

OpenFPGA Independent Study

Jackson Hafele

December 2023

Contents

1	Introduction	2
1.1	OpenFPGA Motivation	2
2	Tool Overview	4
2.1	Supported Tools	4
2.2	Fabric Netlists	5
2.3	Testbenches	7
2.4	Supplemental Resources	9
3	Tool Installation	10
3.1	Operating Systems	10
3.2	Docker Install	10
3.3	Dependencies (Without Docker)	11
3.4	Build Steps	13
4	Running Tools	14
4.1	Folder Structure	14
4.2	Shell Commands	14
4.3	OpenFPGA Tasks	15
4.4	OpenFPGA Flow	17
4.5	Fabric Netlist Generation	17
4.6	Simulation	18
4.7	Synthesis	22
5	Architecture Modeling	23
5.1	Custom Verilog Modules	23
5.2	Custom Cell Library	26
6	Conclusion	30
6.1	Generated Results	30
6.2	Future Work	31

1 Introduction

This documentation was generated for a 1 credit CPRE 595 Independent Study. This document acts as a getting started guide using the open-source tool OpenFPGA, which can generate FPGA fabric netlists, automated simulation verifications, SPICE results, and custom configurations for functional Verilog and cell libraries.

This document overviews:

- Tools used in OpenFPGA
- Overview of OpenFPGA structure
- How to install OpenFPGA locally and with Docker
- How to generate a FPGA fabric netlist
- How to verify a FPGA fabric netlist with multiple benchmarks
- How to add custom Verilog modules to FPGA architecture
- How to add custom cell library to FPGA architecture

1.1 OpenFPGA Motivation

According to the authors of OpenFPGA [1], many FPGA products require specific hardware integration such as large memory blocks or custom optimizations, which can lead to longer development times. Similar to standard ASIC design, there is a large barrier of entry in expertise and time for hardware design and tooling, which can take years to learn. From having to handle manual FPGA layouts or EDA tools for bitstream generation, many different aspects of the FPGA design cycle have led to slowing work. Due to this, many designs have taken a more generalized approach, while missing out on more advanced optimizations that could be required in the future.

OpenFPGA solves this issue by providing an agile framework that can lead to fast turnaround times to build custom FPGA architectures and bitstreams to program said devices. The first design flow converts a XML FPGA architecture description to GL Verilog netlists consisting of the full FPGA design, which could then be fed into another EDA tool for place and route design. This flow can also generate automated testbenches comparing the benchmark design and another design reconfigured in the new FPGA fabric, leading to shorter verification times. The second design flow revolves around generating bitstream files for specific FPGA fabrics based on a set of functional Verilog designs, similar to standard EDA tools such as Vivado. By utilizing both of these toolflows, the authors of OpenFPGA were able to achieve layout designs within 24 hours for an architecture design similar to the Stratix IV. Compared to commercial FPGAs, OpenFPGA has generated a layout with a 60%/20% area/performance gap, with the added benefit of a faster design cycle.

OpenFPGA has included many features in one package that only few previous works have covered at once:

- Multimode logic blocks, only included in more recent works [2][3]
- Heterogenous blocks [3]
- Tile-based architecture, needed for larger scale FPGAs [4]
- Bitstream generation, missing in few works [4] [5] [6]
- Custom cell support, not supported by any previous works

2 Tool Overview

2.1 Supported Tools

The following figure shows a top level diagram of the OpenFPGA Architecture. The sections in green and purple represent custom tools made for OpenFPGA, while tools in grey represent standard open-source tools such as Yosys or simulation tools. The sections in yellow represent interchangeable files that can include timing constraints, netlists, testbenches, and synthesized designs that are dependent on the design being made.

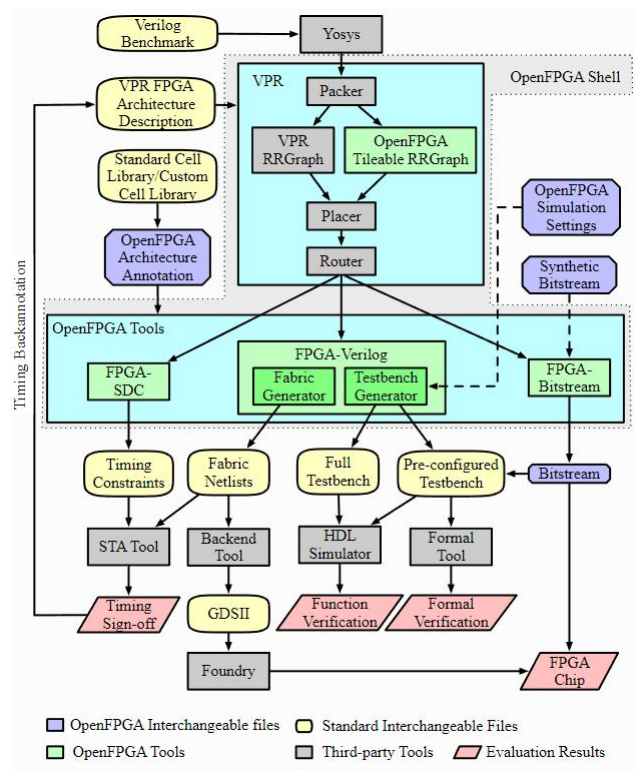


Figure 1: OpenFPGA Internal Tools[7]

The following commercial and open-source tools may be utilized [7]:

- Backend
 - Synopsys IC Compiler II (v2019.03+)
 - Cadence Innovus (v19.1+)
- Timing Analysis

- Synopsys PrimeTime (2019.03+)
- Cadence Tempus (19.15+)
- Verificaiton
 - Synopsys VCS (v2019.06+)
 - Synopsys Formality (v2019.03+)
 - Mentor ModelSim (v10.6+)
 - Mentor QuestaSim (v2019.3+)
 - Cadence NCSim (v15.2+)
 - Icarus iVerilog (v10.1+)

2.2 Fabric Netlists

FPGA Fabrics are created based on the VPR toolflow and XML architecture files which outline the entire FPGA structure. By utilizing OpenFPGA, the Verilog files will get defined with a top level FPGA module and subsequent blocks, as depicted in Figure 2.

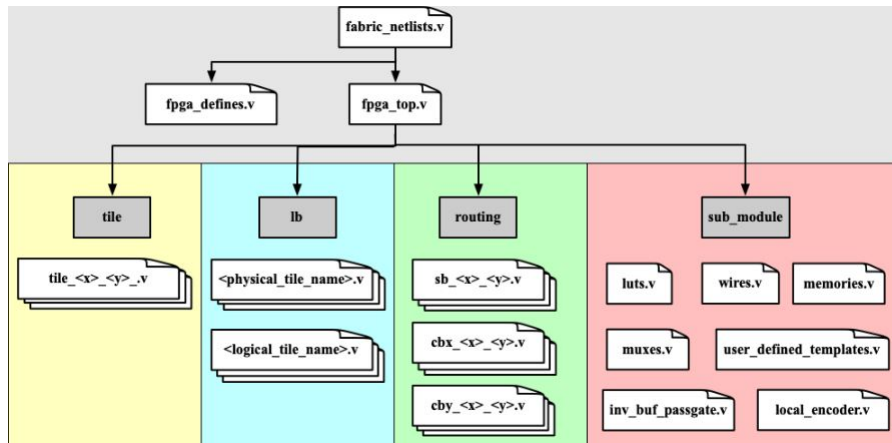


Figure 2: Fabric Netlist Verilog Heirarchy[7]

