OpenFPGA Documentation

Release 1.2.0

Xifan Tang

Sep 22, 2022

OVERVIEW

CHAPTER

ONE

WHY OPENFPGA?

Note: If this is your first time learning OpenFPGA, we strongly recommend you to watch the introduction video

OpenFPGA aims to be an open-source framework that enables rapid prototyping of customizable FPGA architectures. As shown in Fig. 1.1, a conventional approach will take a large group of experienced engineers more than one year to achieve production-ready layout and associated CAD tools. In fact, most of the engineering efforts are spent on manual layouts and developing ad-hoc CAD support.

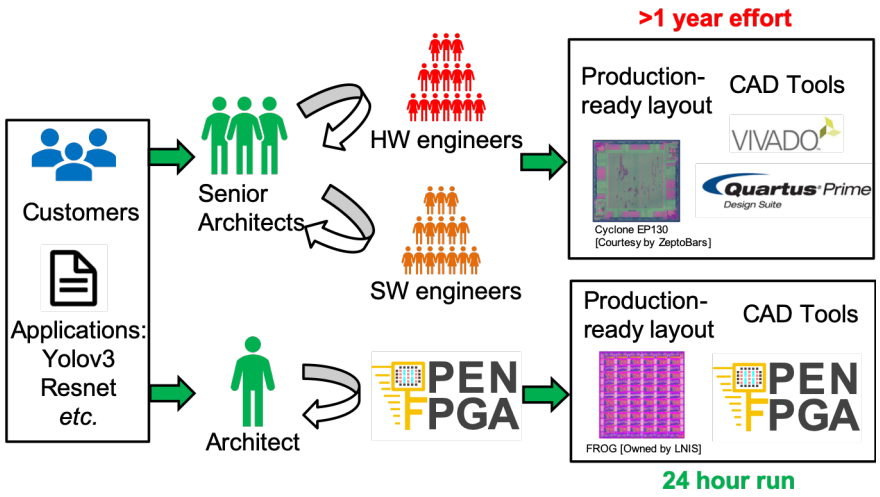


Fig. 1.1: Comparison on engineering time and effort to prototype an FPGA using OpenFPGA and conventional approaches [All the layout figures are publishable under the proper licenses]

Using OpenFPGA, the development cycle in both hardware and software can be significantly accelerated. OpenFPGA can automatically generate Verilog netlists describing a full FPGA fabric based on an XML-based description file. Thanks to modern semi-custom design tools, production-ready layout generation can be achieved within 24 hours. To help sign-off, OpenFPGA can auto-generate Verilog testbenches to validate the correctness of FPGA fabric using modern verification tools. OpenFPGA also provides native bitstream generation support based on the same XML-based description file used in Verilog generation, avoiding the recurring engineering in developing CAD tools for different FPGAs. Once the FPGA architecture is finalized, the CAD tool is ready to use.

OpenFPGA can support any architecture that VPR can describe, covering most of the architecture enhancements available in modern FPGAs, and hence unlocks a large design space in prototyping customizable FPGAs. In addition, OpenFPGA provides enriched syntax which allows users to customize primitive circuits designed down to transistor-level parameters. This helps developers to customize the P.P.A. (Power, Performance and Area) to the best. All these features open the door of prototyping/studying flexible FPGAs to a small group of junior engineers or researchers.

In terms of tool functionality, OpenFPGA consists of the following parts: FPGA-Verilog, FPGA-SDC, FPGA-Bitstream and FPGA-SPICE. The rest of this section will focus on detailed motivation for each of them, as depicted in Fig. 1.2.

