pad_inst_pattern pad_inst_pattern] \
      [-pad_pin_pattern pad_pin_pattern] \
      [-pin_layer pin_layer_name] \
      [-connect_by_abutment signal_list]
```

Description

This command is used to specify general options used to define the padring for a chip. The -type option is required; all others are optional.

The -power and -ground options are used to define the external power and ground nets that will be connected to the pads.

The -offsets option is used to define the offset of the edge of the pads from the edge of the die area. This can be specified as a single number, in which case the offset is applied to all four sides, or as a list of 2 numbers that are respectively applied to the top/bottom and left/right edges, or as a list of 4 numbers that are respectively applied to the bottom, right, top and left edges.

The -pad_inst_pattern option allows a format string to be used to define a default instance name for the padcell connected to a top-level pin of the design.

The -pad_pin_pattern option is used to define a pattern that is to be used to derive the actual signal name in the design from the signal specified with the add_pad command.

The -connect_by_abutment option is used to define the list of signals that are connected by abutment through the padring. The placement of breaker cells within the padring can result in these signals being split into a number of different nets.

Options

Option	Description
-type	Specify whether the chip is wirebond or flipchip.
-power	Define the nets to be used as power nets in the design. It is not required that these nets exist in the design database; they will be created as necessary. Once a power net is defined, it can be used as the -signal argument for the `add_pad` command to enable the addition of power pads to the design.
-ground	Define the nets to be used as ground nets in the design, it is not required that these nets exist in the design database, they will be created as necessary. Once a ground net is defined, it can be used as the -signal argument for the `add_pad` command to enable the addition of ground pads to the design.
-core_area	Specify the coordinates, as a list of 4 numbers (in microns), of the chip core. This area is reserved for stdcell placements. Placement of padcells should be made outside this core area, but within the specified die area.
-die_area	Specify the coordinates, as a list of 4 numbers (in microns), for the chip die area.
-offsets	Specifies the offset, in microns, from the edges of the die area to corresponding edges of the padcells.
-pad_inst_pattern	Specify the name of padcell instances based upon the given format string. The format string is expected to include %s somewhere within the string, which will be replaced by the signal name associated with each pad instance.
-pad_pin_pattern	Specify the name of the signal to connect to the padcell based upon the given format string. The format string is expected to include %s somewhere within the string, which will be replaced by the signal name associated with the pad instance.
-pin_layer	Specify the layer which is to be used to create a top level pin over the pin of the padcell. The creation of a physical pin at this location identifies pin locations for LVS (layout-versus-schematic) verification.
-connect_by_abutment	Specify the list of signals that connect by abutment in the padring. The placement of breaker cells in the padring will split these signals into separate nets as required.

Examples

```
set_padring_options \
  -type wirebond \
  -power  {VDD DVDD_0 DVDD_1} \
  -ground {VSS DVSS_0 DVSS_1} \
  -offsets 35 \
  -pad_inst_pattern "u_%s" \
  -pad_pin_pattern "p_%s" \
  -pin_layer metal10 \
  -connect_by_abutment {SNS RETN DVDD DVSS}
```

TCL Commands to build chip level padrings

The following commands can be used from the command prompt to define the padring structure for the chip.

- *add_pad*
- *define_pad_cell*
- *set_padring_options*
- *set_bump_options*
- *set_bump*

Once the padcells have been added, the padring can be built using the initialize_padring command.

- *initialize_padring*

Use the place_cell command to pre-place additional cells in the floorplan.

- *place_cell*

Full examples of how to use these commands together to build a chip:

- Example for chip with wirebond padring
- Example for chip with flipchip bumps

Alternatively, the information needed to build chip-level padrings can be *bundled up into a separate file and loaded in batch fashion*

5.4.12 Macro Placement

ParquetFP-based macro cell placer, "TritonMacroPlacer". The macro placer places macros/blocks honoring halos, channels and cell row "snapping". Run `global_placement` before macro placement.

Approximately ceil((#macros/3)^(3/2)) sets corresponding to quadrisections of the initial placed mixed-size layout are explored and packed using ParquetFP-based annealing. The best resulting floorplan according to a heuristic evaluation function is kept.

Commands

```
macro_placement [-halo {halo_x halo_y}]
                [-channel {channel_x channel_y}]
                [-fence_region {lx ly ux uy}]
                [-snap_layer snap_layer_number]
```

- `-halo` horizontal/vertical halo around macros (microns)
- `-channel` horizontal/vertical channel width between macros (microns)
- `-fence_region` - restrict macro placements to a region (microns). Defaults to the core area.
- `-snap_layer_number` - snap macro origins to this routing layer track

Macros will be placed with `max(halo * 2, channel)` spacing between macros, and between macros and the fence/die boundary. If no solutions are found, try reducing the channel/halo.

Example scripts

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License.

5.4.13 Tapcell

Tapcell and endcap insertion.

Commands

```
tapcell [-tapcell_master tapcell_master]
        [-endcap_master endcap_master]
        [-distance dist]
        [-halo_width_x halo_x]
        [-halo_width_y halo_y]
        [-tap_nwin2_master tap_nwin2_master]
        [-tap_nwin3_master tap_nwin3_master]
        [-tap_nwout2_master tap_nwout2_master]
        [-tap_nwout3_master tap_nwout3_master]
        [-tap_nwintie_master tap_nwintie_master]
        [-tap_nwouttie_master tap_nwouttie_master]
        [-cnrcap_nwin_master cnrcap_nwin_master]
        [-cnrcap_nwout_master cnrcap_nwout_master]
        [-incnrcap_nwin_master incnrcap_nwin_master]
        [-incnrcap_nwout_master incnrcap_nwout_master]
        [-tap_prefix tap_prefix]
        [-endcap_prefix endcap_prefix]
```

Example scripts

You can find script examples for both 45nm and 14nm in `tap/etc/scripts`

Limitations

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

5.4.14 pdn

This utility aims to simplify the process of adding a power grid into a floorplan. The aim is to specify a small set of power grid policies to be applied to the design, such as layers to use, stripe width and spacing, then have the utility generate the actual metal straps. Grid policies can be defined over the stdcell area, and over areas occupied by macros.

For more details refer to *doc/README.md*

OpenROAD-pdn

This utility aims to simplify the process of adding a power grid into a floorplan. The aim is to specify a small set of power grid policies to be applied to the design, such as layers to use, stripe width and spacing, then have the utility generate the actual metal straps. Grid policies can be defined over the stdcell area, and over areas occupied by macros.

Installation and Setup

This package runs as a utility within the `openraod` application.

Usage

From inside the `openroad` application, the Power Delivery Network Generation utility can be invoked as follows:

```
% pdngen <configuration_file> [-verbose]
```

All inputs and power grid policies are specified in a TCL format in the configuration file.

```
% pdngen PDN.cfg -verbose
```

Configuration File

For further information on the configuration file, and to review an example configuration see the following:

- *PDN configuration help*
- Sample configuration - PDN.cfg
- Sample grid configuration - grid_strategy-M1-M4-M7.cfg

Assumptions and Limitations

Currently the following assumptions are made:

1. The design is rectangular
2. The input floorplan includes the stdcell rows, placement of all macro blocks and IO pins.
3. The stdcells rows will be cut around macro placements

Elements of a configuration file

A configuration file for apply_pdn consists of the following parts

1. Global variable definitions
2. Call a TCL procedure to create grid specifications

Optional Global Variables

Global variables are prefixed by `::` and force the variable to be declared in the global scope, rather than the local scope. This makes the values of these variables available everywhere in the TCL code. The `apply_pdn` command requires the following global variables to exist

Variable Name	Description
`::power_nets`	Name of the power net
`::ground_nets`	Name of the ground net
`::rails_start_with`	POWER|GROUND
`::stripes_start_with`	POWER|GROUND
`::halo`	Halo to apply around macros. Specify one, two or four values. If a HALO is defined in the floorplan DEF, then this will be ignored.

Power grid strategy definitions

A set of power grid specifications are provided by calling the `pdngen::specify_grid` command. At least one power grid specification must be defined.

The command has the following form:

```
pdngen::specify_grid (stdcell|macro) <specification>
```

The specification is a list of key-value pairs:

Key	Description
`rails`	The specification of the layer(s) which should be used to draw the horizontal stdcell rails. Each layer will specify a value for `width` and optionally `offset`.
`straps`	List of layers on which to draw stripes. Each layer will specify a value for `width`, `offset`, `pitch` and optionally `spacing`. By default `spacing = pitch / 2`.
`connect`	List of connections to be made between layers. Macro pin connections are on layer <layer_name>*PIN*<direction>.

Additionally, for macro grids:

Key	Description
`orient`	If the orientation of the macro matches an entry in this list, then apply this grid specification.
`instance`	If the instance name of the macro matches an entry in this list, then apply this grid specification.
`macro`	If the macro name of the macro matches an entry in this list, then apply this grid specification.
`power_pins`	List of power pins on the macro to connect to the grid.
`ground_pins`	List of ground pins on the macro to connect to the grid.
`blockages`	Layers which are blocked by a macro using this grid specification.

The key connect specifies two layers to be connected together wherever stripes of a given net on the first layer overlap with stripes of the same net on the second layer.

Macro pins are extracted from the LEF/DEF and are specified on a layer called <layer_name>_PIN_<dir>, where <layer_name> is the name of the layer and <dir> is `hor` to indicate pins that are horizontally-oriented in the floorplan and `ver` to indicate pins that are vertically-oriented in the floorplan.

A separate grid is built for each macro. The list of macro specifications that have been defined is searched to find a specification with a matching instance name key; failing that, a macro specification with a matching macro name key, or else the first specification with neither an instance nor a macro key, is used. Furthermore, if orient is specified, then the orientation of the macro must match one of the entries in the orient field.

Examples of grid specifications

1. Stdcell grid specification