Generated Verilog Files:

- **fabric_netlists.v:** Top level verilog file which contains top level FPGA module and user-defined Verilog netlists
- **fabric_defines.v:** User-defined Verilog netlists to be referenced in fabric-netlists for verification
- **fabric_top.v:** Top-level module consisting of generated FPGA fabric; Includes tiles, logic blocks, routing blocks, and primitives

- **tile_x.y.v:** Unique netlist for each tile, consists of logic blocks and routing blocks
- **Logic Blocks:** Contains configurable logic blocks, I/O blocks, DSP modules, or Block RAM
 - **physical_tile_name.v:** Verilog netlist generated for each [physical_tile] defined in VPR architecture
 - **logical_tile_name.v:** Verilog netlist for each root pb_type in the [complexblock] for defined in VPR architecture
- **Routing Blocks:**
 - **sb_x.y.v:** Individual netlist for each unique switch block defined by VPR architecture
 - **cbx_x.y.v:** Individual netlist for each unique X-direction connection block defined by VPR architecture
 - **cby_x.y.v:** Individual netlist for each unique Y-direction connection block defined by VPR architecture
- **Primitive Modules** All defined under OpenFPGA XML architecture file
 - **luts.v:** Look-Up Tables, defined under [circuit_model name="lut"]
 - **wires.v:** Routing wires, defined under [circuit_model name="wire—chan_wire"]
 - **memories.v:** Configurable memory, defined under [circuit_model name="cfft—sram"]
 - **muxes.v:** Routing multiplexers, defined under [circuit_model name="mux"]
 - **inv_buf_passgate.v:** Inverters, defined under [circuit_model name="lut"]
 - **local_encoder.v:** Encoders and Decoders, created when routing MUX defined to use local encoders, defined under [circuit_model name="lut"]
 - **user_defined_templates.v:** Template netlist which can be used as a reference for their own user-defined Verilog modules. This file will be created when `--print_user_defined_template` is added to the `write_fabric_verilog` command.

Figure 3 below represents a sample architecture diagram of the generated Verilog netlists. It can be seen that each tile contains a logic block, X/Y-connection blocks, and a switch block. Inside each configurable logic block in the sample shows inverter buffers, multiplexers, and local encoders.

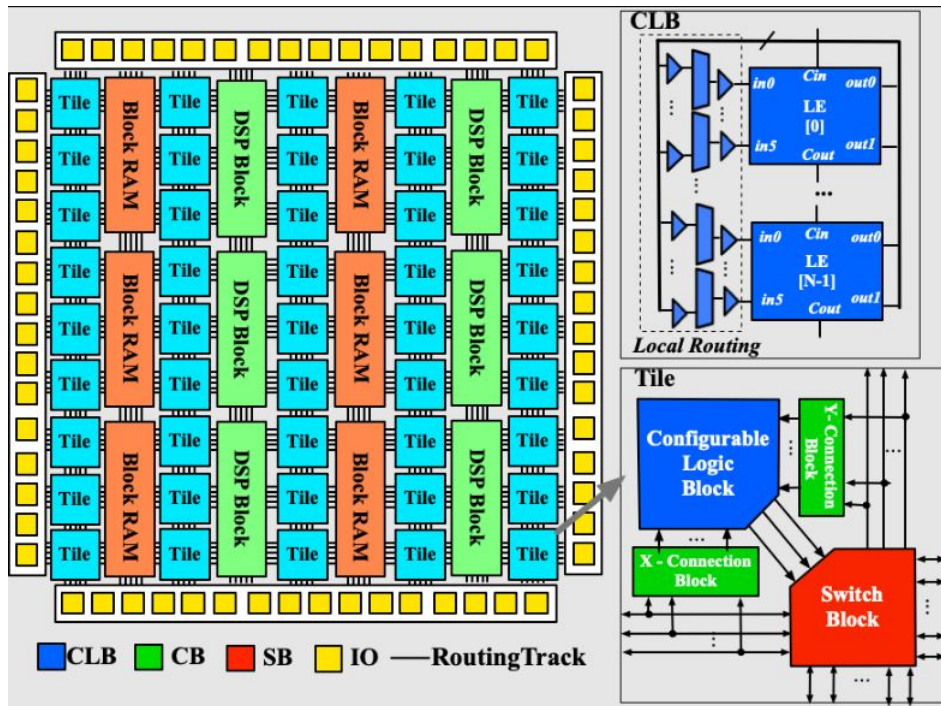


Figure 3: Fabric Netlist Generated Architecture[7]

2.3 Testbenches

To test functional benchmark designs with different FPGA fabrics, verilog testbenches can be used which generates waveforms to be viewed and cross checked. The overall hierarchy for the testflow is demonstrated in Figure 4. The same input stimulus is driven into both the original functional Verilog design that is used as a benchmark as well as the generated FPGA fabric. Then, the expected output from the functional benchmark design is cross checked with the output from the generated FPGA fabric, to ensure that there are no differences in values. If there is, then the design would not function properly on the generated FPGA netlist. A bitstream file is generated utilizing FPGA-Bitstream to program the FPGA fabric before evaluation and cross-checking.

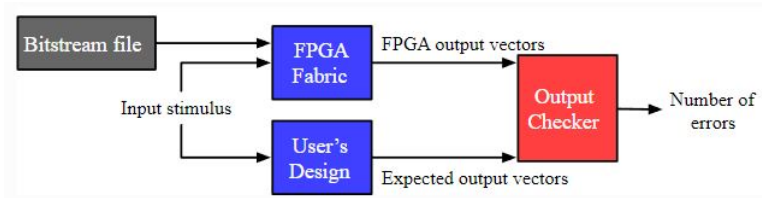


Figure 4: OpenFPGA Internal Tools[7]

Verilog testbenches are generated with the command `write_fabric_verilog` which can be run directly as part of an OpenFPGA task flow. In the designated output directory, two different testbench models are generated, as seen in Figure 5.

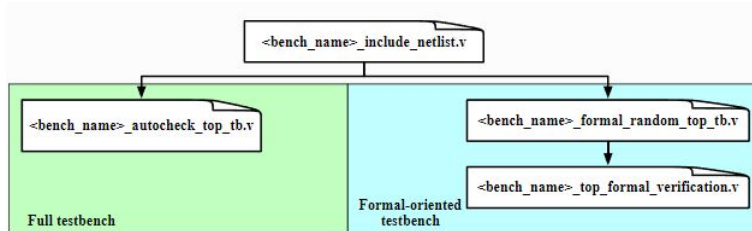


Figure 5: OpenFPGA Internal Tools[7]

The **Full Testbench** is divided into two phases, as seen in Figure 6. The goal is to simulate a FPGA’s entire operating area, by driving inputs to all possible pins on the FPGA. In the first phase, the Configuration Phase, the generated bitstream is loaded into the programmable input of the FPGA fabric, after the FPGA system is reset. After this occurs, the Operating Phase begins, where random input stimulus is generated for the FPGA fabric AND the baseline functional benchmark. If the FPGA outputs from the random vectors do not match the benchmark outputs, then an error counter will tick and the verification will fail.

The **Formal-oriented Testbench** takes less time to simulate, and will test a reconfigured FPGA with an instantiated bitstream. Since the FPGA is already instantiated with the bitstream, the configuration phase does not need to occur like in the Full testbench above. Take this with warning though, as this DOES NOT verify the configuration protocol for this form of testing. Instead, the benchmark module has the same port mappings for both the FPGA fabric with bitstream instantiated as well as the original RTL design, leading to 100% coverage in verification. This form of verification is very useful when validating a large amount of different benchmarks in one script.

