OpenROAD

OpenROAD Team

Apr 02, 2020

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 layout generation across die, package and board, with initial focus 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 eventual zero loss of power-performance-area (PPA) design quality. No humans means that tools must adapt and self-tune, and never get stuck: thus, machine intelligence must replace today's human intelligence within the layout generation process. 24 hours means that problems must be aggressively decomposed into bite-sized subproblems for the design process to remain within the schedule constraint. Eventual zero loss of PPA quality requires parallel and distributed search to recoup the solution quality lost by problem decomposition.

For a technical description of the OpenROAD flow, please refer to our DAC paper: Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. Also, available from ACM Digital Library (doi:10.1145/3316781.3326334)

CHAPTER

ONE

HOW TO NAVIGATE THIS DOCUMENTATION

- If you are a **user**, start with the Getting Started guide, and then move on to the User Guide.
- 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.

See FAQs and Capabilities/Limitations for relevant background on the project.

CHAPTER

TWO

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: https://github.com/The-OpenROAD-Project/OpenROAD/issues
- Gitter Community: https://gitter.im/The-OpenROAD-Project/community
- Inquiries: openroad@eng.ucsd.edu

2.1 Getting Started

2.1.1 Get the tools

There are currently two options to get OpenROAD tools.

Option 1: download pre-build binaries

We currently support pre-built binaries on CentOS 7. Please, refer to the releases page on GitHub.

Option 2: build from sources

OpenROAD is divided into a number of tools that are orchestrated together to achieve RTL-to-GDS. As of the current implementation, the flow is divided into three stages:

1. Logic Synthesis: is performed by yosys.
2. Floorplanning through Global Routing: are performed by OpenROAD App.
3. Detailed Routing: is performed by TritonRoute.

In order to integrate the flow steps, OpenROAD-flow repository includes the necessary scripts to build and test the flow.

Prerequisites

Build dependencies are documented in the Dockerfile of each tool.

1. See yosys Dockerfile
2. See OpenROAD App Dockerfile
3. See TritonRoute Dockerfile

Before proceeding to the next step:

1. [recommended] Install Docker on your machine, OR
2. [bare-metal] Make sure that build dependencies for all the tools are installed on your machine.

Clone and build

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

The build script will automatically use Docker builds if it finds `docker` command installed on the system.

Verify Installation

Setup environment:

1. `./setup_env.sh` if the build was done using Docker.
2. `./setup_env_bare_metal_build.sh` if the build was done on the bare-metal.

Verify

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

- `yosys -h`
- `openroad -h`
- `TritonRoute -h`

2.1.2 Setting up the Flow

1. Clone the repository

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

1. Setup your shell environment. The `openroad` app must be setup to implement designs or run tests. See setup instructions in the repository *Verify Installation* section above.

2.1.3 Designs

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, e.g. `make DESIGN_CONFIG=./designs/nangate45/swerv.mk` or `export DESIGN_CONFIG=./designs/nangate45/swerv.mk; make`

By default, the simple design gcd is selected. 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.

:warning: Please refer to the known issues and limitations document for information on conditioning your design/files for the flow. We are working to reduce the issues and limitations, but it will take time.

2.1.4 Platforms

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

- Nangate45 / FreePDK45

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 also supports the following commercial platforms:

- TSMC65LP
- GF14 (in progress)

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 setup. 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.

2.1.5 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`

2.1.6 Miscellaneous

tiny-tests - easy to add, single concern, single Verilog file

The tiny-tests are have been designed with two design goals in mind:

1. It should be trivial to add a new test: simply add a tiny standalone Verilog file to `OpenROAD-flow/flow/designs/src/tiny-tests`
2. Each test should be as small and as standalone as possible and be a single concern test.

To run a test:

```
make DESIGN_NAME=SmallPinCount DESIGN_CONFIG=`pwd`/designs/tiny-tests.mk
```

nangate45 smoke-test harness for top level Verilog designs

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

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

TIP! Start with a small tiny submodule in your design with few pins

2.2 User Guide

OpenROAD is divided into a number of tools that are orchestrated together to achieve RTL-to-GDS. As of the current implementation, the flow is divided into three stages:

1. Logic Synthesis: is performed by yosys.
2. Floorplanning through Global Routing: are performed by OpenROAD App.
3. Detailed Routing: is performed by TritonRoute.

To Run OpenROAD flow, we provide scripts to automate the RTL-to-GDS stages. Alternatively, you can run the individual steps manually.