```
pdngen::specify_grid stdcell {
    name grid
    rails {
        metal1 {width 0.17}
    }
    straps {
        metal4 {width 0.48 pitch 56.0 offset 2}
        metal7 {width 1.40 pitch 40.0 offset 2}
    }
    connect {{metal1 metal4} {metal4 metal7}}
}
```

This specification adds a grid over the stdcell area, with a metal1 followpin width of 0.17, connecting to metal4 stripes of 0.48um every 56.0um, connecting in turn to metal7 stripes, also 1.40um wide and with 40.0um pitch.

1. Macro grid specification

```
pdngen::specify_grid macro {
    orient {R0 R180 MX MY}
    power_pins "VDD VDDPE VDDCE"
    ground_pins "VSS VSSE"
    blockages "metal1 metal2 metal3 metal4 metal5 metal6"
    straps {
        metal5 {width 0.93 pitch 10.0 offset 2}
        metal6 {width 0.93 pitch 10.0 offset 2}
    }
    connect {{metal4_PIN_ver metal5} {metal5 metal6} {metal6 metal7}}
}
```

If this the only macro grid specification defined, then it will be applied over all the macros in the design that match one of the entries in the orient field.

All vertical metal4 pins on the macros are connected to metal5, which is then connected to metal6, which is connected in turn to metal7.

For macros that have their pins oriented in non-preferred routing direction the grid specification would be as follows.

We define blockages on metal1 to metal6: although the macro itself only blocks from metal1 to metal4, we need to consider metal5 and metal6 to be blocked in case the stdcell grid specification tries to use those layers for its power straps. In that case we need those straps to be cut in order to keep the space for the macro grid.

```
pdngen::specify_grid macro {
    orient {R90 R270 MXR90 MYR90}
    power_pins "VDD VDDPE VDDCE"
    ground_pins "VSS VSSE"
    blockages "metal1 metal2 metal3 metal4 metal5 metal6"
    straps {
        metal6 {width 0.93 pitch 10.0 offset 2}
    }
    connect {{metal4_PIN_hor metal6} {metal6 metal7}}
}
```

Macros with orientations R90, R270, MXR90 or MYR90 will have their metal4 pins in the vertical (non-preferred) direction; the above specification connects these pins directly to the metal6 layer, then on to metal7.

In both of these cases we have specified that the macro pins VDD, VDDPE and VDDCE will be connected to the power net, and the pins VSS and VSSE will be connected to the ground net. Any straps specified for other grids will be blocked for layers metal1 to metal6. No actual blockage is added to the design in this case.

Connection constraints

Further constraints can be applied for the connections between layers that have been specified.

Key	Description
`cut_pitch`	Specify a value greater than minimum cut pitch, e.g., for connecting a dual layer stdcell rail.
`max_rows`	For a large via, limit the maximum number of rows allowed.
`max_columns`	For a large via, limit the maximum number of columns allowed.
`split_cuts`	Use individual vias instead of a via array.
`ongrid`	Force intermediate metal layers to be drawn on the routing grid.

Examples

```
pdngen::specify_grid stdcell {
    name grid
    rails {
        metal1 {width 0.17 pitch  2.4 offset 0}
        metal2 {width 0.17 pitch  2.4 offset 0}
    }
    connect {
      {metal1 metal2 constraints {cut_pitch 0.16}}
    }
}
```

Using fixed VIAs from the tech file

Normally pdngen uses VIARULEs defined in the LEF, along with design rules for spacing, enclosure, etc. to build vias to connect between the layers. Some technologies may not have VIARULEs set up in the LEF file, while other technologies may define a set of fixed vias to be used in the power grid. In such cases, one must use the fixed_vias property to specify a list of fixed vias to be used for a given connection. The specified fixed_vias will be used to connect between layers. For any required vias that do not have any specified fixed_vias, pdngen will revert to building a via from a VIARULE instead.

Example

```
pdngen::specify_grid stdcell {
    name grid
    rails {
        metal1 {width 0.17 pitch  2.4 offset 0}
        metal2 {width 0.17 pitch  2.4 offset 0}
    }
    connect {
      {metal1 metal2 fixed_vias VIA12}
      {metal2 metal5 fixed_vias {VIA23 VIA34}}
    }
}
```

Additional features for top-level power grid

At the top level of the SoC, there is often a requirement to connect the core power and ground pads to the core power grid. This is done by specifying a core power/ground ring between the stdcell area and the padcell placement.

Example

```
pdngen::specify_grid stdcell {
  name grid

  pwr_pads {PADCELL_VDD_V PADCELL_VDD_H}
  gnd_pads {PADCELL_VSS_V PADCELL_VSS_H}

  core_ring {
    metal9  {width 5.0 spacing 2.0 core_offset 4.5}
    metal10 {width 5.0 spacing 2.0 core_offset 4.5}
  }
  rails {
    metal1  {width 0.17 pitch  2.4 offset 0}
  }
  straps {
    metal4  {width 0.48 pitch 56.0 offset 2}
    metal7  {width 1.40 pitch 40.0 offset 2}
    metal8  {width 1.40 pitch 40.0 offset 2}
    metal9  {width 1.40 pitch 40.0 offset 2}
    metal10 {width 1.40 pitch 40.0 offset 2}
```

(continues on next page)

(continued from previous page)

```
  }
  connect {
    {metal1 metal4}
    {metal4 metal7}
    {metal7 metal8}
    {metal8 metal9}
    {metal9 metal10}
  }
}
```

When inserting a grid for a hierarchical sub-block, the top layers are omitted to be added at the SoC level. So, at the SoC level, we have straps for layers metal8 to metal10. We also specify core rings for two of the layers. The core rings are constructed as concentric rings around the stdcell area using the specified metals in their preferred routing directions. Straps from the stdcell area in the these layers will be extended out to connect to the core rings.

The stdcell rails may also be extended out to the core rings by using the extend_to_core_rings property in the rail definition.

Example

```
pdngen::specify_grid stdcell {
  name grid

  core_ring {
    metal4  {width 5.0 spacing 2.0 core_offset 4.5}
    metal5  {width 5.0 spacing 2.0 core_offset 4.5}
  }
  rails {
    metal1  {width 0.17 pitch  2.4 offset 0 extend_to_core_ring 1}
  }
  straps {
    metal4  {width 0.48 pitch 56.0 offset 2}
  }
  connect {
    {metal1 metal4}
  }
}
```

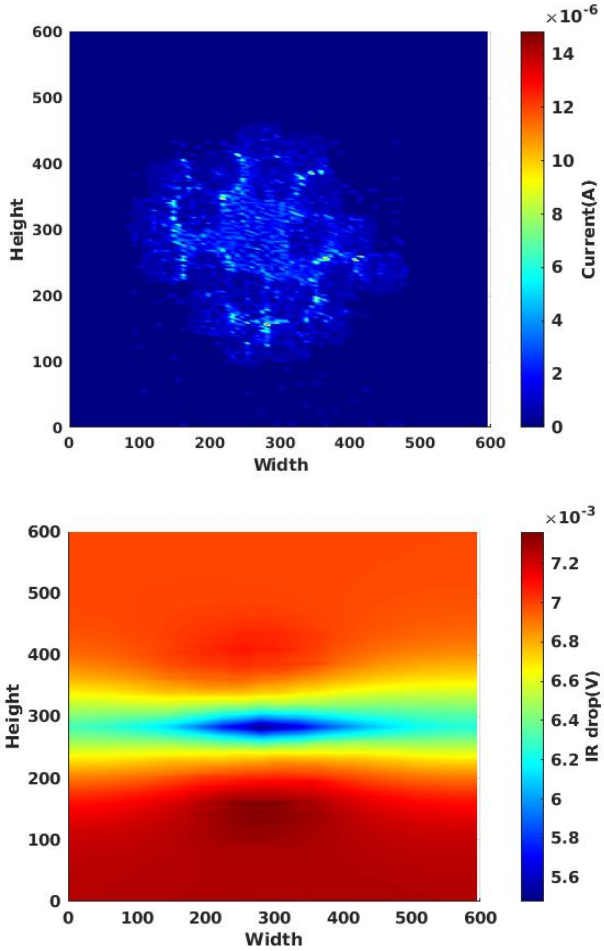
It is assumed that the power pads specified have core-facing pins for power/ground in the same layers that are used in the core rings. Straps having the same width as the pins are used to connect to the core rings. These connections from the padcells have priority over connections from the core area power straps.

5.4.15 PDNSim

PDNSim is an open-source static IR analyzer.

Features:

- Report worst IR drop.
- Report worst current density over all nodes and wire segments in the power distribution network, given a placed and PDN-synthesized design.
- Check for floating PDN stripes on the power and ground nets.
- Spice netlist writer for power distribution network wire segments.



Commands

```
set_pdnsim_net_voltage -net <net_name> -voltage <voltage_value>
check_power_grid -net <net_name>
analyze_power_grid -vsrc <voltage_source_location_file> \
                   -net <net_name> \
                   [-outfile <filename>] \
                   [-enable_em] \
                   [-em_outfile <filename>]
                   [-dx]
                   [-dy]
                   [-em_outfile <filename>]
write_pg_spice -vsrc <voltage_source_location_file> -outfile <netlist.sp> -net <net_name>
```

Options description:

- `vsrc`: (optional) file to set the location of the power C4 bumps/IO pins. Vsrc_aes.loc file for an example with a description specified here.
- `dx,dy`: (optional) these arguments set the bump pitch to decide the voltage source location in the absence of a vsrc file. Default bump pitch of 140um used in absence of these arguments and vsrc
- `net`: (mandatory) is the name of the net to analyze, power or ground net name
- `enable_em`: (optional) is the flag to report current per power grid segment
- `outfile`: (optional) filename specified per-instance voltage written into file
- `em_outfile`: (optional) filename to write out the per segment current values into a file, can be specified only if enable_em is flag exists
- `voltage`: Sets the voltage on a specific net. If this command is not run, the voltage value is obtained from operating conditions in the liberty.

Example scripts

See the file Vsrc_aes.loc file for an example with a description specified *here*.

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

Cite this Work

If you find PDNSim useful, please use the following `bibtex` to cite it:

```
@misc{pdnsim,
author = "V. A. Chhabria and S. S. Sapatnekar",
title={{PDNSim}},
note= "\url{https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/PDNSim}"
}
```

License

BSD 3-Clause License. See LICENSE file.

Voltage source location file description

This file specifies the description of the C4 bump configurations file. The file is a csv as described below:

```
<x_coordinate>, <y_coordinate>, <octagonal_c4_bump_edge_length>, <voltage_value>
```

The x and y coordinate specify the center location of the voltage C4 bumps in micro meter.

The octagonal c4_edge_length specifies the edge length of the C4 to determine the pitch of the RDL layer in micron

Voltage_value specifies the value of voltage source at the C4 bump. In case there is a need to specify voltage drop in micron

Example file

250,250,20,1.1 130,170,20,1.1 370,410,10,1.1 410,450,10,1.1

5.4.16 Parallax Static Timing Analyzer

OpenSTA is a gate level static timing verifier. As a stand-alone executable it can be used to verify the timing of a design using standard file formats.

- Verilog netlist
- Liberty library
- SDC timing constraints
- SDF delay annotation
- SPEF parasitics

OpenSTA uses a TCL command interpreter to read the design, specify timing constraints and print timing reports.

Clocks

- Generated
- Latency
- Source latency (insertion delay)
- Uncertainty
- Propagated/Ideal
- Gated clock checks
- Multiple frequency clocks

Exception paths

- False path
- Multicycle path
- Min/Max path delay
- Exception points
- -from clock/pin/instance -through pin/net -to clock/pin/instance
- Edge specific exception points
- -rise_from/-fall_from, -rise_through/-fall_through, -rise_to/-fall_to

Delay calculation

- Integrated Dartu/Menezes/Pileggi RC effective capacitance algorithm
- External delay calculator API

Analysis

- Report timing checks -from, -through, -to, multiple paths to endpoint
- Report delay calculation
- Check timing setup

Timing Engine

OpenSTA is architected to be easily bolted on to other tools as a timing engine. By using a network adapter, OpenSTA can access the host netlist data structures without duplicating them.

- Query based incremental update of delays, arrival and required times
- Simulator to propagate constants from constraints and netlist tie high/low

See doc/OpenSTA.pdf for command documentiaton. See doc/StaApi.txt for timing engine API documentiaton. See doc/ChangeLog.txt for changes to commands.

Build

OpenSTA is built with CMake.

Prerequisites

The build dependency versions are show below. Other versions may work, but these are the versions used for development.

	from	Ubuntu	Xcode
		18.04.1	11.3
cmake	3.10.2	3.10.2	3.16.2
clang	9.1.0		11.0.0
gcc	3.3.2	7.3.0	
tcl	8.4	8.6	8.6.6
swig	1.3.28	3.0.12	4.0.1
bison	1.35	3.0.4	3.5
flex	2.5.4	2.6.4	2.5.35

Note that flex versions before 2.6.4 contain 'register' declarations that are illegal in c++17.

These packages are **optional**:

libz	1.1.4	1.2.5	1.2.8
cudd		2.4.1	3.0.0

CUDD is a binary decision diageram (BDD) package that is used to improve conditional timing arc handling. OpenSTA does not require it to be installed. It is available here or here.

Note that the file hierarchy of the CUDD installation changed with version 3.0. Some changes to CMakeLists.txt are required to support older versions.

Use the USE_CUDD option to look for the cudd library. Use the CUDD_DIR option to set the install directory if it is not in one of the normal install directories.

When building CUDD you may use the `--prefix` option to `configure` to install in a location other than the default (`/usr/local/lib`).

```
cd $HOME/cudd-3.0.0
mkdir $HOME/cudd
./configure --prefix $HOME/cudd
make
make install

cd <opensta>/build
cmake .. -DUSE_CUDD -DCUDD_DIR=$HOME/cudd
```

The Zlib library is an optional. If CMake finds libz, OpenSTA can read Verilog, SDF, SPF, and SPEF files compressed with gzip.

Installing with CMake

Use the following commands to checkout the git repository and build the OpenSTA library and excutable.

```
git clone https://github.com/The-OpenROAD-Project/OpenSTA.git
cd OpenSTA
mkdir build
cd build
cmake ..
make
```

The default build type is release to compile optimized code. The resulting executable is in `app/sta`. The library without a `main()` procedure is `app/libSTA.a`.

Optional CMake variables passed as -D= arguments to CMake are show below.

```
CMAKE_BUILD_TYPE DEBUG|RELEASE
CMAKE_CXX_FLAGS - additional compiler flags
TCL_LIBRARY - path to tcl library
TCL_HEADER - path to tcl.h
CUDD - path to cudd installation
ZLIB_ROOT - path to zlib
CMAKE_INSTALL_PREFIX
```

If `TCL_LIBRARY` is specified the CMake script will attempt to locate the header from the library path.

The default install directory is `/usr/local`. To install in a different directory with CMake use:

```
cmake .. -DCMAKE_INSTALL_PREFIX=<prefix_path>
```

If you make changes to `CMakeLists.txt` you may need to clean out existing CMake cached variable values by deleting all of the files in the build directory.

Run using Docker

OpenSTA can be run as a Docker container.

- Install Docker on Windows, Mac or Linux.
- Navigate to the directory where you have the input files.
- Run OpenSTA as a binary using

```
docker run -it -v $(pwd):/data openroad/opensta
```

From the interactive terminal, use OpenSTA commands. You can read input files from `/data` directory inside the docker container (e.g. `read_liberty /data/liberty.lib`). You can use OpenSTA in non-interactive mode by passing a command file using the `-f` flag as follows.

```
docker run -it -v $(pwd):/data openroad/opensta /data/cmd_file
```

Note that the path after `-f` is the path inside container, not on the guest machine.

Bug Reports

Use the Issues tab on the github repository to report bugs.

Each issue/bug should be a separate issue. The subject of the issue should be a short description of the problem. Attach a test case to reproduce the issue as described below. Issues without test cases are unlikely to get a response.

The files in the test case should be collected into a directory named YYYYMMDD where YYYY is the year, MM is the month, and DD is the day (this format allows "ls" to report them in chronological order). The contents of the directory should be collected into a compressed tarfile named YYYYMMDD.tgz.

The test case should have a tcl command file recreates the issue named run.tcl. If there are more than one command file using the same data files, there should be separate command files, run1.tcl, run2.tcl etc. The bug report can refer to these command files by name.

Command files should not have absolute filenames like "/home/cho/OpenSTA_Request/write_path_spice/dump_spice" in them. These obviously are not portable. Use filenames relative to the test case directory.

Authors

- James Cherry
- William Scott authored the arnoldi delay calculator at Blaze, Inc which was subsequently licensed to Nefelus, Inc that has graciously contributed it to OpenSTA.

Contributions

External code contributions are not supported.

https://en.wikipedia.org/wiki/Contributor_License_Agreement https://opensource.google/docs/cla/

License

OpenSTA is dual licensed. It is released under GPL v3 as OpenSTA and is also licensed for commerical applications by Parallax Software without the GPL's requirements.

5.4.17 Gate Resizer

Gate Resizer commands are described below. The `resizer` commands stop when the design area is `-max_utilization util` percent of the core area. `util` is between 0 and 100. The `resizer` stops and reports an error if the max utilization is exceeded.

Commands

Set Wire RC

```
set_wire_rc [-clock] [-signal]
            [-layer layer_name]
            [-resistance res]
            [-capacitance cap]
```

The `set_wire_rc` command sets the resistance and capacitance used to estimate delay of routing wires. Separate values can be specified for clock and data nets with the `-signal` and `-clock` flags. Without either `-signal` or `-clock` the resistance and capacitance for clocks and data nets are set. Use `-layer` or `-resistance` and `-capacitance`. If `-layer` is used, the LEF technology resistance and area/edge capacitance values for the layer are used for a minimum-width wire on the layer. The resistance and capacitance values are per length of wire, not per square or per square micron. The units for `-resistance` and `-capacitance` are from the first Liberty file read, resistance_unit/distance_unit (typically kohms/micron) and Liberty capacitance_unit/distance_unit (typically pf/micron or ff/micron). If distance units are not specified in the Liberty file, microns are used.

Set Layer RC

The `set_layer_rc` command can be used to set the resistance and capacitance for a layer or via. This is useful if these values are missing from the LEF file, or to override the values in the LEF.