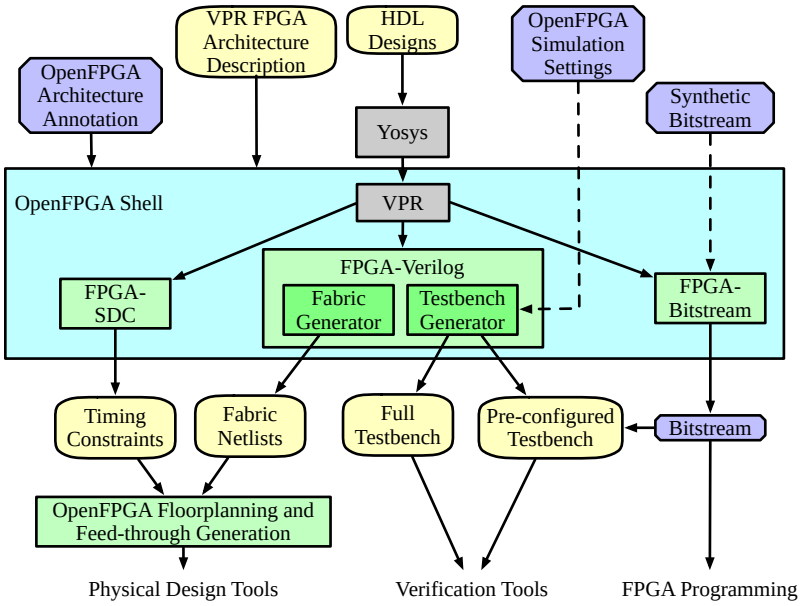


Fig. 1.2: OpenFPGA: a unified framework for chip designer and FPGA programmer

1.1 Fully Customizable Architecture

OpenFPGA supports VPR's architecture description language, which allows users to define versatile programmable fabrics down to point-to-point interconnection. OpenFPGA leverages VPR's architecture description by introducing an XML-based architecture annotation, enabling fully customizable FPGA fabric down to circuit elements. As illustrated in *OpenFPGA architecture description language enabling fully customizable FPGA architecture and circuit-level implementation*, OpenFPGA's architecture annotation covers a complete FPGA fabric, including both the programmable fabric and the configuration peripheral.

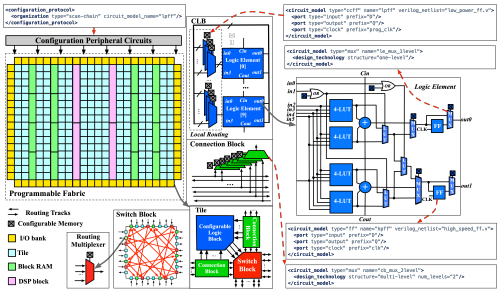


Fig. 1.3: OpenFPGA architecture description language enabling fully customizable FPGA architecture and circuit-level implementation

The technical details can be found in our TVLSI'19 paper [TGMG19] and FPL'19 paper [TGA+19].

1.2 FPGA-Verilog

Driven by the strong need in data processing applications, Field Programmable Gate Arrays (FPGAs) are playing an ever-increasing role as programmable accelerators in modern computing systems. To fully unlock processing capabilities for domain-specific applications, FPGA architectures have to be tailored for seamless cooperation with other computing resources. However, prototyping and bringing to production a customized FPGA is a costly and complex endeavor even for industrial vendors.

OpenFPGA, an opensource framework, aims to rapidly prototype customizable FPGA architectures through a semi-custom design approach. We propose an XML-to-Prototype design flow, where the Verilog netlists of a full FPGA fabric can be autogenerated using an extension of the XML language from the VTR framework and then fed into a back-end flow to generate production-ready layouts. FPGA-Verilog is designed to output flexible and standard Verilog netlists, enabling various backend choices, as illustrated in *FPGA-Verilog enabling flexible backend flows*.

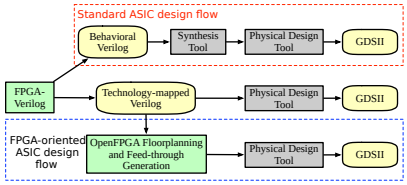


Fig. 1.4: FPGA-Verilog enabling flexible backend flows

The technical details can be found in our TVLSI'19 paper [TGMG19] and FPL'19 paper [TGA+19].

1.3 FPGA-SDC

Design constraints are indepensible in modern ASIC design flows to guarantee the performance level. OpenFPGA includes a rich SDC generator in the OpenFPGA framework to deal with both PnR constraints and sign-off timing analysis. Our flow automatically generates two sets of SDC files.

- The first set of SDC is designed for the P&R flow, where all the combinational loops are broken to enable well controlled timing-driven P&R. In addition, there are SDC files devoted to constrain pin-to-pin timing for all the resources in FPGAs, in order to obtain nicely constrained and homogeneous delays across the fabric. OpenFPGA allows users to define timing constraints in the architecture description and outputs timing constraints in standard format, enabling fully timing constrained backend flow (see *FPGA-SDC enabling iterative timing constrained backend flow*).
- The second set of SDC is designed for the timing analysis of a benchmark at the post P&R stage.