2.2.1 [OPTION 1] RTL-to-GDS Flow

GitHub: https://github.com/The-OpenROAD-Project/OpenROAD-flow

Code Organization

This repository serves as an example RTL-to-GDS flow using the OpenROAD tools.

The two main components are:

1. **tools**: This directory contains the source code for the entire `openroad` app (via submodules) as well as other tools required for the flow. The script `build_openroad.sh` in this repository will automatically build the OpenROAD toolchain.
2. **flow**: This directory contains reference recipes and scripts to run | designs through the flow. It also contains platforms and test designs.

Setup

The flow has the following dependencies:

- OpenROAD
- KLayout
- TritonRoute
- Yosys

The dependencies can either be obtained from a pre-compiled build export or built manually. See the KLayout website for installation instructions.

Option 1: Installing build exports**

1. Clone the OpenROAD-flow repository

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

2. Navigate to the "Releases" tab and download the latest release
3. Extract the tar to `OpenROAD-flow/tools/OpenROAD`
4. Update your shell environment

```
source setup_env.sh
```

Option 2: Building the tools using docker**

This build option leverages a multi-step docker flow to install the tools and dependencies to a runner image. To follow these instructions, you must have docker installed, permissions to run docker, and docker container network access enabled. This step will create a runner image tagged as `openroad/flow`.

1. Clone the OpenROAD-flow repository

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

2. Ensure your docker daemon is running and `docker` is in your PATH, then run the docker build.

```
./build_openroad.sh
```

3. Start an interactive shell in a docker container using your user credentials

```
docker run -u $(id -u ${USER}):$(id -g ${USER}) openroad/flow bash
```

Option 3: Building the tools locally**

1. Reference the Dockerfiles and READMEs for the separate tools on the build steps and dependencies.

```
OpenROAD-flow/tools/OpenROAD/Dockerfile
OpenROAD-flow/tools/yosys/Dockerfile
OpenROAD-flow/tools/TritonRoute/Dockerfile
```

See the KLayout instructions for installing KLayout from source.

1. Run the build script

```
./build_openroad.sh
```

2. Update your shell environment

```
source setup_env.sh
```

`klayout` must be added to the path manually.

Using the flow

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

2.2.2 [OPTION 2] Individual Flow Steps

Logic Synthesis

GitHub: https://github.com/The-OpenROAD-Project/yosys

Setup

Requirements

- C++ compiler with C++11 support (up-to-date CLANG or GCC is recommended)
- GNU Flex, GNU Bison, and GNU Make.
- TCL, readline and libffi.

On Ubuntu:

```
$ sudo apt-get install build-essential clang bison flex \
        libreadline-dev gawk tcl-dev libffi-dev git \
        graphviz xdot pkg-config python3 libboost-system-dev \
        libboost-python-dev libboost-filesystem-dev zlib1g-dev
```

On Mac OS X Homebrew can be used to install dependencies (from within cloned yosys repository):

```
$ brew tap Homebrew/bundle && brew bundle
```

To configure the build system to use a specific compiler, use one of

```
$ make config-clang
$ make config-gcc
```

Build

To build Yosys simply type 'make' in this directory.

```
$ make
$ sudo make install
```

Synthesis Script

```
yosys -import

if {[info exist ::env(DC_NETLIST)]} {
exec cp $::env(DC_NETLIST) $::env(RESULTS_DIR)/1_1_yosys.v
exit
}

# Don't change these unless you know what you are doing
set stat_ext    "_stat.rep"
set gl_ext      "_gl.v"
set abc_script  "+read_constr,$::env(SDC_FILE);strash;ifraig;retime,-D,{D},-M,6;
↪strash;dch,-f;map,-p-M,1,{D},-f;topo;dnsize;buffer,-p;upsize;"
```

(continues on next page)