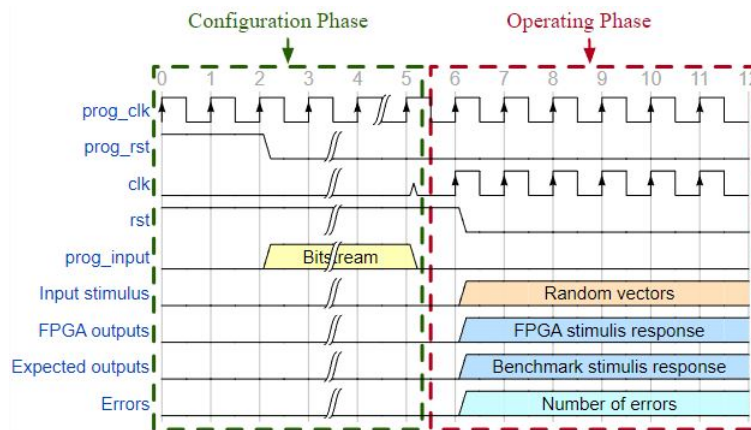


Figure 6: OpenFPGA Internal Tools[7]

2.4 Supplemental Resources

The following list of YouTube videos were included in the user documentation to help first time users:

- **Why OpenFPGA?:** <https://www.youtube.com/watch?v=oc0DUGcYGqo>
- **How to install OpenFPGA?:** <https://www.youtube.com/watch?v=F9sMRmDewM0>
- **How to generate a fabric?:** <https://www.youtube.com/watch?v=aJ00kZ1uh68>
- **Integrating Custom Verilog Modules:** <https://www.youtube.com/watch?v=YTggSZHsTjg>

3 Tool Installation

This section will overview the required operating system, dependencies, git repository, and build commands to properly setup and confirm OpenFPGA setup.

The following user guide was utilized to build the OpenFPGA tool chain[7]: https://openfpga.readthedocs.io/en/master/tutorials/getting_started/compile/

3.1 Operating Systems

OpenFPGA is continuously tested using Ubuntu 20.04. Alongside this, the following operating systems have been tested by community members:

- CentOS 7.8
- CentOS 8
- Ubuntu 18.04
- Ubuntu 21.04
- Ubuntu 22.04

For this independent study, Ubuntu 20.04 was used on a WSL2 Instance of Windows 10, as recommended in the documentation.

3.2 Docker Install

A docker image can be installed on Ubuntu 20.04, which contains pre-compiled OpenFPGA binaries with all dependencies:

```
# To get the docker image from the repository,
docker pull ghcr.io/lнис-uofu/openfpga-master:latest

# Create Local folder to link with Docker Volume
mkdir work

# Create Docker container and volume
docker run -it -v work:/opt/openfpga/
ghcr.io/lнис-uofu/openfpga-master:latest bash

# Navigate to openfpga repo
cd /opt/openfpga/

# Source dependencies
source openfpga.sh
```

```
# Verify tools run
run-task compilation_verification
```

3.3 Dependencies (Without Docker)

An issue arose where I did not have write permissions in the Docker image, so I was unable to run any OpenFPGA tasks to generate netlists or testbenches. For this reason, I opted to install all of the dependencies locally instead on an Ubuntu 20.04 WSL2 instance on my Windows 10 computer. This had no issues.

Dependencies to build source code:

```
sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install -y \
    autoconf \
    automake \
    bison \
    ccache \
    cmake \
    ctags \
    curl \
    doxygen \
    flex \
    fontconfig \
    gdb \
    git \
    gperf \
    iverilog \
    libc6-dev \
    libcairo2-dev \
    libevent-dev \
    libffi-dev \
    libfontconfig1-dev \
    liblist-moreutils-perl \
    libncurses5-dev \
    libreadline-dev \
    libreadline8 \
    libx11-dev \
    libxft-dev \
    libxml++2.6-dev \
    make \
    perl \
```

```
pkg-config \  
python3 \  
python3-setuptools \  
python3-lxml \  
python3-pip \  
qt5-default \  
tcllib \  
tcl8.6-dev \  
texinfo \  
time \  
valgrind \  
wget \  
zip \  
swig \  
expect \  
g++-7 \  
gcc-7 \  
g++-8 \  
gcc-8 \  
g++-9 \  
gcc-9 \  
g++-10 \  
gcc-10 \  
g++-11 \  
gcc-11 \  
clang-6.0 \  
clang-7 \  
clang-8 \  
clang-10 \  
clang-format-10 \  
libxml2-utils \  
libssl-dev \  
gtkwave
```

Packages for regression tests:

```
# Update as required by some packages  
sudo apt-get update  
sudo apt-get install --no-install-recommends -y \  
libdatetime-perl libc6 libffi-dev libgcc1 libreadline8 \  
libstdc++6 libtcl8.6 tcl python3.8 python3-pip zlib1g \  
libbz2-1.0 iverilog git rsync make curl wget tree \  
python3.8-venv
```

3.4 Build Steps

Use the following steps to clone the OpenFPGA git repository and build the project, without using Docker:

Clone Repository:

```
git clone https://github.com/LNIS-Projects/OpenFPGA.git
```

Python packages required:

```
cd OpenFPGA
python3 -m pip install -r requirements.txt
```

Build source code:

```
cd OpenFPGA
make all
```

To verify that the tool has successfully compiled, run the following Python script in the project root directory:

```
python3 openfpga_flow/scripts/run_fpga_task.py \
compilation_verification --debug --show_thread_logs
```

```
jmhafele@DESKTOP-SA1D2UB:~/OpenFPGA$ run-task compilation_verification
INFO ( MainThread) - Set up to run 2 Parallel threads
INFO ( MainThread) - Currently running task compilation_verification
INFO ( MainThread) - Created "run001" directory for current task run
INFO ( MainThread) - Running "vpr_blif" flow
INFO ( MainThread) - Found 1 Architectures 1 Benchmarks & 1 Script Parameters
INFO ( MainThread) - Created total 1 jobs
INFO (00_and2_MIN_ROUTE_CHAN_WIDTH) - 00_and2_MIN_ROUTE_CHAN_WIDTH Finished with returncode 0, Time Taken a second
INFO (00_and2_MIN_ROUTE_CHAN_WIDTH) - ***** 0 runs pending *****
INFO ( MainThread) - Task execution completed
```

Figure 7: Compilation Verification Task Output

4 Running Tools

4.1 Folder Structure

In the root directory of the OpenFPGA project, the following directories are provided:

```
OpenFPGA
├── .github
├── build
├── cmake
├── dev:
├── docker
├── docs
├── libs
├── openfpga
├── openfpga_flow
├── vtr-verilog-to-routing
├── yosys
├── yosys-plugins
└── openfpga.sh
```

- **.github**, **build**, **cmake**, **dev**, **docker**, **libs**, and **openfpga** are all used to either compile or develop the OpenFPGA repository. These folders should not be modified while using OpenFPGA.
- **docs** can be used to rebuild the OpenFPGA user documentation.
- **openfpga_flow** is used to create tasks, cell libraries, and FPGA architecture definitions
- **vtr-verilog-to-routing**, **yosys**, and **yosys-plugins** are all submodules that are used for architecture definitions and compilations.
- **openfpga.sh** is used to source commands referenced in section 4.2.