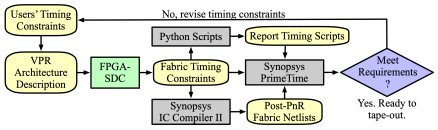


Fig. 1.5: FPGA-SDC enabling iterative timing constrained backend flow

The technical details can be found in our FPL'19 paper [TGA+19].

1.4 FPGA-Bitstream

EDA support is essential for end-users to implement designs on a customized FPGA. OpenFPGA provides a general-purpose bitstream generator FPGA-Bitstream for any architecture that can be described by VPR. As the native CAD tool for any customized FPGA that is produced by FPGA-Verilog, FPGA-Bitstream is ready to use once users finalize the XML-based architecture description file. This eliminates the huge engineering efforts spent on developing bitstream generators for customized FPGAs. Using FPGA-Bitstream, users can launch (1) Verilog-to-Bitstream flow, the typical implementation flow for end-users; (2) Verilog-to-Verification flow. OpenFPGA can output Verilog testbenches with self-testing features to validate users' implemetations on their customized FPGA fabrics.

The technical details can be found in our TVLSI'19 paper [TGMG19] and FPL'19 paper [TGA+19].

1.5 FPGA-SPICE

The built-in timing and power analysis engines of VPR are based on analytical models [BRM99][GW12]. Analytical model-based analysis can promise accuracy only on a limited number of circuit designs for which the model is valid. As the technology advancements create more opportunities on circuit designs and FPGA architectures, the analytical power model requires updates to follow the new trends. However, without referring to simulation results, the analytical power models cannot prove their accuracy. SPICE simulators have the advantages of generality and accuracy over analytical models. For this reason, SPICE simulation results are often selected to check the accuracy of analytical models. Therefore, there is a strong need for a simulation-based power analysis approach for FPGAs, which can support general circuit designs.

It motivates us to develop FPGA-SPICE, an add-on for the current State-of-Art FPGA architecture exploration tools, VPR [RLY+12]. FPGA-SPICE aims at generating SPICE netlists and testbenches for the FPGA architectures supported by VPR. The SPICE netlists and testbenches are generated according to the placement and routing results of VPR. As a result, SPICE simulator can be used to perform precise delay and power analysis. The SPICE simulation results are useful in three aspects: (1) they provide accurate power analysis; (2) they help to improve the accuracy of built-in analytical models; and moreover (3) they create opportunities in developing novel analytical models.

SPICE modeling for FPGA architectures requires detailed transistor-level modeling for all the circuit elements within the considered FPGA architecture. However, current VPR architectural description language [LAR11] does not offer enough transistor-level parameters to model the most common circuit modules, such as multiplexers and LUTs. Therefore, we are developing an extension on the VPR architectural description language to model the transistor-level circuit designs.

The technical details can be found in our ICCD'15 paper [TGM15] and TVLSI'19 paper [TGMG19].

CHAPTER

TWO

TECHNICAL HIGHLIGHTS

The following lists of technical features were created to help users find their needs for customizing FPGA fabrics.(**as of February 2021**)

2.1 Supported Circuit Designs

Circuit Types	Auto-generation	User-Defined	Design Topologies
Inverter	Yes	Yes	• *Power-gated Inverter 1x example* • *Inverter 1x Example* • *Tapered inverter 16x example*
Buffer	Yes	Yes	• *Buffer 2x example* • *Power-gated Buffer 4x example* • *Tapered buffer 64x example*
AND gate	Yes	Yes	• *2-input AND Gate*
OR gate	Yes	Yes	• *2-input OR Gate*
MUX2 gate	Yes	Yes	• *MUX2 Gate*
Pass gate	Yes	Yes	• *Transmission-gate Example* • *Pass-transistor Example*
Look-Up Table	Yes	Yes	• **Any size** • *Single-Output LUT* • *Standard Fracturable LUT* • *LUT with Harden Logic*
Routing Multiplexer	Yes	No	• **Any size** • *Multi-level Multiplexer* • *One-level Multiplexer* • *Tree-like Multiplexer* • *Standard Cell Multiplexer* • *Multiplexer with Local Encoder* • *Multiplexer with Constant Input*
Configurable Memory	No	Yes	• *Configurable Latch* • *SRAM with BL/WL* • *Regular*

- The user defined netlist could come from a standard cell. See *Build an FPGA fabric using Standard Cell Libraries* for details.