(continued from previous page)

```
# Setup verilog include directories
set vIdirsArgs ""
if {[info exist ::env(VERILOG_INCLUDE_DIRS)]} {
    foreach dir $::env(VERILOG_INCLUDE_DIRS) {
        lappend vIdirsArgs "-I$dir"
    }
    set vIdirsArgs [join $vIdirsArgs]
}


# read verilog files
foreach file $::env(VERILOG_FILES) {
    read_verilog -sv {*}$vIdirsArgs $file
}


# Read blackbox stubs of standard cells. This allows for standard cell (or
# structural netlist) support in the input verilog
read_verilog $::env(BLACKBOX_V_FILE)

# Apply toplevel parameters (if exist)
if {[info exist ::env(VERILOG_TOP_PARAMS)]} {
    dict for {key value} $::env(VERILOG_TOP_PARAMS) {
        chparam -set $key $value $::env(DESIGN_NAME)
    }
}

# Read platform specific mapfile for OPENROAD_CLKGATE cells
if {[info exist ::env(CLKGATE_MAP_FILE)]} {
    read_verilog $::env(CLKGATE_MAP_FILE)
}

# Use hierarchy to automatically generate blackboxes for known memory macro.
# Pins are enumerated for proper mapping
if {[info exist ::env(BLACKBOX_MAP_TCL)]} {
    source $::env(BLACKBOX_MAP_TCL)
}

# generic synthesis
synth  -top $::env(DESIGN_NAME) -flatten

# Optimize the design
opt -purge

# technology mapping of latches
if {[info exist ::env(LATCH_MAP_FILE)]} {
    techmap -map $::env(LATCH_MAP_FILE)
}

# technology mapping of flip-flops
dfflibmap -liberty $::env(OBJECTS_DIR)/merged.lib
opt

# Technology mapping for cells
abc -D [expr $::env(CLOCK_PERIOD) * 1000] \
    -constr "$::env(SDC_FILE)" \
```

(continues on next page)

(continued from previous page)

```
    -liberty $::env(OBJECTS_DIR)/merged.lib \
    -script $abc_script \
    -showtmp

# technology mapping of constant hi- and/or lo-drivers
hilomap -singleton \
        -hicell {*}$::env(TIEHI_CELL_AND_PORT) \
        -locell {*}$::env(TIELO_CELL_AND_PORT)

# replace undef values with defined constants
setundef -zero

# Splitting nets resolves unwanted compound assign statements in netlist (assign {..}
↪= {..})
splitnets

# insert buffer cells for pass through wires
insbuf -buf {*}$::env(MIN_BUF_CELL_AND_PORTS)

# remove unused cells and wires
opt_clean -purge

# reports
tee -o $::env(REPORTS_DIR)/synth_check.txt check
tee -o $::env(REPORTS_DIR)/synth_stat.txt stat -liberty $::env(OBJECTS_DIR)/merged.lib

# write synthesized design
write_verilog -noattr -noexpr -nohex -nodec $::env(RESULTS_DIR)/1_1_yosys.v
```

Initialize Floorplan

```
initialize_floorplan
[-site site_name]           LEF site name for ROWS
[-tracks tracks_file]       routing track specification
-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]         space around core, default 0.0 (microns)
```

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

If no -tracks file is used the routing layers from the LEF are used.

```
core_area = design_area / (utilization / 100)
core_width = sqrt(core_area / aspect_ratio)
core_height = core_width * aspect_ratio
core = ( core_space, core_space ) ( core_space + core_width, core_space + core_height
↪)
die = ( 0, 0 ) ( core_width + core_space * 2, core_height + core_space * 2 )
```

Place pins around core boundary.

```
auto_place_pins pin_layer
```

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.

```
set_wire_rc [-layer layer_name]
            [-resistance res ]
        [-capacitance cap]
        [-corner corner_name]
```

The `set_wire_rc` command sets the resistance and capacitance used to estimate delay of routing wires. Use `-layer` or `-resistance` and `-capacitance`. If `-layer` is used, the LEF technology resistance and area/edge capacitance values for the layer are used. The units for `-resistance` and `-capacitance` are from the first liberty file read, resistance_unit/distance_unit and liberty capacitance_unit/distance_unit. RC parasitics are added based on placed component pin locations. If there are no component locations no parasitics are added. The resistance and capacitance are per distance unit of a routing wire. Use the `set_units` command to check units or `set_cmd_units` to change units. They should represent "average" routing layer resistance and capacitance. If the set_wire_rc command is not called before resizing, the default_wireload model specified in the first liberty file or with the SDC set_wire_load command is used to make parasitics.

```
buffer_ports [-inputs]
        [-outputs]
        -buffer_cell buffer_cell
```

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. If The default behavior is `-inputs` and `-outputs` if neither is specified.

```
resize [-libraries resize_libraries]
    [-dont_use cells]
    [-max_utilization util]
```

The `resize` command resizes gates to normalize slews.