4.2 Shell Commands

A set of commands can be run after the `openfpga.sh` script is sourced in the root OpenFPGA direction:

```
export OPENFPGA_PATH=<path-to-openfpga-repository-root>
cd ${OPENFPGA_PATH} && source openfpga.sh
```

The following list of commands can now be run:

- **list-tasks**
Lists all OpenFPGA tasks from current task directory. Default task director is `OpenFPGA/openfpga_flow/tasks`

- `run-task <task_name> *kwarags`
Runs the specified task. Will first look in current working directory, then search in defined `TASK_DIRECTORY`. Can also provide a path as a task name.
- `create-task <task_name> <template>`
Create a template task in current directory with given task name. The template is optional, and can be configured as one of two options. `vpr_blif` is a template for running toolflow with a `.blif` file as an input (VPR + netlist generation). `yosys_vpr` is a template for running flow with a Verilog file as an input (Synthesis + VPR + Netlist generation). This command can also be used to copy example projects.
- `goto_task <task_name> <run_num[default 0]>`
This command will change directories to a specified run-directory of the given task.
- `clear-task-run <task_name>`
Clear all run directories of a given task.
- `run-modelsim <task_name>`
Runs verification using ModelSim. Test benches are generated during the toolflow run. VSIM must be installed and configured.
- `run-regression-local`
Runs the regression test locally using the current version of OpenFPGA.
- `unset-openfpga`
Unregisters all shortcuts and commands from current shell.

4.3 OpenFPGA Tasks

Important Notes:

- Task config is setup under `OPENFPGA_PATH/openfpga_flow/tasks/TASK_PATH/config`
- All tasks are defined under: `OPENFPGA_PATH/openfpga_flow/tasks`
- All FPGA Architecture XML's are located under: `OPENFPGA_PATH/openfpga_flow/openfpga_arch/` or `OPENFPGA_PATH/vpr_arch/`
- All OpenFPGA Shell Scripts are located under: `OPENFPGA_PATH/openfpga_flow/openfpga_shell_scripts/`
- All benchmarks are located under: `OPENFPGA_PATH/openfpga_flow/benchmarks/`
- Generated netlists written to `openfpga_verilog_output_dir`

- Multiple benchmarks can be utilized by incrementing the number up by 1 for every new benchmark

Defining Task Config Parameters:

- General
 - **power_tech_file:** XML File for power analysis
 - **power_analysis:** Set to 'True' if using power analysis
 - **spice_output:** Set to 'True' if generating SPICE output
 - **verilog_output:** Set to 'True' if generating Verilog FPGA Fabric netlist
 - **arch_variable_file:** yml file in local task configuration to define benchmark variables
- OpenFPGA Shell
 - **openfpga_shell_template:** Shell script to run under OPENFPGA_PATH/openfpga_flow/openfpga_shell_scripts/
 - **openfpga_arch_file:** OpenFPGA XML Architecture file sourced under OPENFPGA_PATH/openfpga_flow/openfpga_arch/
 - **openfpga_sim_setting_file:** Simulation settings under OPENFPGA_PATH/openfpga_flow/openfpga_simulation_settings/
 - **openfpga_verilog_output_dir:** Output directory for generated Verilog fabric netlists
- Architectures
 - **arch0:** VPR XML Architecture file sourced under OPENFPGA_PATH/openfpga_flow/vpr_arch/
- Benchmarks
 - **bench0:** Benchmark to validate generated FPGA fabric OPENFPGA_PATH/openfpga_flow/benchmarks/BENCHMARK_PATH/BENCHMARK.blif
- Synthesis
 - **bench0_top:** Name of benchmark folder
 - **bench0_act:** .act file for benchmark OPENFPGA_PATH/openfpga_flow/benchmarks/BENCHMARK_PATH/BENCHMARK.act
 - **bench0_verilog:** .v file for benchmark OPENFPGA_PATH/openfpga_flow/benchmarks/BENCHMARK_PATH/BENCHMARK.v

Additional notes on Tasks: https://openfpga.readthedocs.io/en/master/manual/openfpga_flow/run_fpga_task/#creating-a-new-openfpga-task

4.4 OpenFPGA Flow

An alternative way to run OpenFPGA with one benchmark and XML architecture definition is by utilizing the provided python script under path `OPENFPGA_PATH/openfpga_flow/scripts/run_fpga_flow.py`.

The minimum command line arguments are required:

```
open_fpga_flow.py <architecture_file> \  
<benchmark_files> --top_module <top_module_name>
```

File Descriptions for OpenFPGA Flow:

- [**architecture_file**]: Target FPGA architecture
- [**benchmark_files**]: List of benchmark files to test at `/path/to/benchmarks/*.v`
- [**top_module_name**]: Name of top level module in Verilog project

This script will create a `/tmp/` directory under the root OpenFPGA repository path, which will overwrite any previous contents to the directory when a new flow is run. Architecture files will be copied to the `/tmp/arch/` directory and benchmark files will be copied to the `/tmp/bench/` directory before execution.

4.5 Fabric Netlist Generation

Many sample tasks are given to run to provide net-list generation, including the below `generate_fabric` task under the `basic_tests` folder. This task generates an FPGA fabric net list based on a defined FPGA architecture XML file (`arch_variable_file` and `arch0`), which can then be compiled using `iVerilator` or `ModelSim`.

Run `generate_fabric` Task in project root folder:

```
python3 openfpga_flow/scripts/run_fpga_task.py \  
basic_tests/generate_fabric  
  
# If sourced openfpga.sh, can instead run:  
run-task basic_tests/generate_fabric
```

Once Verilog netlists are generated, they can be compiled with `iVerilator`:

```

jmhafel@DESKTOP-SA1D2UB:~/OpenFPGA$ run-task basic_tests/generate_fabric
INFO ( MainThread) - Set up to run 2 Parallel threads
INFO ( MainThread) - Currently running task basic_tests/generate_fabric
INFO ( MainThread) - Created "run001" directory for current task run
INFO ( MainThread) - Running "vpr_blif" flow
INFO ( MainThread) - Found 1 Architectures 1 Benchmarks & 1 Script Parameters
INFO ( MainThread) - Created total 1 jobs
INFO (00_and2_MIN_ROUTE_CHAN_WIDTH) - 00_and2_MIN_ROUTE_CHAN_WIDTH Finished with returncode 0, Time Taken a moment
INFO (00_and2_MIN_ROUTE_CHAN_WIDTH) - ***** 0 runs pending *****
INFO ( MainThread) - Task execution completed

```

Figure 8: Generate Fabric Task Output

```

cd ${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/
generate_fabric/latest/k6_frac_N10_tileable_40nm/
and2/MIN_ROUTE_CHAN_WIDTH

iverilog SRC/fabric_netlists.v

```

After the iverilog command is run, it can be verified that the generated output a.out is made in the output folder for the generated netlists9.

```

jmhafel@DESKTOP-SA1D2UB:~/OpenFPGA/openfpga_flow/tasks/basic_tests/generate_fabric/latest/k6_frac_N10_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH$ ls
SRC                and2.route         fabric_hierarchy.txt      report_timing.setup.rpt
and2_ace_out.act   and2_ace_out.act   fabric_io_location.xml   report_unconstrained_timing_hold.rpt
a.out              and2_output_verilog.v netlist_renaming.xml     report_unconstrained_timing_setup.rpt
and2.blif          and2_run.openfpga  openfpgashell.log        vpr_stat.result
and2.net           and2_template.openfpga packing_pin_util.rpt     vpr_stdout.log
and2.net_post_routing arch                pre_pack.report_timing.setup.rpt
and2.place         benchmark          report_timing_hold.rpt

```

Figure 9: Generate Fabric iVerilog Compilation Output

4.6 Simulation

Other task flows can be ran to generate testbenches and matching waveform outputs, to verify that the generated FPGA netlists will function with varying benchmarks. By using the default benchmarks in the `full_testbench/configuration_chain` OpenFPGA Task, a testbench with a 2 input AND gate, 2 input OR gate, and latched 2 input AND gate will all be generated.