```
set_layer_rc [-layer layer]
             [-via via_layer]
             [-capacitance cap]
             [-resistance res]
             [-corner corner]
```

For layers the resistance and capacitance units are the same as `set_wire_rc` (per length of minimum-width wire). `layer` must be the name of a routing layer.

Via resistance can also be set with the `set_layer_rc` command, using the -via keyword. `-capacitance` is not supported for vias. `via_layer` is the name of a via layer. Via resistance is per cut/via, not area based.

Remove Buffers

```
remove_buffers
```

Use the `remove_buffers` command to remove buffers inserted by synthesis. This step is recommended before using `repair_design` so that there is more flexibility in buffering nets.

Estimate Parasitics

```
estimate_parasitics -placement|-global_routing
```

Estimate RC parasitics based on placed component pin locations. If there are no component locations, then no parasitics are added. The resistance and capacitance values are per distance unit of a routing wire. Use the `set_units` command to check units or `set_cmd_units` to change units. The goal is to represent "average" routing layer resistance and capacitance. If the set_wire_rc command is not called before resizing, then the default_wireload model specified in the first Liberty file read or with the SDC set_wire_load command is used to make parasitics.

After the `global_route` command has been called, the global routing topology and layers can be used to estimate parasitics with the `-global_routing` flag.

Set Don't Use

```
set_dont_use lib_cells
```

The `set_dont_use` command removes library cells from consideration by the `resizer`. `lib_cells` is a list of cells returned by `get_lib_cells` or a list of cell names (`wildcards` allowed). For example, `DLY*` says do not use cells with names that begin with `DLY` in all libraries.

Buffer Ports

```
buffer_ports [-inputs] [-outputs] [-max_utilization util]
```

The `buffer_ports -inputs` command adds a buffer between the input and its loads. The `buffer_ports -outputs` adds a buffer between the port driver and the output port. The default behavior is `-inputs` and `-outputs` if neither is specified.

Repair Design

```
repair_design [-max_wire_length max_length]
              [-max_slew_margin slew_margin]
              [-max_cap_margin cap_margin]
              [-max_utilization util]
```

The `repair_design` command inserts buffers on nets to repair max slew, max capacitance and max fanout violations, and on long wires to reduce RC delay in the wire. It also resizes gates to normalize slews. Use `estimate_parasitics -placement` before `repair_design` to estimate parasitics considered during repair. Placement-based parasitics cannot accurately predict routed parasitics, so a margin can be used to "over-repair" the design to compensate. Use `-max_slew_margin` to add a margin to the slews, and `-max_cap_margin` to add a margin to the capacitances. Use `-max_wire_length` to specify the maximum length of wires. The maximum wirelength defaults to a value that minimizes the wire delay for the wire resistance/capacitance values specified by `set_wire_rc`.

Set Max Fanout

Use the `set_max_fanout` SDC command to set the maximum fanout for the design.

```
set_max_fanout <fanout> [current_design]
```

Repair Tie Fanout

```
repair_tie_fanout [-separation dist]
                  [-verbose]
                  lib_port
```

The `repair_tie_fanout` command connects each tie high/low load to a copy of the tie high/low cell. `lib_port` is the tie high/low port, which can be a library/cell/port name or object returned by `get_lib_pins`. The tie high/low instance is separated from the load by `dist` (in Liberty units, typically microns).

Repair Timing

```
repair_timing [-setup]
              [-hold]
              [-slack_margin slack_margin]
              [-allow_setup_violations]
              [-max_utilization util]
              [-max_buffer_percent buffer_percent]
```

The `repair_timing` command repairs setup and hold violations. It should be run after clock tree synthesis with propagated clocks. While repairing hold violations buffers are not inserted that will cause setup violations unless '-allow_setup_violations' is specified. Use `-slack_margin` to add additional slack margin. To specify different slack margins use separate `repair_timing` commands for setup and hold. Use `-max_buffer_percent` to specify a maximum number of buffers to insert to repair hold violations as a percentage of the number of instances in the design. The default value for `buffer_percent` is 20, for 20%.

Report Design Area

```
report_design_area
```

The `report_design_area` command reports the area of the design's components and the utilization.

Report Floating Nets

```
report_floating_nets [-verbose]
```

The `report_floating_nets` command reports nets with only one pin connection. Use the `-verbose` flag to see the net names.

Example scripts

A typical `resizer` command file (after a design and Liberty libraries have been read) is shown below.

```
read_sdc gcd.sdc

set_wire_rc -layer metal2

set_dont_use {CLKBUF_* AOI211_X1 OAI211_X1}

buffer_ports
repair_design -max_wire_length 100
repair_tie_fanout LOGIC0_X1/Z
repair_tie_fanout LOGIC1_X1/Z
# clock tree synthesis...
repair_timing
```

Note that OpenSTA commands can be used to report timing metrics before or after resizing the design.

```
set_wire_rc -layer metal2
report_checks
report_tns
report_wns
report_checks

repair_design

report_checks
report_tns
report_wns
```

Regression tests

Limitations

FAQs

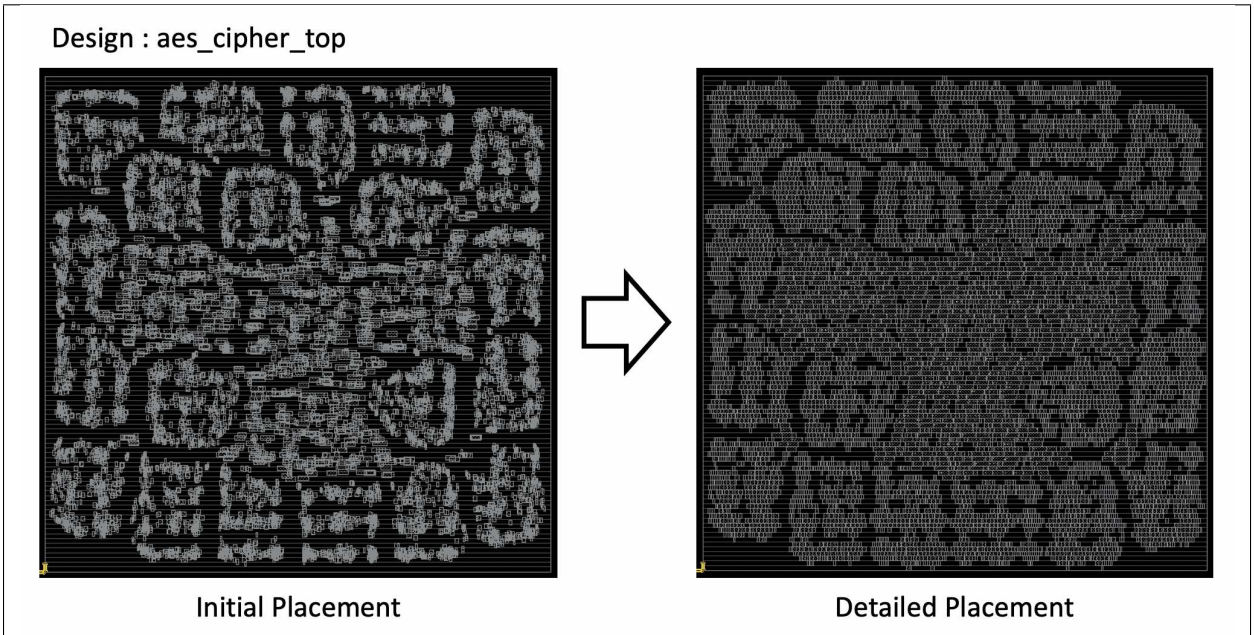
Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

5.4.18 OpenDP

OpenDP: Open-Source Detailed Placement Engine



Visualized detailed placement result using `aes_cipher_top` design with Innovus initial placement; doc/OpenDP.jpg

Features:

- Fence region and multi-height cells. (ICCAD 2017 Contest benchmarks)
- Fragmented ROWs.
- Macro blocks.

Commands

The `detailed_placement` command performs detailed placement of instances to legal locations after global placement.

```
set_placement_padding -global|-instances insts|-masters masters
                      [-left pad_left] [-right pad_right]
detailed_placement [-max_displacement disp|{disp_x disp_y}]
check_placement [-verbose]
filler_placement [-prefix prefix] filler_masters
optimize_mirroring
```

The `set_placement_padding` command sets left and right padding in multiples of the row site width. Use the `set_placement_padding` command before legalizing placement to leave room for routing. Use the `-global` flag for padding that applies to all instances. Use `-instances` for instance-specific padding. The instances `insts` can be a list of instance names, or an instance object returned by the SDC `get_cells` command. To specify padding for all instances of a common master, use the `-filter` "ref_name == " option to `get_cells`.

The `set_power_net` command is used to set the power and ground special net names. The defaults are `VDD` and `VSS`.

The `-max_displacement` argument to `detailed_placement` specifies how far an instance can be moved when finding a site where it can be placed. The default values are `{500 100}` sites. The x/y displacement arguments are in microns.

The `check_placement` command checks the placement legality. It returns `0` if the placement is legal.

The `filler_placement` command fills gaps between detail-placed instances to connect the power and ground rails in the rows. `filler_masters` is a list of master/macro names to use for filling the gaps. Wildcard matching is supported, so `FILL*` will match, e.g., `FILLCELL_X1 FILLCELL_X16 FILLCELL_X2 FILLCELL_X32 FILLCELL_X4 FILLCELL_X8`. To specify a different naming prefix from `FILLER_` use `-prefix <new prefix>`.

The `optimize_mirroring` command mirrors instances about the Y axis in a weak attempt to reduce total wirelength (HPWL).

Example scripts

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

Authors

- SangGi Do and Mingyu Woo (respective Ph. D. advisors: Seokhyeong Kang, Andrew B. Kahng).
- Rewrite and port to OpenDB/OpenROAD by James Cherry, Parallax Software
- Paper reference: S. Do, M. Woo and S. Kang, "Fence-Region-Aware Mixed-Height Standard Cell Legalization", Proc. Great Lakes Symposium on VLSI, 2019, pp. 259-262. (link)

License

BSD 3-Clause License. See LICENSE file.

5.4.19 TritonCTS 2.0

TritonCTS 2.0 is available under the OpenROAD app as `clock_tree_synthesis` command. The following tcl snippet shows how to call TritonCTS. TritonCTS 2.0 performs on-the-fly characterization. Thus there is no need to generate characterization data. On-the-fly characterization feature could still be optionally controlled by parameters specified to `configure_cts_characterization` command. Use `set_wire_rc` command to set clock routing layer.

Commands

Configure CTS Characterization

```
configure_cts_characterization [-max_slew <max_slew>] \
                               [-max_cap <max_cap>] \
                               [-slew_inter <slew_inter>] \
                               [-cap_inter <cap_inter>]
```

Argument description:

- `-max_slew` is the max slew value (in seconds) that the characterization will test. If this parameter is omitted, the code would use max slew value for specified buffer in `buf_list` from liberty file.
- `-max_cap` is the max capacitance value (in farad) that the characterization will test. If this parameter is omitted, the code would use max cap value for specified buffer in `buf_list` from liberty file.
- `-slew_inter` is the time value (in seconds) that the characterization will consider for results. If this parameter is omitted, the code gets the default value (`5.0e-12`). Be careful that this value can be quite low for bigger technologies (>`65nm`).
- `-cap_inter` is the capacitance value (in farad) that the characterization will consider for results. If this parameter is omitted, the code gets the default value (`5.0e-15`). Be careful that this value can be quite low for bigger technologies (>`65nm`).

Clock Tree Synthesis