The `-libraries` option specifies which libraries to use when resizing. `resize_libraries` defaults to all of the liberty libraries that have been read. Some designs have multiple libraries with different transistor thresholds (Vt) and are used to trade off power and speed. Chosing a low Vt library uses more power but results in a faster design after the resizing step. Use the `-dont_use` option to specify a list of patterns of cells to not use. For example, `*/DLY*` says do not use cells with names that begin with `DLY` in all libraries.

```
repair_max_cap -buffer_cell buffer_cell
            [-max_utilization util]
repair_max_slew -buffer_cell buffer_cell
                [-max_utilization util]
```

The `repair_max_cap` and `repair_max_slew` commands repair nets with maximum capacitance or slew violations by inserting buffers in the net.

```
repair_max_fanout -max_fanout fanout
                -buffer_cell buffer_cell
                [-max_utilization util]
```

The `repair_max_fanout` command repairs nets with a fanout greater than `fanout` by inserting buffers between the driver and the loads. Buffers are located at the center of each group of loads.

```
repair_tie_fanout [-max_fanout fanout]
                  [-verbose]
                  lib_port
```

The `repair_tie_fanout` command repairs tie high/low nets with fanout greater than `fanout` by cloning the tie high/low driver. `lib_port` is the tie high/low port, which can be a library/cell/port name or object returned by `get_lib_pins`. Clones are located at the center of each group of loads.

```
repair_hold_violations -buffer_cell buffer_cell
                       [-max_utilization util]
```

The `repair_hold_violations` command inserts buffers to repair hold check violations.

```
report_design_area
```

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

```
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.

A typical resizer command file is shown below.

```
read_lef nlc18.lef
read_liberty nlc18.lib
read_def mea.def
read_sdc mea.sdc
set_wire_rc -layer metal2
set buffer_cell [get_lib_cell nlc18_worst/snl_bufx4]
set max_util 90
buffer_ports -buffer_cell $buffer_cell
resize -resize
repair_max_cap -buffer_cell $buffer_cell -max_utilization $max_util
repair_max_slew -buffer_cell $buffer_cell -max_utilization $max_util
# repair tie hi/low before max fanout so they don't get buffered
repair_tie_fanout -max_fanout 100 Nangate/LOGIC1_X1/Z
repair_max_fanout -max_fanout 100 -buffer_cell $buffer_cell -max_utilization $max_util
repair_hold_violations -buffer_cell $buffer_cell -max_utilization $max_util
```

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

resize

report_checks
report_tns
report_wns
```

Timing Analysis

Timing analysis commands are documented in src/OpenSTA/doc/OpenSTA.pdf.

After the database has been read from LEF/DEF, Verilog or an OpenDB database, use the `read_liberty` command to read Liberty library files used by the design.

The example script below timing analyzes a database.

```
read_liberty liberty1.lib
read_db reg1.db
create_clock -name clk -period 10 {clk1 clk2 clk3}
set_input_delay -clock clk 0 {in1 in2}
set_output_delay -clock clk 0 out
report_checks
```

MacroPlace

TritonMacroPlace https://github.com/The-OpenROAD-Project/TritonMacroPlace

```
macro_placement -global_config <global_config_file>
```

- **global_config** : Set global config file loction. [string]

Global Config Example

```
set ::HALO_WIDTH_V 1
set ::HALO_WIDTH_H 1
set ::CHANNEL_WIDTH_V 0
set ::CHANNEL_WIDTH_H 0
```

- **HALO_WIDTH_V** : Set macro's vertical halo. [float; unit: micron]
- **HALO_WIDTH_H** : Set macro's horizontal halo. [float; unit: micron]
- **CHANNEL_WIDTH_V** : Set macro's vertical channel width. [float; unit: micron]
- **CHANNEL_WIDTH_H** : Set macro's horizontal channel width. [float; unit: micron]

Tapcell

Tapcell and endcap insertion.

```
tapcell -tapcell_master <tapcell_master>
        -endcap_master <endcap_master>
        -endcap_cpp <endcap_cpp>
        -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>
```

(continues on next page)

(continued from previous page)

```
        -cnrcap_nwin_master <cnrcap_nwin_master>
        -cnrcap_nwout_master <cnrcap_nwout_master>
        -incnrcap_nwin_master <incnrcap_nwin_master>
        -incnrcap_nwout_master <incnrcap_nwout_master>
        -tbtie_cpp <tbtie_cpp>
        -no_cell_at_top_bottom
        -add_boundary_cell
```