2.2 Supported FPGA Architectures

We support most FPGA architectures that VPR can support! The following are the most commonly seen architectural features:

Block Type	Architecture features
Programmable Block	• Single-mode Configurable Logic Block (CLB) • Multi-mode Configurable Logic Block (CLB) • Single-mode heterogeneous blocks • Multi-mode heterogeneous blocks • Flexible local routing architecture
Routing Block	• Tileable routing architecture • Flexible connectivity • Flexible Switch Block Patterns
Configuration Protocol	• Chain-based organization • Frame-based organization • Memory bank organization • Flatten organization

2.3 Supported Verilog Modeling

OpenFPGA supports the following Verilog features in auto-generated netlists for circuit designs

- Synthesizable Behavioral Verilog
- Structural Verilog
- Implicit/Explicit port mapping

CHAPTER

THREE

GETTING STARTED

3.1 How to Compile

Note: We recommend you to watch a tutorial video about how-to-compile before getting started

3.1.1 General Guidelines

OpenFPGA uses CMake to generate the Makefile scripts. In general, please follow the steps to compile

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

Note: OpenFPGA requires gcc/g++ version >5

Note: cmake3.12+ is recommended to compile OpenFPGA with GUI

Note: Recommend using `make -j<int>` to accelerate the compilation, where `<int>` denotes the number of cores to be used in compilation.

Note: VPR's GUI requires gtk-3, and can be enabled with `cmake .. -DVPR_USE_EZGL=on`

Quick Compilation Verification

Note: Ensure that you install python dependences in *Dependencies*.

To quickly verify the tool is well compiled, users can run the following command from OpenFPGA root repository

```
python3 openfpga_flow/scripts/run_fpga_task.py compilation_verification --debug --show_
↪thread_logs
```

3.1.2 Dependencies

Full list of dependencies can be found at install_dependencies_build. In particular, OpenFPGA requires specific versions for the following dependencies:

cmake
version >3.12 for graphical interface

iverilog
version 10.1+ is required to run Verilog-to-Verification flow

python dependencies
python packages are also required:

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

3.1.3 Running with the docker image

Users can skip the traditional installation process by using the Dockerized version of the OpenFPGA tool. The OpenFPGA project maintains the docker image/Github package of the latest stable version of OpenFPGA in the following repository openfpga-master. This image contains precompiled OpenFPGA binaries with all prerequisites installed.

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

# To invoke openfpga_shell
docker run -it ghcr.io/lnis-uofu/openfpga-master:latest openfpga/openfpga bash

# To run the task that already exists in the repository.
docker run -it ghcr.io/lnis-uofu/openfpga-master:latest bash -c "source openfpga.sh &&␣
↪run-task compilation_verification"

# To link the local directory wihth docker
mkdir work

docker run -it -v work:/opt/openfpga/ ghcr.io/lnis-uofu/openfpga-master:latest bash
# Inside container
source openfpga.sh
cd work
create_task _my_task yosys_vpr
```

3.2 OpenFPGA shortcuts

OpenFPGA provides *bash*/*zsh* shell-based shortcuts to perform all essential functions and navigate through the directories. Go to the OpenFPGA directory and source `openfpga.sh`