```
clock_tree_synthesis -buf_list <list_of_buffers> \
                     [-root_buf <root_buf>] \
                     [-wire_unit <wire_unit>] \
                     [-clk_nets <list_of_clk_nets>] \
                     [-out_path <lut_path>] \
                     [-post_cts_disable] \
                     [-distance_between_buffers] \
                     [-branching_point_buffers_distance] \
                     [-clustering_exponent] \
                     [-clustering_unbalance_ratio] \
                     [-sink_clustering_enable] \
                     [-sink_clustering_size <cluster_size>] \
                     [-sink_clustering_max_diameter <max_diameter>]
```

Argument description:

- `-buf_list` are the master cells (buffers) that will be considered when making the wire segments.
- `-root_buffer` is the master cell of the buffer that serves as root for the clock tree. If this parameter is omitted, the first master cell from `-buf_list` is taken.
- `-wire_unit` is the minimum unit distance between buffers for a specific wire. If this parameter is omitted, the code gets the value from ten times the height of `-root_buffer`.
- `-clk_nets` is a string containing the names of the clock roots. If this parameter is omitted, TritonCTS looks for the clock roots automatically.
- `-out_path` is the output path (full) that the `lut.txt` and `sol_list.txt` files will be saved. This is used to load an existing characterization, without creating one from scratch.

- `-post_cts_disable` is a flag that, when specified, disables the post-processing operation for outlier sinks (buffer insertion on 10% of the way between source and sink).
- `-distance_between_buffers` is the distance (in micron) between buffers that TritonCTS should use when creating the tree. When using this parameter, the clock tree algorithm is simplified, and only uses a fraction of the segments from the LUT.
- `-branching_point_buffers_distance` is the distance (in micron) that a branch has to have in order for a buffer to be inserted on a branch end-point. This requires the `-distance_between_buffers` value to be set.
- `-clustering_exponent` is a value that determines the power used on the difference between sink and means on the CKMeans clustering algorithm. If this parameter is omitted, the code gets the default value (4).
- `-clustering_unbalance_ratio` is a value that determines the maximum capacity of each cluster during CK-Means. A value of 50% means that each cluster will have exactly half of all sinks for a specific region (half for each branch). If this parameter is omitted, the code gets the default value (0.6).
- `-sink_clustering_enable` enables pre-clustering of sinks to create one level of sub-tree before building H-tree. Each cluster is driven by buffer which becomes end point of H-tree structure.
- `-sink_clustering_size` specifies the maximum number of sinks per cluster. Default value is 20.
- `sink_clustering_max_diameter` specifies maximum diameter (in micron) of sink cluster. Default value is 50.
- `-clk_nets` is a string containing the names of the clock roots. If this parameter is omitted, TritonCTS looks for the clock roots automatically.

Report CTS

Another command available from TritonCTS is `report_cts`. It is used to extract metrics after a successful `clock_tree_synthesis` run. These are: Number of Clock Roots, Number of Buffers Inserted, Number of Clock Subnets, and Number of Sinks. The following tcl snippet shows how to call `report_cts`.

```
report_cts [-out_file <file>]
```

Argument description:

- `-out_file` (optional) is the file containing the TritonCTS reports. If this parameter is omitted, the metrics are shown on the standard output.

Example scripts

```
clock_tree_synthesis -root_buf "BUF_X4" \
                     -buf_list "BUF_X4" \
                     -wire_unit 20

report_cts "file.txt"
```

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

External references

- LEMON - **L**ibrary for **E**fficient **M**odeling and **O**ptimization in **N**etworks
- Capacitate k-means package from Dr. Jiajia Li (UCSD). Published here.

Authors

TritonCTS 2.0 is written by Mateus Fogaça, PhD student in the Graduate Program on Microelectronics from the Federal University of Rio Grande do Sul (UFRGS), Brazil. Mr. Fogaça advisor is Prof. Ricardo Reis

Many guidance provided by (alphabetic order):

- Andrew B. Kahng
- Jiajia Li
- Kwangsoo Han
- Tom Spyrou

License

BSD 3-Clause License. See LICENSE file.

CTS LUT characterization

January 2021

CTS LUT characterization has been removed from the system. Characterization is now done automatically on the fly during clock tree synthesis. This uses technology parameters from LEF metal layer specified by "set_wire_rc -clock" and buffer cap/slew values from liberty file. Characterization uses OpenSTA interface to generate timing information required to complete characterization table required during CTS clock building process.

5.4.20 FastRoute

FastRoute is an open-source global router originally derived from Iowa State University's FastRoute4.1.

Commands

```
global_route [-guide_file out_file]
             [-verbose verbose]
             [-congestion_iterations iterations]
             [-grid_origin {x y}]
             [-allow_congestion]
```

Options description:

- `guide_file`: Set the output guides file name (e.g., -guide_file `route.guide`)
- `verbose`: Set verbosity of reporting: 0 for less verbosity, 1 for medium verbosity, 2 for full verbosity (e.g., -verbose 1)
- `congestion_iterations`: Set the number of iterations made to remove the overflow of the routing (e.g., `-congestion_iterations 50`)
- `grid_origin`: Set the (x, y) origin of the routing grid in DBU. For example, `-grid_origin {1 1}` corresponds to the die (0, 0) + 1 DBU in each x-, y- direction.
- `allow_congestion`: Allow global routing results to be generated with remaining congestion

```
set_routing_layers [-signal min-max]
                   [-clock min-max]
```

The `set_routing_layers` command sets the minimum and maximum routing layers for signal nets, with the `-signal` option, and the minimum and maximum routing layers for clock nets, with the `-clock` option.

Example: `set_routing_layers -signal Metal2-Metal10 -clock Metal6-Metal9`

```
set_macro_extension extension
```

The `set_macro_extension` command sets the number of `GCells` added to the blockages boundaries from macros. A `GCell` is typically defined in terms of `Mx` routing tracks. The default `GCell` size is 15 `M3` pitches.

Example: `set_macro_extension 2`

```
set_global_routing_layer_adjustment layer adjustment
```

The `set_global_routing_layer_adjustment` command sets routing resource adjustments in the routing layers of the design. Such adjustments reduce the number of routing tracks that the global router assumes to exist. This promotes the spreading of routing and reduces peak congestion, to reduce challenges for detailed routing.

You can set adjustment for a specific layer, e.g., `set_global_routing_layer_adjustment Metal4 0.5` reduces the routing resources of routing layer `Metal4` by 50%. You can also set adjustment for all layers at once using *, e.g., `set_global_routing_layer_adjustment * 0.3` reduces the routing resources of all routing layers by 30%. And, you can also set resource adjustment for a layer range, e.g.: `set_global_routing_layer_adjustment Metal4-Metal8 0.3` reduces the routing resources of routing layers `Metal4`, `Metal5`, `Metal6`, `Metal7` and `Metal8` by 30%.

```
set_routing_alpha [-net net_name] alpha
```

By default the global router uses heuristic rectilinear Steiner minimum trees (RSMTs) as an initial basis to construct route guides. An RSMT tries to minimize the total wirelength needed to connect a given set of pins. The Prim-Dijkstra heuristic is an alternative net topology algorithm that supports a trade-off between total wirelength and maximum path depth from the net driver to its loads. The `set_routing_alpha` command enables the Prim/Dijkstra algorithm and

sets the alpha parameter used to trade-off wirelength and path depth. Alpha is between 0.0 and 1.0. When alpha is 0.0 the net topology minimizes total wirelength (i.e. capacitance). When alpha is 1.0 it minimizes longest path between the driver and loads (i.e., maximum resistance). Typical values are 0.4-0.8. For more information about PDRev, check the paper [here](in https://vlsicad.ucsd.edu/Publications/Journals/j18.pdf). You can call it multiple times for different nets.

Example: `set_routing_alpha -net clk 0.3` sets the alpha value of 0.3 for net *clk*.

```
set_global_routing_region_adjustment {lower_left_x lower_left_y upper_right_x upper_
↪right_y}
                                     -layer layer -adjustment adjustment
```

The `set_global_routing_region_adjustment` command sets routing resource adjustments in a specific region of the design. The region is defined as a rectangle in a routing layer.

Example: `set_global_routing_region_adjustment {1.5 2 20 30.5} -layer Metal4 -adjustment 0.7`

```
set_global_routing_random [-seed seed]
                          [-capacities_perturbation_percentage percent]
                          [-perturbation_amount value]
```

The `set_global_routing_random` command enables randomization of global routing results. The randomized global routing shuffles the order of the nets and randomly subtracts or adds to the capacities of a random set of edges. The `-seed` option sets the random seed and is required to enable the randomization mode. The `-capacities_perturbation_percentage` option sets the percentage of edges whose capacities are perturbed. By default, the edge capacities are perturbed by adding or subtracting 1 (track) from the original capacity. The `-perturbation_amount` option sets the perturbation value of the edge capacities. This option will only have meaning and effect when `-capacities_perturbation_percentage` is used. The random seed must be different from 0 to enable randomization of the global routing.

Example: `set_global_routing_random -seed 42 -capacities_perturbation_percentage 50 -perturbation_amount 2`

```
repair_antennas diodeCellName/diodePinName [-iterations iterations]
```

The repair_antenna command evaluates the global routing results to find antenna violations, and repairs the violations by inserting diodes. The input for this command is the diode cell and its pin names, and a prescribed number of iterations. By default, the command runs only one iteration to repair antennas. It uses the `antennachecker` tool to identify any nets with antenna violations and, for each such net, the exact number of diodes necessary to fix the antenna violation.

Example: `repair_antenna sky130_fd_sc_hs__diode_2/DIODE`

```
write_guides file_name
```

The `write_guides` generates the guide file from the routing results.

Example: `write_guides route.guide`.

To estimate RC parasitics based on global route results, use the `-global_routing` option of the `estimate_parasitics` command.