You can find script examples for both 45nm/65nm and 14nm in `tapcell/etc/scripts`

Global Placement

RePlAce global placement. https://github.com/The-OpenROAD-Project/RePlAce

```
global_placement -skip_initial_place
                 -incremental
                 -bin_grid_count <grid_count>
                 -density <density>
                 -init_density_penalty <init_density_penalty>
                 -init_wirelength_coef <init_wirelength_coef>
                 -min_phi_coef <min_phi_coef>
                 -max_phi_coef <max_phi_coef>
                 -overflow <overflow>
                 -initial_place_max_iter <max_iter>
                 -initial_place_max_fanout <max_fanout>
                 -verbose_level <level>
```

Flow Control

- **skip_initial_place** : Skip the initial placement (BiCGSTAB solving) before Nesterov placement. IP improves HPWL by ~5% on large designs. Equal to '-initial_place_max_iter 0'
- **incremental** : Enable the incremental global placement. Users would need to tune other parameters (e.g. init_density_penalty) with pre-placed solutions.

Tuning Parameters

- **bin_grid_count** : Set bin grid's counts. Default: Defined by internal algorithm. [64,128,256,512,..., int]
- **density** : Set target density. Default: 0.70 [0-1, float]
- **init_density_penalty** : Set initial density penalty. Default: 8e-5 [1e-6 - 1e6, float]
- **init_wire_length__coef** : Set initial wirelength coefficient. Default: 0.25 [unlimited, float]
- **min_phi_coef** : Set pcof_min(μ_k Lower Bound). Default: 0.95 [0.95-1.05, float]
- **max_phi_coef** : Set pcof_max(μ_k Upper Bound). Default: 1.05 [1.00-1.20, float]
- **overflow** : Set target overflow for termination condition. Default: 0.1 [0-1, float]
- **initial_place_max_iter** : Set maximum iterations in initial place. Default: 20 [0-, int]
- **initial_place_max_fanout** : Set net escape condition in initial place when 'fanout >= initial_place_max_fanout'. Default: 200 [1-, int]

Other Options

- **verbose_level** : Set verbose level for RePlAce. Default: 1 [0-10, int]

Detailed Placement

Legalize a design that has been globally placed.

```
legalize_placement [-constraints constraints_file]
```

Clock Tree Synthesis

Create clock tree subnets.

```
clock_tree_synthesis -lut_file <lut_file> \
                     -sol_list <sol_list_file> \
                     -wire_unit <wire_unit> \
                     -root_buf <root_buf> \
                     [-clk_nets <list_of_clk_nets>]
```

- `lut_file`, `sol_list` and `wire_unit` are parameters related to the technology characterization described here.
- `root_buffer` is the master cell of the buffer that serves as root for the clock tree.
- `clk_nets` is a string containing the names of the clock roots. If this parameter is ommitted, TritonCTS looks for the clock roots automatically.

Global Routing

FastRoute global route. Generate routing guides given a placed design.

```
fastroute -output_file out_file
        -capacity_adjustment <cap_adjust>
        -min_routing_layer <min_layer>
        -max_routing_layer <max_layer>
        -pitches_in_tile <pitches>
        -layers_adjustments <list_of_layers_to_adjust>
        -regions_adjustments <list_of_regions_to_adjust>
        -nets_alphas_priorities <list_of_alphas_per_net>
        -verbose <verbose>
        -unidirectional_routing
        -clock_net_routing
```

Options description:

- **capacity_adjustment**: Set global capacity adjustment (e.g.: -capacity_adjustment *0.3*)
- **min_routing_layer**: Set minimum routing layer (e.g.: -min_routing_layer *2*)
- **max_routing_layer**: Set maximum routing layer (e.g.: max_routing_layer *9*)
- **pitches_in_tile**: Set the number of pitches inside a GCell
- **layers_adjustments**: Set capacity adjustment to specific layers (e.g.: -layers_adjustments {{ } ... })
- **regions_adjustments**: Set capacity adjustment to specific regions (e.g.: -regions_adjustments { } ... })