Generated results will be placed under the task directory, which includes .ved files to verify designs using a waveform viewer, such as GTKWave. To run additional testbenches, add in source code for behavioral verilog under `OPENFPGA_PATH/openfpga_flow/benchmarks/`. The architecture files for the specific FPGA fabric can be altered under the config file for the `full_testbench/configuration_chain` task under `OPENFPGA_PATH/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain`. Many benchmarks are already included which can be tested for functionality with a waveform viewer, including adders, basic logic gates, clock dividers, dual port RAM, and FIR filters.

Figure 10 references the Task configuration settings when running the testbench task. As outlined before, it sets a power analysis to a 45nm tech node, and designates TRUE for a Verilog output to generate a verilog FPGA fabric netlist. FPGA Architectures can be altered with the `openfpga_arch_file` and `arch0` settings in the config file. Three different benchmarks are defined, under the `BENCHMARKS` section, with an increasing number for each following benchmark.

```
[GENERAL]
run_engine=openfpga_shell
power_tech_file = ${PATH:OPENFPGA_PATH}/openfpga_flow/tech/PTM_45nm/45nm.xml
power_analysis = true
spice_output=false
verilog_output=true
timeout_each_job = 20*60
fpga_flow=yosys_vpr

[OpenFPGA_SHELL]
openfpga_shell_template=${PATH:OPENFPGA_PATH}/openfpga_flow/openfpga_shell_scripts/write_full_testbench_example_script.openfpga
openfpga_arch_file=${PATH:OPENFPGA_PATH}/openfpga_flow/openfpga_arch/k4_N4_40nm_cc_openfpga.xml
openfpga_sim_setting_file=${PATH:OPENFPGA_PATH}/openfpga_flow/openfpga_simulation_settings/auto_sim_openfpga.xml
openfpga_vpr_device_layout=
openfpga_fast_configuration=

[ARCHITECTURES]
arch0=${PATH:OPENFPGA_PATH}/openfpga_flow/vpr_arch/k4_N4_tileable_40nm.xml

[BENCHMARKS]
bench0=${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/and2/and2.v
bench1=${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/or2/or2.v
bench2=${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/and2_latch/and2_latch.v

[SYNTHESIS_PARAM]
bench_read_verilog_options_common = -nolatches
bench0_top = and2
bench0_chan_width = 300

bench1_top = or2
bench1_chan_width = 300

bench2_top = and2_latch
bench2_chan_width = 300

[SCRIPT_PARAM_MIN_ROUTE_CHAN_WIDTH]
end_flow_with_test=
```

Figure 10: full_testbench/configuration_chain Task Config

To generate and view full testbench waveforms:

```
run-task basic_tests/full_testbench/configuration_chain

gtkwave ${OPENFPGA_PATH}/openfpga_flow/tasks/
\basic_tests/full_testbench/configuration_chain/
latest/k4_N4_tileable_40nm/and2/
MIN_ROUTE_CHAN_WIDTH/and2_formal.vcd &
```

Simulation results can be seen in Figure 11. It can be verified that all three

benchmarks (or jobs), have ran serially, verifying that the AND2, OR2, and latched AND2 benchmarks all completed and successfully passed.

```

jmhafel@DESKTOP-SA1D2UB:~/OpenFPGA$ run-task basic_tests/full_testbench/configuration_chain
INFO ( MainThread) - Set up to run 2 Parallel threads
INFO ( MainThread) - Currently running task basic_tests/full_testbench/configuration_chain
INFO ( MainThread) - Created "run001" directory for current task run
INFO ( MainThread) - Running "yosys_vpr" flow
INFO ( MainThread) - Found 1 Architectures 3 Benchmarks & 1 Script Parameters
INFO ( MainThread) - Created total 3 jobs
INFO (00_or2_MIN_ROUTE_CHAN_WIDTH) - 00_or2_MIN_ROUTE_CHAN_WIDTH Finished with returncode 0, Time Taken a second
INFO (00_or2_MIN_ROUTE_CHAN_WIDTH) - ***** 2 runs pending *****
INFO (00_and2_MIN_ROUTE_CHAN_WIDTH) - 00_and2_MIN_ROUTE_CHAN_WIDTH Finished with returncode 0, Time Taken a second
INFO (00_and2_MIN_ROUTE_CHAN_WIDTH) - ***** 1 runs pending *****
INFO (00_and2_latch_MIN_ROUTE_CHAN_WIDTH) - 00_and2_latch_MIN_ROUTE_CHAN_WIDTH Finished with returncode 0, Time Taken a second
INFO (00_and2_latch_MIN_ROUTE_CHAN_WIDTH) - ***** 0 runs pending *****
INFO ( MainThread) - Task execution completed

```

Figure 11: full_testbench/configuration_chain Task Results

As mentioned before, waveforms can be investigated using GTKWave, as seen in Figure 12. This waveform overviews the majority of the 2 input AND gate benchmark with a **full testbench** model. In this model, as introduced previously, the FPGA fabric is programmed with a generated bitstream in the Configuration Phase. It should be noted how long this takes to configure, by incrementing through every bit index of the bitstream to program the K4 N4 FPGA fabric. By running the other testbench mode, the **Formal-oriented Testbench**, simulation times could be reduced.

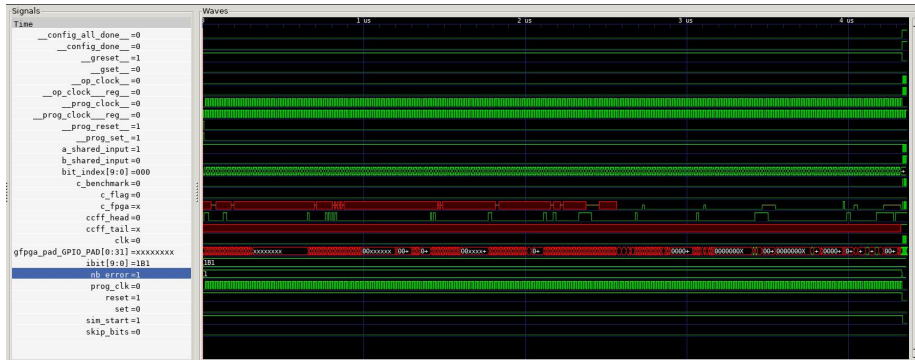


Figure 12: 2 Input AND Configuration Phase

In the Evaluation Phase, seen in Figure 13, the inputs a_shared.input and b_shared.input are randomly driven. The expected output c_benchmark, from the functional AND2 verilog module, is cross-checked with the FPGA fabric output c_fpga. As can be seen, every output matches, so the error counter does not increment, and the benchmark passes. A similar process follows for the other two simulated benchmarks, including a 2 input OR gate and latched 2 input AND gate.