```
estimate_parasitics -global_routing
```

Example scripts

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

External references

- The algorithm base is from FastRoute4.1, and the database comes from OpenDB
- FastRoute 4.1 documentation. The FastRoute4.1 version was received from mailto:yuexu@iastate.edu on June 15, 2019, with the BSD-3 open source license as given in the FastRoute website.
- Min Pan, Yue Xu, Yanheng Zhang and Chris Chu. "FastRoute: An Efficient and High-Quality Global Router. VLSI Design, Article ID 608362, 2012." Available here.
- P-D paper is C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng and D. Karger, "Prim-Dijkstra Tradeoffs for Improved Performance-Driven Global Routing", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14(7) (1995), pp. 890-896. Available here.

License

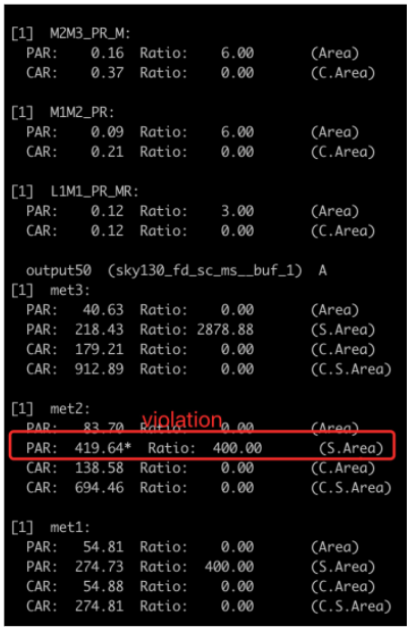
BSD 3-Clause License. See LICENSE file.

5.4.21 Antenna Rule Checker

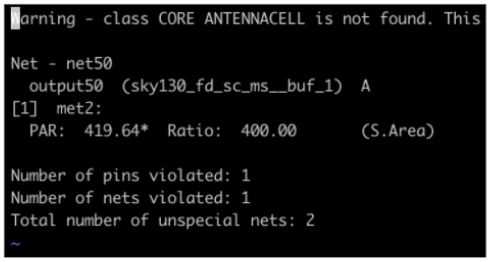
This tool checks antenna violations of a design in OpenDB, and generates a report to indicate violated nets. APIs are provided to help fix antenna violations during the diode insertion flow in global route.

Antenna Report Example

This is an example of the detailed and simple reports of the antenna checker:



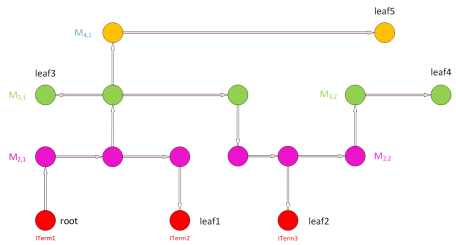
Full report



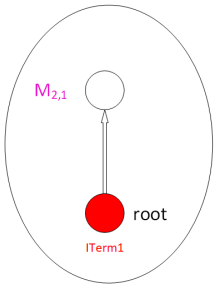
Simple report

Abbreviations Index:

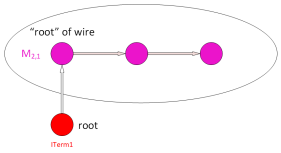
- `PAR`: Partial Area Ratio
- `CAR`: Cumulative Area Ratio
- `Area`: Gate Area
- `S. Area`: Side Diffusion Area
- `C. Area`: Cumulative Gate Area
- `C. S. Area`: Cumulative Side (Diffusion) Area



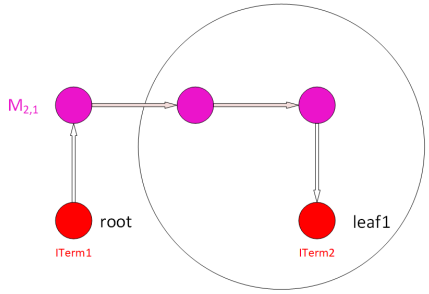
Antenna Checker Algorithm: WireGraph Example



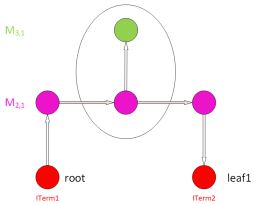
Step 1: (a) Start from the root node (ITerm) using upper Via to find a node for a new wire. (b) Save the ITerm area for cumulative gate/diffusion area.



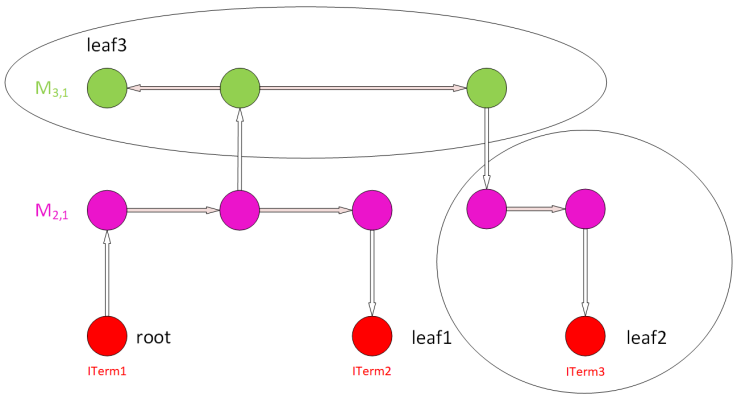
Step 2: From the node of the wire, find all the nodes in the wire through segment wires and find the "root" node of this wire.



Step 3: (a) From the "root" node of the wire, along the outgoing segment edge that goes to other nodes belonging to this wire, calculate the area of this wire. (b) Then, find all the ITerms below these nodes, except for the root node (directly use an ITerm or lower Vias to find ITerms for lower metals). © Sum up the areas of all the ITerms found with the cumulative areas and calculate the PAR of this wire. (d) Add the PAR value and the wire info (layer, Index) into the PAR table. Add the new area to the cumulative areas.



Step 4: Find all the upper Vias on this wire (for all the nodes on this wire), and go to the higher-level metal.



Step 5: Repeat Steps 2 and 3 for new-found upper-level wires.

Commands

check_antennas

Check antenna violations on all nets and generate a report.

```
check_antennas [-report_filename <FILE>] [-report_violating_nets]
```

- `-report_filename`: specifies the filename path where the antenna violation report is to be saved.
- `-report_violating_nets`: provides a summary of the violated nets.

Limitations

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

5.4.22 TritonRoute

TritonRoute is an open-source detailed router for modern industrial designs. The router consists of several main building blocks, including pin access analysis, track assignment, initial detailed routing, search and repair, and a DRC engine. The initial development of the router is inspired by the ISPD-2018 initial detailed routing contest. However, the current framework differs and is built from scratch, aiming for an industrial-oriented scalable and flexible flow.

TritonRoute provides industry-standard LEF/DEF interface with support of ISPD-2018 and ISPD-2019 contest-compatible route guide format.

Commands

Example scripts

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

References

Please cite the following paper(s) for publication:

- A. B. Kahng, L. Wang and B. Xu, "TritonRoute: The Open Source Detailed Router", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020), doi:10.1109/TCAD.2020.3003234.
- A. B. Kahng, L. Wang and B. Xu, "The Tao of PAO: Anatomy of a Pin Access Oracle for Detailed Routing", Proc. ACM/IEEE Design Automation Conf., 2020, pp. 1-6.

Authors

TritonRoute was developed by graduate students Lutong Wang and Bangqi Xu from UC San Diego, and serves as the detailed router in the OpenROAD project.

License

BSD 3-Clause License. See LICENSE file.

5.4.23 Metal fill

This module inserts floating metal fill shapes to meet metal density design rules while obeying DRC constraints. It is driven by a `json` configuration file.

Commands

```
% density_fill -rules <json_file> [-area <list of lx ly ux uy>]
```

If `-area` is not specified, the core area will be used.

Example scripts

The rules `json` file controls fill and you can see an example here.

The schema for the `json` is:

```
{
  "layers": {
    "<group_name>": {
      "layers": "<list of integer gds layers>",
      "names": "<list of name strings>",
      "opc": {
        "datatype":  "<list of integer gds datatypes>",
        "width":   "<list of widths in microns>",
        "height":   "<list of heightsin microns>",
        "space_to_fill": "<real: spacing between fills in microns>",
        "space_to_non_fill": "<real: spacing to non-fill shapes in microns>",
        "space_line_end": "<real: spacing to end of line in microns>"
      },
      "non-opc": {
        "datatype":  "<list of integer gds datatypes>",
```

(continues on next page)

(continued from previous page)

```
        "width":   "<list of widths in microns>",
        "height":   "<list of heightsin microns>",
        "space_to_fill": "<real: spacing between fills in microns>",
        "space_to_non_fill": "<real: spacing to non-fill shapes in microns>"
      }
    }, ...
  }
}
```

The `opc` section is optional depending on your process.

The width/height lists are effectively parallel arrays of shapes to try in left to right order (generally larger to smaller).

The layer grouping is for convenience. For example in some technologies many layers have similar rules so it is convenient to have a `Mx`, `Cx` group.

This all started out in `klayout` so there are some obsolete fields that the parser accepts but ignores (e.g., `space_to_outline`).

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

License

BSD 3-Clause License. See LICENSE file.

5.4.24 Flute3

Flute3 is an open-source rectilinear Steiner minimum tree heuristic with improvements made by UFRGS students and James Cherry. The version in this repository uses CMake and C++ namespace, and has dynamic memory allocation. Flute3 can handle nets with any degree.

External references (Optional)

The algorithm base is Flute3.1, extracted from the FastRoute4.1 version that was received from mailto:yuexu@iastate.edu on June 15, 2019, with the BSD-3 open source license as given in the FastRoute website.

License

See LICENSE file.

5.4.25 UCSD Prim-Dijkstra Revisited