- **nets_alphas_priorities**: Set alphas for specific nets when using clock net routing (e.g.: -nets_alphas_priorities {{<net_name> } . . . })
- **verbose**: Set verbose of report. 0 for less verbose, 1 for medium verbose, 2 for full verbose (e.g.: -verbose 1)
- **unidirectional_routing**: Activate unidirectional routing *(flag)*
- **clock_net_routing**: Activate clock net routing *(flag)*
- **NOTE 1:** if you use the flag *unidirectional_routing*, the minimum routing layer will be assigned as "2" automatically
- **NOTE 2:** the first routing layer of the design have index equal to 1
- **NOTE 3:** if you use the flag *clock_net_routing*, only guides for clock nets will be generated

Detailed Routing

GitHub: https://github.com/The-OpenROAD-Project/TritonRoute

Build

TritonRoute is tested in 64-bit CentOS 6/7 environments with the following prerequisites:

- A compatible C++ compiler supporting C++17 (GCC 7 and above)
- Boost >= 1.68.0
- Bison >= 3.0.4
- zlib >= 1.2.7
- CMake >= 3.1

To install TritonRoute:

```
$ git clone https://github.com/The-OpenROAD-Project/TritonRoute.git
$ cd TritonRoute
$ mkdir build
$ cd build
$ cmake -DBOOST_ROOT=<BOOST_ROOT> ../
$ make
```

Run

```
$ ./TritonRoute -lef <LEF_FILE> -def <DEF_FILE> -guide <GUIDE_FILE> -output <OUTPUT_
↪DEF>
```

2.3 Capabilities/Limitations

2.3.1 Global Considerations

- OpenROAD v1.0 production will be focused on the tapeout mentioned in the above introduction. Features will be implemented in priority order based on our sponsor requirement to make the chosen design manufacturable. In Phase 2 of the IDEA program, the OpenROAD tool feature set will be rounded out and more of the project's flow and tool research objectives will be addressed.

- Each new design enablement (foundry process/PDK, library, IPs) will require setup via configuration files, one-time characterizations, etc. as documented with the tool. Examples include (i) the setup of PDN generation, (ii) the creation of "wrapped LEF abstracts" for cells and/or macros to comply with Generic Node Enablement (see Routing, below), and (iii) the creation of characterized lookup tables to guide CTS buffering.

2.3.2 Supported Platforms

- OpenROAD v1.0 will build on "bare metal", CentOS 7 with required packages installed as specified in the README.
- MacOS will also be supported.
- Users with access to Docker will also be able to build on any machine using the included Dockerfile.

2.3.3 Design Partitioning and Logic Synthesis

- Logic Synthesis (Yosys) will accept only hierarchical RTL Verilog.
- SystemVerilog to Verilog conversion must be performed by the user (e.g., using bsg sv2v or any tool of their choosing) before running Yosys.
- Logic Synthesis is one of potentially multiple steps in OpenROAD that may require a single merged LEF as of the v1.0 release. A utility script to perform merging is here.
- To support convergence in the downstream place-CTS-route steps, it is advisable to exclude cells that risk difficult pin access (e.g., sub-X1 sizes) and/or to invoke cell padding during placement. The cell exclusion would be akin to a "dont_use" list, which is not currently supported and must be manually implemented by editing the library files.

2.3.4 STA

- Supports multi-corner analysis (e.g., setup and hold), but with limit of one mode.
- SDC support up to latest public, open version (e.g., SDC 1.4).
- No SI analysis: any coupling caps can be multiplied by a "Miller Coupling Factor" (MCF) and then treated as grounded.
- No CCS/ECSM (current-source model) support.
- No LVF support.
- No PBA analysis option.
- No instance IR drop (i.e., setting a rail voltage for given instance).
- No reduction of non-tree wiring topologies. (Arnoldi reduction provided along with O'Brien-Savarino, 2-pole, Elmore reduction and delay calculation options.)

2.3.5 Floorplan

- Macro placement is limited to 100 RAMs/macros per P&R block.
- PDN configuration files must be provided by the user. These are documented in the "pdngen" tool repo, here.

2.3.6 Placement

- A P&R block is limited to one logic power domain and one I/O power domain. Additional power domains must be handled manually (OpenROAD Tcl scripting).
- Isolation cells, level converters and power management must be manually inserted into the layout by the user (e.g., as pre-placements).
- No support of UPF/CPF formats for power intent.
- Support of user guidance for logic clustering and placement will be limited to "fence" and "pre-placement" guidance, with the caveat that such guidance may degrade solution QOR in the OpenROAD flow.