```
cd ${OPENFPGA_PATH} && source openfpga.sh
```

Note: The OpenFPGA shortcut works with only a bash-like shell. e.g., *bash*/*zsh*/*fish,* etc.

3.2.1 Shortcut Commands

Once the `openfpga.sh` script is sourced, you can run any following commands directly in the terminal.

`list-tasks`

This command lists all the OpenFPGA tasks from the current task directory. default task directory is considered as `${OPENFPGA_PATH}/openfpga_flow/tasks`

`run-task` `<task_name> **kwarags`

This command runs the specified task. The script will first look for the task in the current working directory. If it is not in the current directory, it will then search in `TASK_DIRECTORY` (relative to task directory). You can also provide a path as a task_name, for example, `run-task basic_tests/generate_fabric` The valid arguments listed here <_openfpga_task_args>`_, you can also run *run-task run-task* to get the list of command-line arguments.

`create-task` `<task_name> <template>`

It creates a template task in the current directory with the given task_name. the template is an optional argument; there are two templates currently configured - `vpr_blif`: A template task for running flow with *.blif* file as an input (VPR + Netlist generation) - `yosys_vpr`: A template task for running flow with *.v* file as an input (Synthesis + VPR + Netlist generation) you can also use this command to copy any example project; use a `list-tasks` command to get the list of example projects for example `create-task _my_task_copy basic_tests/generate_fabric` create a copy of the `basic_tests/generate_fabric` task in the current directory with `_my_task_copy` name.

`run-modelsim`

This command runs the verification using ModelSim. The test benches are generated during the OpenFPGA run. **Note**: users need to have `VSIM` installed and configured

`run-regression-local`

This script runs the regression test locally using the current version of OpenFPGA. **NOTE** Important before making a pull request to the master

`unset-openfpga`

Unregisters all the shortcuts and commands from the current shell session

3.3 Supported Tools

3.3.1 Internal Tools

To enable various design purposes, OpenFPGA integrates several tools to i.e., FPGA-Verilog, FPGA-SDC and FPGA-bitstream (highlighted green in *OpenFPGA tool suites and design flows*, with other popular open-source EDA tools, i.e., VPR and Yosys.

3.3.2 Third-Party Tools

OpenFPGA accepts and outputs in standard file formats, and therefore can interface a wide range of commercial and open-source tools.

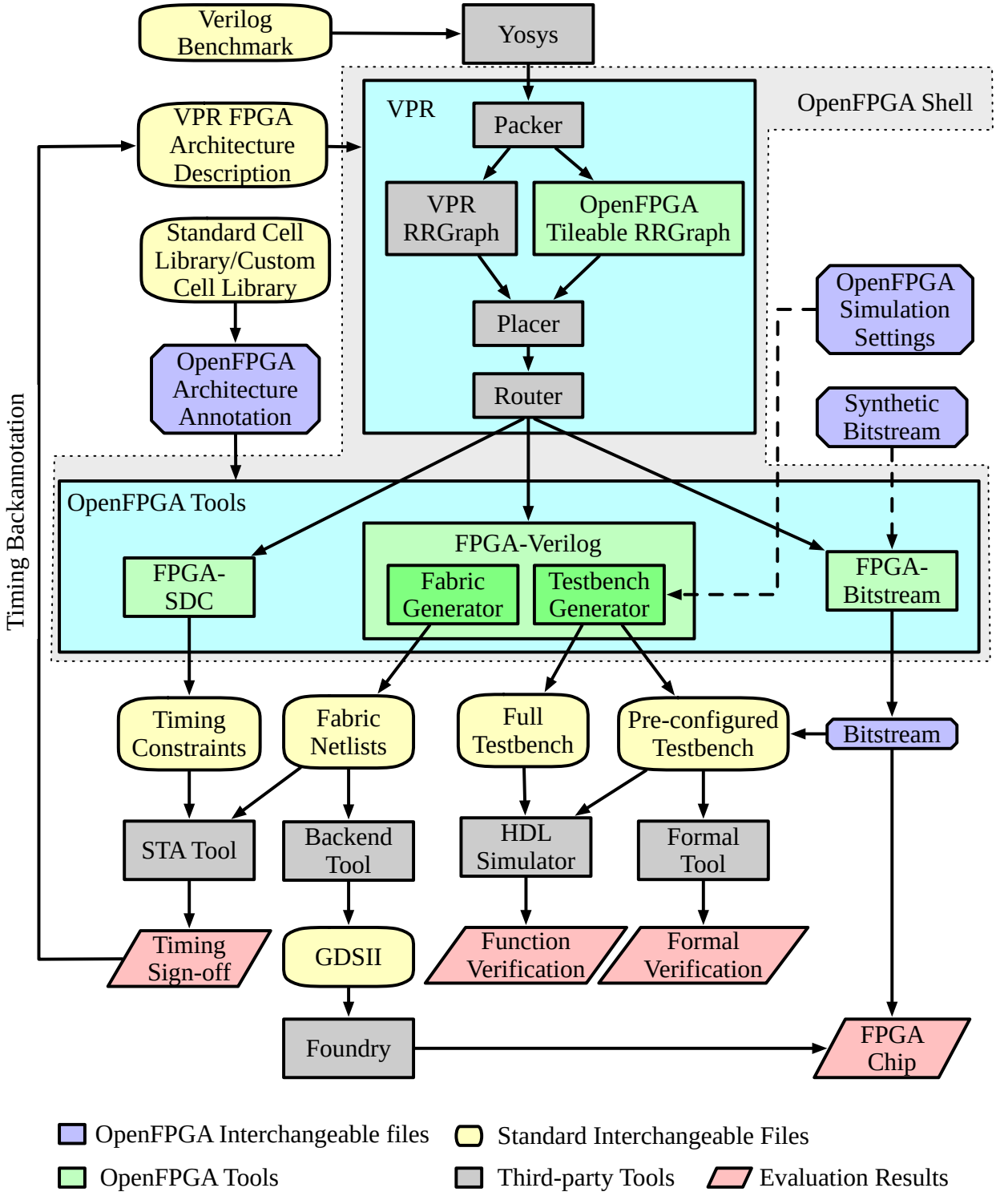


Fig. 3.1: OpenFPGA tool suites and design flows

Usage	Tools	Version Requirement
Back-end	Synopsys IC Compiler II	v2019.03 or later
	Cadence Innovus	v19.1 or later
Timing Analyzer	Synopsys PrimeTime	v2019.03 or later
	Cadence Tempus	v19.15 or later
Verification	Synopsys VCS	v2019.06 or later
	Synopsys Formality	v2019.03 or later