```
Usage: ./pd_rev -h
    ------------------------------------------------------
      UCSD Prim-Dijkstra Revisited
      coded by Kwangsoo Han and Sriram Venkatesh
    ------------------------------------------------------
    -v       verbose mode (1 - 4)
    -a1      alpha for PD (default: 0.3)
    -a2      alpha for PD-II (default: 0.45)
    -bu      enable PD-II (default: off)
    -m1      Upperbound of max pathlength degradation in DAS wirelength minimization
             (default: 1.1 --> allow 10% max PL degradation for WL optimization)
    -f       input file with pointset
    -hv      enable HoVW Steinerization and Detour-Aware Steinerization (default off)
```

Sample input file: test.in

```
Net FE_OFN298734_n20421 10 (Net NetName NumOfTerminals)
0 10672450 199765 (Index x_loc y_loc, index 0 is root)
1 10769055 425415
2 10861105 425270
3 10734350 308990
4 10762195 340095
5 10741025 374295
6 10844360 425415
7 10862030 415010
8 10724765 254415
9 10745900 326235
```

Example runs:

1. run Prim-Dijkstra with alpha = 0.3

```
./pd_rev -f ./test.in -a1 0.3
```

1. run Prim-Dijkstra with alpha = 0.3 and PD-II with alpha = 0.45

```
./pd_rev -f ./test.in -a1 0.3 -bu 1 -a2 0.45
```

1. run Prim-Dijkstra with alpha = 0.3 and PD-II with alpha = 0.45 and HVW Steinerization and DAS

```
./pd_rev -f ./test.in -a1 0.3 -bu 1 -a2 0.45 -hv 1
```

External references

- C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu and S. Venkatesh, "Prim-Dijkstra Revisited: Achieving Superior Timing-driven Routing Trees", Proc. ACM/IEEE Intl. Symp. on Physical Design, 2018, pp. 10-17.

Authors

- Kwangsoo Han
- Andrew B. Kahng
- Sriram Venkatesh

Contact: kwhan@ucsd.edu, abk@ucsd.edu, srvenkat@ucsd.edu

Affiliation: Computer Science and Engineering Department, UC San Diego, La Jolla, CA 92093-0404, USA

License

5.4.26 PartitionMgr

PartitionMgr is a tool that performs partitioning/clustering on a specific netlist. It provides a wrapper of three well-known permissively open-sourced tools: Chaco, GPMetis and MLPart.

Commands

PartitionMgr offers six commands: partition_netlist, evaluate_partitioning, write_partitioning_to_db, cluster_netlist, write_clustering_to_db and report_netlist_partitions.

partition_netlist

Divides the netlist into N partitions and returns the id (partition_id) of the partitioning solution. The command may be called many times with different parameters. Each time, the command will generate a new solution.

The following Tcl snippet shows how to call partition_netlist:

```
partition_netlist -tool <name>
                  -num_partitions <value>
                  [-graph_model <name>]
                  [-clique_threshold <value>]
                  [-weight_model <name>]
                  [-max_edge_weight <value>]
                  [-max_vertex_weight range]
                  [-num_starts <value>]
                  [-random_seed <value>]
                  [-seeds <value>]
                  [-balance_constraint <value>]
                  [-coarsening_ratio <value>]
                  [-coarsening_vertices <value>]
```

(continues on next page)

(continued from previous page)

```
                [-enable_term_prop <value>]
                [-cut_hop_ratio <value>]
                [-architecture <value>]
                [-refinement <value>]
                [-partition_id <value>]
```

Argument description:

- `tool` (Mandatory) defines the partitioning tool. Can be "chaco", "gpmetis" or "mlpart".
- `num_partitions` (Mandatory) defines the final number of partitions. Can be any integer greater than 1.
- `graph_model` is the hypergraph to graph decomposition approach. Can be "clique", "star" or "hybrid".
- `clique_threshold` is the maximum degree of a net decomposed with the clique net model. If using the clique net model, nets with degree higher than the threshold are ignored. In the hybrid net model, nets with degree higher than the threshold are decomposed using the star model.
- `weight_model` is the edge weight scheme for the graph model of the netlist. Can be any integer from 1 to 7.
- `max_edge_weight` defines the maximum weight of an edge.
- `max_vertex_weight` defines the maximum weight of a vertex.
- `num_starts` is the number of solutions generated with different random seeds.
- `random_seed` is the seed used when generating new random set seeds.
- `seeds` is the number of solutions generated with set seeds.
- `balance_constraint` is the maximum vertex area percentage difference between two partitions. E.g., a 50% difference means that neither partition in a 2-way partitioning can contain less than 25% or more than 75% of the total vertex area in the netlist.
- `coarsening_ratio` defines the minimal acceptable reduction in the number of vertices in the coarsening step.
- `coarsening_vertices` defines the maximum number of vertices that the algorithm aims to have in its most-coarsened graph or hypergraph.
- `enable_term_prop` enables terminal propagation (Dunlop and Kernighan, 1985), which aims to improve data locality. This adds constraints to the Kernighan-Lin (KL) algorithm. Improves the number of edge cuts and terminals, at the cost of additional runtime.
- `cut_hop_ratio` controls the relative importance of generating a new cut edge versus increasing the interprocessor distance associated with an existing cut edge (tradeoff of data locality versus cut edges). This is largely specific to Chaco.
- `architecture` defines the topology (for parallel processors) to be used in the partitioning. These can be 2D or 3D topologies, and they define the total number of partitions. This is largely specific to Chaco.
- `refinement` specifies how many times a KL refinement is run. Incurs a modest runtime hit, but can generate better partitioning results.
- `partition_id` is the partition_id (output from partition_netlist) from a previous computation. This is used to generate better results based on past results or to run further partitioning.

evaluate_partitioning

Evaluates the partitioning solution(s) based on a specific objective function. This function is run for each partitioning solution that is supplied in the partition_ids parameter (return value from partition_netlist) and returns the best partitioning solution according to the specified objective (i.e., metric).

```
evaluate_partitioning -partition_ids <id> -evaluation_function <function>
```

Argument description:

- `-partition_ids` (mandatory) are the partitioning solution id's. These are the return values from the partition_netlist command. They can be a list of values or only one id.
- `-evaluation_function` (mandatory) is the objective function that is evaluated for each partitioning solution. It can be `terminals`, `hyperedges`, `size`, `area`, `runtime`, or `hops`.

The following Tcl snippet shows how to call evaluate_partitioning:

```
set id [ partition_netlist -tool chaco -num_partitions 4 -num_starts 5 ]

evaluate_partitioning -partition_ids $id -evaluation_function function
```

write_partitioning_to_db

Writes the partition id of each instance (i.e., the cluster that contains the instance) to the DB as a property called `partitioning_id`.

The following Tcl snippet shows how to call `write_partitioning_to_db`:

```
write_partitioning_to_db -partitioning_id <id> [-dump_to_file <file>]
```

Argument description:

- `-partitioning_id` (Mandatory) is the partitioning solution id. These are the return values from the partition_netlist command.
- `-dump_to_file` is the file where the vertex assignment results will be saved. The assignment results consist of lines that each contain a vertex name (e.g. an instance) and the partition it is part of.

Another Tcl snippet showing the use of `write_partitioning_to_db` is:

```
set id [partition_netlist -tool chaco -num_partitions 4 -num_starts 5]
evaluate_partitioning -partition_ids $id -evaluation_function "hyperedges"
write_partitioning_to_db -partitioning_id $id -dump_to_file "file.txt"
```

write_partition_verilog

Writes the partitioned network to a Verilog file containing modules for each partition.

```
write_partition_verilog -partitioning_id <id>
                        [-port_prefix <prefix>]
                        [-module_suffix <suffix>]
                        <file.v>
```

Argument description:

- `partitioning_id` (Mandatory) is the partitioning solution id. These are the return values from the partition_netlist command.
- `filename` (Mandatory) is the path to the Verilog file.
- `port_prefix` is the prefix to add to the internal ports created during partitioning; default is `partition_`.
- `module_suffix` is the suffix to add to the submodules; default is `_partition`.

The following Tcl snippet shows how to call write_partition_verilog:

```
set id [partition_netlist -tool chaco -num_partitions 4 -num_starts 5 ]
evaluate_partitioning -partition_ids $id -evaluation_function "hyperedges"
write_partition_verilog -partitioning_id $id -port_prefix prefix -module_suffix suffix␣
↪filename.v
```

cluster_netlist

Divides the netlist into N clusters and returns the id (cluster_id) of the clustering solution. The command may be called many times with different parameters. Each time, the command will generate a new solution. (Note that when we partition a netlist, we typically seek N = O(1) partitions. On the other hand, when we cluster a netlist, we typically seek N = Theta(|V|) clusters.)

```
cluster_netlist -tool name
                [-coarsening_ratio value]
                [-coarsening_vertices value]
                [-level value]
```

Argument description:

- `tool` (Mandatory) defines the multilevel partitioning tool whose recursive coarsening is used to induce a clustering. Can be “chaco”, “gpmetis”, or “mlpart”.
- `coarsening_ratio` defines the minimal acceptable reduction in the number of vertices in the coarsening step.
- `coarsening_vertices` defines the maximum number of vertices that the algorithm aims to coarsen a graph to.
- `level` defines which is the level of clustering to return.

write_clustering_to_db

Writes the cluster id of each instance (i.e., the cluster that contains the instance) to the DB as a property called `cluster_id`.