2.3.7 Clock Tree Synthesis

- Support only positive edge-triggered FFs
- Hold buffering will be at post-CTS and not later in the flow

2.3.8 Routing

- The TritonRoute router will not understand LEF57, LEF58 constructs in techlef: the workaround is OpenROAD Generic Node Enablement (see "OpenROAD Requirements for Generic Node Enablement, at this link).
- Users should be advised that TritonRoute does not handle coloring explicitly; a color-correct-by-construction methodology (e.g., for Mx layers in 14/12nm) is achieved via Generic Node Enablement.
- Antenna checking and fixing capability is committed for v1.0.

2.3.9 Layout Finishing and Final Verifications

- Parasitic extraction (SPEF from layout) is unlikely to comprehend coupling.
- There is no "signoff-quality electrical/performance analysis" counterpart to "PrimeTime-SI" (timing, signal integrity) or "Voltus"/"RedHawk" (power integrity).
- A golden PV tool will be the evaluator for DRC.
- Generation of merged GDS currently requires a Magic 8.2 tech file. Details are given here.
- Export of merged GDS does not add text markings that may be expected by commercial physical verification tools.
- For supported design tape-outs (particularly, at a commercial 14/12nm node, up through July 2020), physical verification (DRC/LVS) is expected to be performed by the design team using commercial tools. (Everything up to routed DEF and merged GDS will be produced by OpenROAD or other open-source tools.)

2.4 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
- Improving documentation including this document
- Star our project and repos so we can see the number of people interested

2.4.1 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 in the spirit of OpenROAD's permissively 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.

2.4.2 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. 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.

2.4.3 Contributing Scripts and Code

We follow the Google C++ style guide. If you find code that is not following this guide, within each file that you edit, follow the style in that file. Please pay careful attention to the 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 code architecture document to work with the app and its submodule repos in an efficient way.

Please pay attention to the test directory and be sure to add tests for any code changes that you make with open sourceable PDK and design information. We provide the nandgate45 PDK in the OpenROAD-flow repo to help with this. Pull requests with code changes are unlikely to be accepted without accompanying test cases. There are many example 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.

If you want to add a new tool please look in the src/tool directory of the add_tool branch for an example of how to add one.

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 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 sub components, 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 code itself.

2.4.4 Questions

You can file git issues to ask questions, file issues or you can contact us via email openroad at eng.ucsd.edu

2.5 Developer Guide

OpenROAD is a tool to build a chip from a synthesized netlist to a physical design for manufacturing.

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.

2.5.1 Tool File Organization

Every tool follows the following file structure.

```
CMakelists.txt - add_subdirectory's src/CMakelists.txt
src/ - sources and private headers
src/CMakelists.txt
include/<toolname>/ - exported headers
test/
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
src/OpenROAD.hh - OpenROAD top level class, has instances of tools
```

Submodule repos in /src (note these are NOT in src/module)

```
OpenDB
OpenSTA
replace
ioPlacer
FastRoute
TritonMacroPlace
OpenRCX
flute3
eigen
```

Submodules that are shared by multiple tools are owned by OpenROAD so that there are not redundant source trees and compiles.

Each tool submodule 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.

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

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) 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. Eventually all of the tools should be upgraded to remove this deficiency as they are re-written to work in the OpenROAD framework.

Each tool should use a unique namespace for all of its code. The same namespace should be used for any Tcl commands. Internal Tcl commands stay inside the namespace, user visible Tcl commands will be exported to the global namespace. User commands should be simple Tcl commands such as 'global_place_design' that do not create tool instances that must be based to the commands. Defining Tcl commands for a tool class is fine for internals, 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.

2.5.2 Initialization (c++ tools only)

The OpenRoad class only has pointers to each tools with functions to get each tool. Each tool has (at a minimum) a function to make an instance of the tool class, and an initialization function that is called after all of the tools have been made, and a funtion 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 dependences. Each init function grabs the state it needs out of the OpenRoad instance.

2.5.3 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 arguements (sta::parse_keyword_args). See OpenSTA/tcl/*.tcl for examples. Use swig to define internal functions to C++ functionality.p

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

2.5.4 Test

Each "tool" has a /test directory containing a script nameed "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 the current working directory.

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

2.5.5 Builds

Checking out the OpenROAD repo with –recursive installs all of the OpenRoad tools and their submodules.

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

All tools build using cmake and must have a CMakeLists.txt file in their tool directory.

This builds the 'openroad' executable in /build.