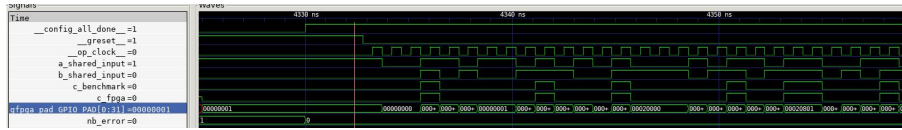


Figure 13: 2 Input AND Evaluation Phase

To generate and view formal-oriented testbench waveforms:

```
run-task basic_tests/preconfig_testbench/
configuration_chain

gtkwave ${OPENFPGA_PATH}/openfpga_flow/tasks/
\basic_tests/preconfig_testbench/configuration_chain/
latest/k4_N4_tileable_40nm/and2/
MIN_ROUTE_CHAN_WIDTH/and2_formal.vcd &
```

A different testbench was generated and simulated based on the `openfpga_shell_template` script. For the formal-oriented testbench, the script `example_script` was ran, which included two additional commands for the shell script, including `write_preconfigured_fabric_wrapper` and `write_preconfigured_testbench`, which would alter the generated testbench to be formal-oriented.

The waveform below¹⁴ was generated by running the `preconfig_testbench` task with the path `tasks/basic_tests/preconfig_testbench/configuration_chain`. This task runs the same three sample benchmarks, including a 2 input AND gate, 2 input OR gate, and latched 2 input AND gate, with the formal-oriented testbench protocol. While this does not test the FPGA configuration, simulation time run for a duration of 8ns against the previous 4359ns for the full testbench for verifying a 2-input AND gate. The full testbench would prove to take even longer with larger FPGA protocols, since it would take longer to index for every bit in the FPGA fabrics bitstream.

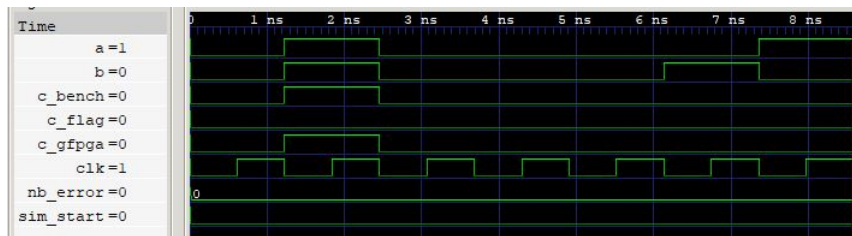


Figure 14: 2 Input AND Formal Results

4.7 Synthesis

Synthesis can be achieved using commercial tools such as Cadence alongside the OpenFPGA toolflow to create GDSII layout files for designated FPGA fabrics. Since the tools are commercially available and not open-source, the scripts provided to complete layout have not been provided. This would be interesting to follow up in future work with open-source tools such as OpenROAD. An example image of an FPGA Layout from the documentation can be seen in Figure 15.

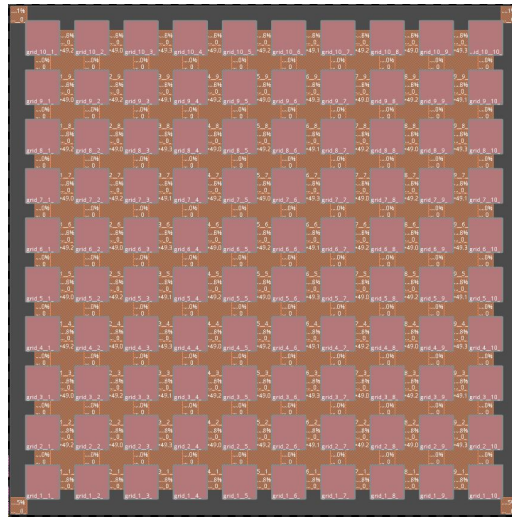


Figure 15: Sample Layout[7]

5 Architecture Modeling

5.1 Custom Verilog Modules

Using OpenFPGA, it is also possible to define custom verilog modules to be generated alongside the FPGA fabric netlist.

To achieve this, we will begin by editing the XML file under the path `openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml`. This new line of code removes the reference for the `verilog_netlist` path, which previously pointed to the `openfpga` cell library and related functional adder description. By removing the netlist path, we will generate an intended error and a user defines template which we can insert our own adder design.

Replace Line's 183 with the following in `k6_frac_N10_adder_chain_40nm_openfpga`:

```
<circuit_model type="hard_logic" name="ADDF" prefix="ADDF"
is_default="true" spice_netlist="${OPENFPGA_PATH}/
openfpga_flow/openfpga_cell_library/spice/adder.sp"
verilog_netlist="">
```

Run the `hard_adder` task at the root directory:

```
source openfpga.sh

run-task fpga_verilog/adder/hard_adder
```

Figure 16 demonstrates the expected iVerilog compilation error after removing the adders original verilog netlist path.

```

jmhafele@DESKTOP-SA1D2UB:~/OpenFPGA$ run-task fpga_verilog/adder/hard_adder
INFO ( MainThread) - Set up to run 2 Parallel threads
INFO ( MainThread) - Currently running task fpga_verilog/adder/hard_adder
INFO ( MainThread) - Created "run003" directory for current task run
INFO ( MainThread) - Running "yosys_vpr" flow
INFO ( MainThread) - Found 1 Architectures 1 Benchmarks & 1 Script Parameters
INFO ( MainThread) - Created total 1 jobs
ERROR (00_adder_8_MIN_ROUTE_CHAN_WIDTH) - Failed to execute openfpga flow - 00_adder_8_MIN_ROUTE_CHAN_WIDTH
Traceback (most recent call last):
  File "/home/jmhafele/OpenFPGA/openfpga_flow/scripts/run_fpga_task.py", line 565, in run_single_script
    raise subprocess.CalledProcessError(0, ".join(command))
subprocess.CalledProcessError: Command 'python3 /home/jmhafele/OpenFPGA/openfpga_flow/scripts/run_fpga_flow.py /home/jmhafele/OpenFPGA/openfpga_flow/vpr_arch/k6_frac_N10_tileable_adder_chain_40nm.xml /home/jmhafele/OpenFPGA/openfpga_flow/benchmarks/micro_benchmark/adder/adder_8/adder_8.v --top module adder_8 --run_dir /home/jmhafele/OpenFPGA/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/run003/k6_frac_N10_tileable_adder_chain_40nm/adder_8/MIN_ROUTE_CHAN_WIDTH --fpga_flow yosys_vpr --openfpga_shell_template /home/jmhafele/OpenFPGA/openfpga_flow/openfpga_shell_scripts/example_without_ace_script.openfpga --openfpga_arch_file /home/jmhafele/OpenFPGA/openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm.openfpga.xml --openfpga_sim_setting_file /home/jmhafele/OpenFPGA/openfpga_flow/openfpga_simulation_settings/fixd_sim.openfpga.xml --openfpga_pin_constraints_file /home/jmhafele/OpenFPGA/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/config/default_dummy_pin_constraints.xml --yosys_tmpl /home/jmhafele/OpenFPGA/openfpga_flow/misc/ys_tmpl_yosys_vpr_adder_flow.ys --ys_rewrite_tmpl /home/jmhafele/OpenFPGA/openfpga_flow/misc/ys_tmpl_yosys_vpr_flow_with_rewrite.ys;/home/jmhafele/OpenFPGA/openfpga_flow/misc/ys_tmpl_rewrite_flow.ys --vpr_fpga_verilog --vpr_fpga_verilog_dir /home/jmhafele/OpenFPGA/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/run003/k6_frac_N10_tileable_adder_chain_40nm/adder_8/MIN_ROUTE_CHAN_WIDTH --vpr_fpga_x2p_rename_illegal_port --end_flow_with_test --vpr_fpga_verilog_formal_verification_top_netlist --flow_config /home/jmhafele/OpenFPGA/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/config/task.conf --TOP adder_8 --READ_VERILOG_OPTIONS --nolatches --YOSYS_ADDER_MAP_VERILOG /home/jmhafele/OpenFPGA/openfpga_flow/openfpga_yosys_techlib/openfpga_arith_map.v --YOSYS_CELL_SIM_VERILOG /home/jmhafele/OpenFPGA/openfpga_flow/openfpga_yosys_techlib/openfpga_adders_sim.v' returned non-zero exit status 0.