```
write_clustering_to_db -clustering_id <id> [-dump_to_file <name>]
```

Argument description:

- `clustering_id` (Mandatory) is the clustering solution id. These are the return values from the cluster_netlist command.
- `dump_to_file` is the file where the vertex assignment results will be saved. The assignment results consist of lines that each contain a vertex name (e.g. an instance) and the cluster it is part of.

The following Tcl snippet shows how to call write_clustering_to_db:

```
set id [cluster_netlist -tool chaco -level 2 ]
write_clustering_to_db -clustering_id $id -dump_to_file name
```

report_netlist_partitions

Reports the number of partitions for a specific partition_id and the number of vertices present in each one.

```
report_netlist_partitions -partitioning_id <id>
```

Argument description:

- `partitioning_id` (Mandatory) is the partitioning solution id. These are the return values from the partition_netlist command.

The following Tcl snippet shows how to call report_netlist_partitions:

```
set id [partition_netlist -tool chaco -num_partitions 4 -num_starts 5 ]
evaluate_partitioning -partition_ids $id -evaluation_function "hyperedges"
report_netlist_partitions -partitioning_id $id
```

Regression tests

Limitations

FAQs

Check out GitHub discussion about this tool.

External references

- Chaco.
- GPMetis.
- MLPart.

Authors

PartitionMgr is written by Mateus Fogaça and Isadora Oliveira from the Federal University of Rio Grande do Sul (UFRGS), Brazil, and Marcelo Danigno from the Federal University of Rio Grande (FURG), Brazil.

Mateus's and Isadora's advisor is Prof. Ricardo Reis; Marcelo's advisor is Prof. Paulo Butzen.

Many guidance provided by:

- Andrew B. Kahng
- Tom Spyrou

License

BSD 3-Clause License. See LICENSE file.

5.5 Getting Started with OpenROAD Flow

OpenROAD Flow is a full RTL-to-GDS flow built entirely on open-source tools. The project aims for automated, no-human-in-the-loop digital circuit design with 24-hour turnaround time.

5.5.1 Setup

System Requirements

To build the binaries and run `gcd` through the flow:

- Minimum: 1 CPU core and 4GB RAM.
- Recommend: 4 CPU cores and 16GB of RAM.

NOTE

`gcd` is a small design, and thus requires less computational power. Larger designs may require better hardware.

Building and Installing the Software

There are currently two options to set up the OpenROAD Flow:

- Build from sources using Docker, *instructions here*.
- Build from sources locally, *instructions here*.

5.5.2 Running a Design

Sample design configurations are available in the `designs` directory. You can select a design using either of the following methods:

1. The flow Makefile contains a list of sample design configurations at the top of the file. Uncomment the respective line to select the design.
2. Specify the design using the shell environment. For example:

```
make DESIGN_CONFIG=./designs/nangate45/swerv/config.mk
# or
export DESIGN_CONFIG=./designs/nangate45/swerv/config.mk
make
```

By default, the `gcd` design is selected using the `nangate45` platform. The resulting GDS will be available at `flow/results/nangate45/gcd/6_final.gds`. The flow should take only a few minutes to produce a GDS for this design. We recommend implementing this design first to validate your flow and tool setup.

Adding a New Design

To add a new design, we recommend looking at the included designs for examples of how to set one up.

5.5.3 Platforms

OpenROAD-flow-scripts supports Verilog to GDS for the following open platforms:

- ASAP7
- Nangate45 / FreePDK45
- SKY130

These platforms have a permissive license which allows us to redistribute the PDK and OpenROAD platform-specific files. The platform files and license(s) are located in `platforms/{platform}`.

OpenROAD-flow-scripts also supports the following commercial platforms:

- GF12
- TSMC65LP

The PDKs and platform-specific files for these kits cannot be provided due to NDA restrictions. However, if you are able to access these platforms, you can create the necessary platform-specific files yourself.

Once the platform is set up, you can create a new design configuration with information about the design. See sample configurations in the `design` directory.

Adding a New Platform

At this time, we recommend looking at the Nangate45 as an example of how to set up a new platform for OpenROAD-flow-scripts.

5.5.4 Implement the Design

Run `make` to perform Verilog to GDS. The final output will be located at `flow/results/{platform}/{design_name}/6_final.gds`

5.5.5 Miscellaneous

nangate45 smoke-test harness for top-level Verilog designs

1. Drop your Verilog files into designs/src/harness
2. Start the workflow:

TIP!

Start with a very small submodule in your design that has only a few pins.

```
make DESIGN_NAME=TopLevelName DESIGN_CONFIG=`pwd`/designs/harness.mk
```

5.5.6 Build from sources using Docker

Prerequisites

For this method you only need to install Docker on your machine.

WARNING

The `build_openroad.sh` will use the host number of CPUs to compile `openroad`.

Please check your Docker daemon setup to make sure all host CPUs are available. If you are not sure, you can check with the command below. If the output number is different from the number of CPUs from your machine, then is recommended that you restrict the number of CPUs used by the scripts (see instructions below).

```
docker run <IMAGE> nproc
# <IMAGE> can be any commonly used OS, e.g., 'centos:centos7'
docker run centos:centos7 nproc
```

You can restrict the number of CPUs with the `-t|--threads N` argument:

```
./build_openroad.sh --threads N
```

Clone and Build

```
git clone --recursive https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts
cd OpenROAD-flow-scripts
./build_openroad.sh
```

Verify Installation

The binaries are only available from inside the Docker container, thus to start one use:

```
docker run -it -u $(id -u ${USER}):$(id -g ${USER}) -v $(pwd)/flow/platforms:/OpenROAD-
↪flow-scripts/flow/platforms:ro openroad/flow-scripts
```

Then, inside docker:

```
source ./setup_env.sh
yosys -help
openroad -help
cd flow
make
exit
```

5.5.7 Build from sources locally

Prerequisites

OpenROAD

Dependencies for OpenROAD app are documented in the script below. During initial setup or if you have a new machine, run this script:

```
# either run as root or use sudo
./etc/DependencyInstaller.sh
```

KLayout

OpenROAD Flow requires KLayout `v0.27.1`.

Flow Scripts

- `libXScrnSaver`
- `libXft`
- `libffi-devel`
- `python3`
- `python3-pip`
  - Use pip to install pandas: `pip3 install --user pandas`
- `qt5-qtbase`
- `tcl`
- `time`
- `which`

Clone and Build

```
git clone --recursive https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts
cd OpenROAD-flow-scripts
./build_openroad.sh --local
```

Verify Installation

The binaries should be available on your `$PATH` after setting up the environment.

```
source ./setup_env.sh
yosys -help
openroad -help
exit
```

5.5.8 User Guide

The OpenROAD Project uses three tools to perform automated RTL-to-GDS layout generation:

1. yosys: Logic Synthesis
2. OpenROAD App: Floorplanning through Detailed Routing
3. KLayout: GDS merge, DRC and LVS (public PDKs)

To automate RTL-to-GDS we provide OpenROAD Flow, which contains scripts that integrate the three tools.

Code Organization

The OpenROAD Flow repository serves as an example RTL-to-GDS flow using the OpenROAD tools. The script `build_openroad.sh` in the repository will automatically build the OpenROAD toolchain.

The two main directories are:

1. `tools/`: contains the source code for the entire yosys and OpenROAD App (both via submodules) as well as other tools required for the flow.
2. `flow/`: contains reference recipes and scripts to run designs through the flow. It also contains public platforms and test designs.

Setup

See *Getting Started* guide.

Using the OpenROAD Flow

See the flow README for details about the flow and how to run designs through the flow.

Using the OpenROAD App

See the app README for details about the app and the available features and commands.

5.5.9 Metrics

The OpenROAD-flow-scripts repository contains source files (e.g., LEF/DEF, Verilog, SDC, Liberty, RC extraction) and configuration files (e.g. `config.mk`) that enable the user to run a small set of example designs through our complete RTL-to-GDS flow.

To keep track of the quality of the results, we maintain inside each design folder two files:

1. `metadata-base-ok.json` which contains all the relevant information extracted from the "golden" execution of the flow (i.e., last known good result).
2. `rules.json` which holds a set of rules that we use to evaluate new executions when a change is made.

Checking against golden

The evaluation checks for key metrics (e.g., worst slack, number of DRCs) to ensure that changes do not degrade too much with respect to the "golden" values.

After you make a significant change, e.g., fixing a bug in a piece of code, or changing some configuration variable (`PLACE_DENSITY`), you should review the results and compare them with the "golden". To perform the check, you will need to run the following command:

```
cd OpenROAD-flow-scripts/flow
# the clean_metadata is only required if you need to re-run
make [clean_metadata] metadata
```

If the command above yields any error message, please review to make sure the change in metrics is expected and justifiable. If so, proceed to the next section to update the "golden" reference.

Update process

Update of the reference files is mandatory if any metrics became worse than the values limited by the `rules.json` (see previous section on how to perform the check). Also, it is a good idea to update the "golden" files if your changes have improved a metric, to ensure that the improvement will not be lost in the future.

To update all the reference files:

```
cd OpenROAD-flow-scripts/flow
make update_ok
```

In case you have a special reason to only update one of the files, you can do so with the following commands:

```
# update metadata-base-ok.json file for the design
make update_metadata

# update rules.json file for the design
# this will use the current (+ a padding) metadata-base-ok.json
# the padding ensures that small changes do not break the flow
make update_rules
```

5.6 FAQs

If you cannot find your question/answer here, please file a GitHub issue to the appropriate repository or start a discussion.

- Issues and bugs:
  - OpenROAD: https://github.com/The-OpenROAD-Project/OpenROAD/issues
  - OpenROAD Flow: https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/issues
- Discussions:
  - OpenROAD: https://github.com/The-OpenROAD-Project/OpenROAD/discussions
  - OpenROAD Flow: https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/discussions

5.6.1 How can I contribute?

Thank you for your willingness to contribute. Please see the *Getting Involved* guide.

5.6.2 How do I update the design reference files?

See how to update the Metrics *here*.