Note that removing submodules from a repo when moving it into OpenROAD is less than obvious. Here are the steps:

```
git submodule deinit <path_to_submodule>
git rm <path_to_submodule>
git commit-m "Removed submodule "
rm -rf .git/modules/<path_to_submodule>
```

2.5.6 Tool Work Flow

To work on one of the tools inside OpenROAD when it is a submodule requires updating the OpenROAD repo to integrate your changes. Submodules point to a specific version (hash) of the submodule repo and do not automatically track changes to the submodule repo.

Work on OpenROAD should be done in the `openroad` branch.

To make changes to a submodule, first check out a branch of the submodule (git clone –recursive does not check out a branch, just a specific commit).

```
cd src/<tool>
git checkout <branch>
```

`<branch>` is the branch used for development of the tool when it is inside OpenROAD. The convention is for to be named 'openroad'.

After making changes inside the tool source tree, stage and commit them to the tool repo and push them to the remote repo.

```
git add ...
git commit -m "massive improvement"
git push
```

If instead you have done development in a different branch or source tree, merge those changes into the branch used for OpenROAD.

Once the changes are in the OpenROAD submodule source tree it will show them as a diff in the hash for the directory.

```
cd openroad
git stage <tool_submodule_dir>
git commit -m "merge tool massive improvement"
git push
```

2.5.7 Example of Adding a Tool to OpenRoad

The branch "add_tool" illustrates how to add a tool to OpenRoad. Use `git checkout add_tool` to checkout the branch. To see the changes between OpenRoad with and without Tool use `git diff master`.

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

```
% toolize foo
Helping 23/6
Gotta pos_arg1 foo
Gotta param1 0.000000
Gotta flag1 false

% toolize -flag1 -key1 2.0 bar
Helping 23/6
Gotta pos_arg1 bar
Gotta param1 2.000000
Gotta flag1 true

% help toolize
toolize [-key1 key1] [-flag1] pos_arg1
```

2.5.8 Documentation

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

2.5.9 Tool Flow

1. Verilog to DB (dbSTA)
2. Init Floorplan (OpenROAD)
3. I/O placement (ioPlacer)
4. PDN generation (pdngen
5. Tapcell and Welltie insertion (tapcell with LEF/DEF)
6. I/O placement (ioPlacer)

7. Global placement (RePlAce)
8. Gate Resizing and buffering (Resizer)
9. Detailed placement (OpenDP)
10. Clock Tree Synthesis (TritonCTS)
11. Repair Hold Violations (Resizer)
12. Global route (FastRoute)
13. Detailed route (TritonRoute)n
14. Final timing/power report (OpenSTA)

2.5.10 Tool Checklist

- OpenROAD submodules reference tool `openroad` branch head
- No `develop`, `openroad_app`, `openroad_build` branches.
- CMakeLists.txt does not use glob. https://gist.github.com/mbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1
- No main.cpp or main procedure.
- No compiler warnings for gcc, clang with optimization enabled.
- Does not call flute::readLUT (called once by OpenRoad).
- Tcl command(s) documented in top level README.md in flow order.
- Command line tool documentation in tool README.
- Conforms to Tcl command naming standards (no camel case).
- Does not read configuration files.
- Use command arguments or support commands.
- `.clang-format` at tool root directory to aid foreign programmers.
- No jenkins/, Jenkinsfile, Dockerfile in tool directory.
- regression script named "test/regression" with default argument that runs tests. Not tests/regression-tcl.sh, not test/run_tests.py etc.
- Regression runs independent of current directory.
- Regression only prints test results or summary, does not belch 1000s of lines of output.
- Test scripts use OpenROAD tcl commands (not itcl, not internal accessors).
- Regressions report no memory errors with valgrind.
- Regressions report no memory leaks with valgrind (difficult).

James Cherry, Dec 2019

2.6 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, or 2000 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 is 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", it's 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 a `int64_t`. Doing an area calculation on a large chip that is more than `sqrt(5e+15) = 7e+7 DBU` will overflow the mantissa and truncate the result.

Not only is a `double` less capable than an `int64_t`, using it the tells any reader of the code that the value can be 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 like 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

2.7 FAQs

2.7.1 Where can I download OpenROAD tools?

Currently, we don't provide pre-built binaries for the tools. You need to build the tools yourself on a supported platform. Please, refer to the Getting Started section to build the tools.

2.7.2 How can I contribute?

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