```

Figure 16: Synthesis Fail with No Verilog Netlist Adder Path

While the compilation failed, the Verilog file `user_defined_templates` was created which can be utilized to reference a custom Verilog adder.

Path to replace code with: `openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_adder_chain_40nm/adder_8/MIN_ROUTE_CHAN_WIDTH/SRC/sub_module/user_defined_templates.v`

Add the following ADFF module to `user_define_templates` :

```

module ADFF(A,
            B,
            CI,
            SUM,
            CO);
//----- INPUT PORTS -----
input [0:0] A;
//----- INPUT PORTS -----
input [0:0] B;
//----- INPUT PORTS -----
input [0:0] CI;
//----- OUTPUT PORTS -----
output [0:0] SUM;
//----- OUTPUT PORTS -----
output [0:0] CO;

//----- BEGIN wire-connection ports -----

```



```

//----- END wire-connection ports -----

//----- BEGIN Registered ports -----
//----- END Registered ports -----

// ----- Internal logic should start here -----
    assign SUM = A ^ B ^ CI;
    assign CO = (A & B) | (A & CI) | (B & CI);
// ----- Internal logic should end here -----
endmodule

```

Now, go back to the previously edited XML file, and under Line 183, set the following path for the verilog_netlist: `${OPENFPGA_PATH}/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/**YOUR_RUN_NUMBER**/k6_frac_N10_tileable_adder_chain_40nm/adder_8/MIN_ROUTE_CHAN_WIDTH/SRC/sub_module/user_defined_templates.v`.

NOTE: The path states "YOUR_RUN_NUMBER" since the newest run of the defines will switch the symlink to latest. By default, the first run of the Hard Adder task should be "run001".

This will guarantee that the newly edited user define template, which contains our new adder definition, will be referenced in the architecture XML file.

Re-run the hard_adder task at the root directory:

```
run-task fpga_verilog/adder/hard_adder
```

Figure 17 demonstrates a successful run with the implemented custom Verilog adder to be used in the CLB for the FPGA fabric configuration.

```

jmhafel@DESKTOP-SA1D2UB:~/OpenFPGA$ run-task fpga_verilog/adder/hard_adder
INFO ( MainThread) - Set up to run 2 Parallel threads
INFO ( MainThread) - Currently running task fpga_verilog/adder/hard_adder
INFO ( MainThread) - Created "run018" directory for current task run
INFO ( MainThread) - Running "yosys_vpr" flow
INFO ( MainThread) - Found 1 Architectures 1 Benchmarks & 1 Script Parameters
INFO ( MainThread) - Created total 1 jobs
INFO (00_adder_8_MIN_ROUTE_CHAN_WIDTH) - 00_adder_8_MIN_ROUTE_CHAN_WIDTH Finished with returncode 0, Time Taken 31 seconds
INFO (00_adder_8_MIN_ROUTE_CHAN_WIDTH) - ***** 0 runs pending *****
INFO ( MainThread) - Task execution completed

```

Figure 17: Custom Verilog Adder Successful Run

Figure 18 demonstrates verified functionality through the formal-oriented testbench. As before, the inputs are driven for a, b, and cin and the benchmark output is cross-checked with the output from the mapped FPGA fabric to validate the adder.

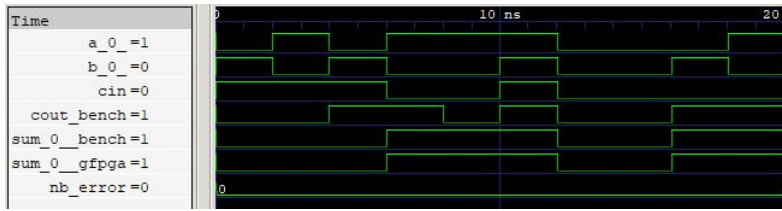


Figure 18: Custom Verilog Adder Waveform Results

5.2 Custom Cell Library

Open-source standard cell libraries can also be utilized with OpenFPGA, such as the open-source SkyWater 130nm PDK.

First, clone and make the Skywater PDK into the root OpenFPGA folder:

```
git clone https://github.com/google/skywater-pdk.git
```

Next, run the following make command in the cloned Skywater PDK:

```
cd skywater-pdk/

SUBMODULE_VERSION=latest make submodules -j3 \
|| make submodules -j1
```

With the Skywater PDK installed, the OpenFPGA XML architecture file can now be edited. For this example, we will edit the XML file under the path `openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml`, and utilize the task `fpga_verilog/adder/hard_adder`.

First, open the OpenFPGA XML configuration file under the path `openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml`. The code snippet below will be the first to change to the architecture, being that the model for the 2 input OR gate will be updated to use the Skywater PDK or cell in the `sky130_fd_sc_1s` library. Most notable, the name, prefix, verilog netlist path, and port names (prefixes) will all be updated to match the Skywater PDK 2 input OR cell.

NOTE: The documentation online stated to update lines 67 to 81, but it appears the XML file has been updated since then. Look for the standard OR gate under the circuit library section near the top of the XML file.

Replace Line's 68 to 82 with the following in `k6_frac_N10_adder_chain_40nm_openfpga`:

```

<circuit_model type="gate" name="sky130_fd_sc_ls__or2_1"
prefix="sky130_fd_sc_ls__or2_1"
verilog_netlist="${OPENFPGA_PATH}/skywater-pdk/libraries
/sky130_fd_sc_ls/latest/cells/or2/sky130_fd_sc_ls__or2_1.v">
  <design_technology type="cmos" topology="OR"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="input" prefix="A" size="1"/>
  <port type="input" prefix="B" size="1"/>
  <port type="output" prefix="X" size="1"/>
</circuit_model>

```

A second change must also occur at Line 156, and replaced with the following code below. This will ensure that the added skywater PDK OR gate, with an updated circuit model name, will now be utilized when defining the netlists for the Look-Up Tables.

NOTE: Like before, the referenced line numbers were off, and the referenced documentation noted Line 160 should change. The updated line should be the OR gate in the LUT with name `frac_lut6`.

Replace Line 156 with the following in `k6_frac_N10_adder_chain_40nm_openfpga`:

```

<port type="input" prefix="in" size="6" tri_state_map="----11"
circuit_model_name="sky130_fd_sc_ls__or2_1"/>

```

Run FPGA-Verilog task to generate adder benchmark with Skywater PDK:

```

source openfpga.sh

run-task fpga_verilog/adder/hard_adder

```

At this point, when attempting to run the `hard_adder` task, a compilation will occur in `iverilog`. Luckily, an `iverilog` output file has been generated, which can be updated and manually recompiled to generate a FPGA fabric netlist with our Skywater cell and a pairing waveform output.

To solve this, open the generated output `iverilog_output.txt` at path `openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/iverilog_output.txt`. Next, we will include the path to the 2 input or gate in the skywater pdk instead of the `/SRC/` directory local to the task flow.

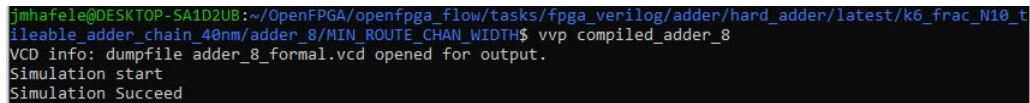
Replace all text inside `iverilog_output.txt` with the following:

```
iverilog -o compiled_adder_8 \  
./SRC/adder_8_include_netlists.v \  
-s adder_8_top_formal_verification_random_tb -I \  
${OPENFPGA_PATH}/skywater-pdk/libraries/  
sky130_fd_sc_ls/latest/cells/or2
```

Manually recompile using iVerilog:

```
cd openfpga_flow/tasks/fpga_verilog/adder/hard_adder/  
latest/k6_frac_N10_tileable_adder_chain_40nm/adder_8/  
MIN_ROUTE_CHAN_WIDTH/  
  
source iverilog_output.txt  
  
vvp compiled_adder_8
```

Figure 19 displays the expected terminal output after manually running iVerilog and generating waveform results.



```
jmhafele@DESKTOP-SA1D2UB:~/OpenFPGA/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_t  
ileable_adder_chain_40nm/adder_8/MIN_ROUTE_CHAN_WIDTH$ vvp compiled_adder_8  
VCD info: dumpfile adder_8_formal.vcd opened for output.  
Simulation start  
Simulation Succeed
```

Figure 19: Skywater Hard Adder Task Result

Figure 20 shows the synthesized netlist including the sky130 PDK 2 input OR gates in the `luts.v` Verilog module. The path for this generated result is `openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/sub_module/luts.v`

```

61 wire [0:0] sky130_fd_sc_ls_or2_1_0_X;
62 wire [0:0] sky130_fd_sc_ls_or2_1_1_X;
63
64 // ---- BEGIN Local short connections ----
65 // ---- END Local short connections ----
66 // ---- BEGIN Local output short connections ----
67 // ---- END Local output short connections ----
68
69 sky130_fd_sc_ls_or2_1 sky130_fd_sc_ls_or2_1_0 (
70   .A(mode[0]),
71   .B(in[4]),
72   .X(sky130_fd_sc_ls_or2_1_0_X));
73
74 sky130_fd_sc_ls_or2_1 sky130_fd_sc_ls_or2_1_1 (
75   .A(mode[1]),
76   .B(in[5]),
77   .X(sky130_fd_sc_ls_or2_1_1_X));
78

```

Figure 20: Skywater Synthesized LUT Netlist

Figure 21 demonstrates a functional output utilizing the Skywater 2 input OR gate as part of the LUT netlist for the FPGA fabric. The inputs a, b, and cin are driven with all 8 possible input combinations, and guarantees matching behavior between the benchmark output and the FPGA fabric output.

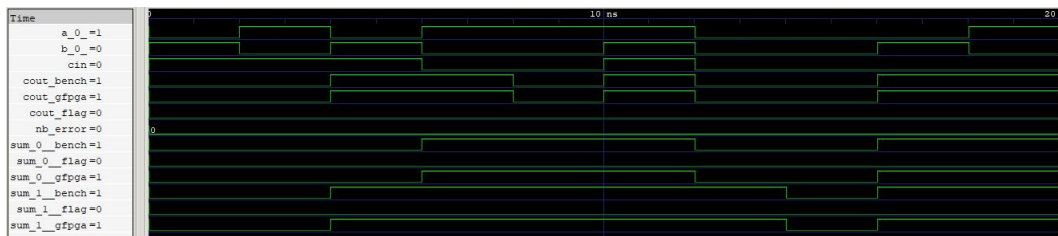


Figure 21: Skywater Hard Adder Task Waveform Results

6 Conclusion

To conclude, through this independent study I was successfully able to clone, build, and run through OpenFPGA by utilizing multiple tasks. I was able to learn more about generating FPGA fabrics, verifying functional correctness, and modifying their architecture with custom functional Verilog designs and custom cell libraries. With this independent study, I was able to integrate the Skywater library that I have been using for senior design in a new light, and connect the dots with other classes such as EE 465. Future work could be done to investigate the place and route flow with either commercial tools such as Cadence or open-source flows such as OpenROAD and the OpenMPW Shuttle project hosted by eFabless.

6.1 Generated Results

The following list of images and results were my own work:

- 3.4 Build Steps
 - Figure 7, Compilation Verification Task Output
- 4.5 Fabric Netlist Generation
 - Figure 8, Generate Fabric Task Output
 - Figure 9, Generate Fabric iVerilog Compilation Output
- 4.6 Simulation
 - Figure 10, full_testbench/configuration_chain Task Config
 - Figure 11, full_testbench/configuration_chain Task Results
 - Figure 12, 2 Input AND Configuration Phase
 - Figure 13, 2 Input AND Evaluation Phase
 - Figure 14, 2 Input AND Formal Results
- 5.1 Custom Verilog Modules
 - Figure 16, Synthesis Fail with No Verilog Netlist Adder Path
 - Figure 17, Custom Verilog Adder Successful Run
 - Figure 18, Custom Verilog Adder Waveform Results
- 5.2 Custom Cell Library
 - Figure 19, Skywater Hard Adder Task Result
 - Figure 20, Skywater Synthesized LUT Netlist
 - Figure 21, Skywater Hard Adder Task Waveform Results

6.2 Future Work

A list of future items to learn more about OpenFPGA:

- Research FPGA-Spice
- Setup scripting for Place and Route with FPGA fabric
- Add more complex benchmark designs, compare tradeoffs of two verification processes
- Integrate more heavily into Skywater 130nm open-source PDK and eFabless

References

- [1] X. Tang, E. Giacomini, B. Chauviere, A. Alacchi, and P.-E. Gaillardon, “Openfpga: An open-source framework for agile prototyping customizable fpgas,” *IEEE Micro*, vol. 40, no. 4, pp. 41–48, 2020.
- [2] J. H. Kim and J. H. Anderson, “Synthesizable fpga fabrics targetable by the verilog-to-routing (vtr) cad flow,” in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2015.
- [3] B. Grady and J. H. Anderson, “Synthesizable heterogeneous fpga fabrics,” in *2018 International Conference on Field-Programmable Technology (FPT)*, pp. 222–229, 2018.
- [4] I. Kuon, A. Egier, and J. Rose, “Design, layout and verification of an fpga using automated tools,” in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, FPGA '05*, (New York, NY, USA), p. 215–226, Association for Computing Machinery, 2005.
- [5] V. Aken'Ova and R. Saleh, “A ”soft++” efpga physical design approach with case studies in 180nm and 90nm,” in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pp. 6 pp.–, 2006.
- [6] H. Liu, “Archipelago—an open source fpga with toolflow support,” FPGA '05, (Univ. California, Berkeley), Association for Computing Machinery, 2014.
- [7] X. Tang, “Welcome to openfpga’s documentation!” <https://openfpga.readthedocs.io/en/master/>.