	Mentor ModelSim	v10.6 or later
	Mentor QuestaSim	v2019.3 or later
	Cadence NCSim	v15.2 or later
	Icarus iVerilog	v10.1 or later

- The version requirements is based on our local tests. Older versions may work.

CHAPTER

FOUR

DESIGN FLOWS

4.1 Generate Fabric Netlists

Note: You may watch the video representation of this tutorial

This tutorial will show an example how to

- generate Verilog netlists for a FPGA fabric

Note: Before running any design flows, please checkout the tutorial *How to Compile*, to ensure that you have an operating copy of OpenFPGA installed on your computer.

4.1.1 Prepare Task Configuration File

OpenFPGA provides push-button scripts for users to run design flows (see details in *OpenFPGA Task*). Users can customize their flow-run by crafting a task configuration file.

Here, we consider an existing test case `generate_fabric`. In the task configuration file, you can specify the XML-based architecture files in `LINE 21` and `LINE 25` that describe the architecture of the FPGA fabric. In this example, we are using a low-cost FPGA architecture similar to the lattice ICE40 series

Also, in `LINE 20`, you can specify the openfpga shell script to be executed. Here, we are using an example script which is golden reference to generate Verilog netlists

Note: You can use text editor to customize the configuration file. Here, we use it as is.

4.1.2 Run OpenFPGA Task

After finalizing your configuration file, you can run the task by calling the python script with the given path to task configuration file.

```
python3 openfpga_flow/scripts/run_fpga_task.py basic_tests/generate_fabric
```

When the flow run is executed, you can visit the runtime directory and check the Verilog netlists.

Note that your task-run outcomes are stored in the directory called `latest` in the same level of your task configuration file.

The Verilog netlists are generated in the following directory

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/generate_fabric/latest/k6_frac_N10_
↪tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC
```

Note: `${OPENFPGA_PATH}` is the root directory of OpenFPGA

Note: See *Fabric Netlists* for the netlist details.

In the Verilog files, you can validate if the Verilog description is consistent as your definition in the architecture file. The Verilog files can be then used to drive different tools, such as layout generation *etc*.

4.1.3 Run icarus iVerilog Compilation

Go to the directory

```
cd ${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/generate_fabric/latest/k6_frac_N10_
↪tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH
```

Compile with iVerilog command:

```
iverilog SRC/fabric_netlists.v
```

Note: Please ensure that iVerilog is installed correctly on your computer

If compilation is successful, you can see a file `a.out` in the directory.

4.2 From Verilog to Verification

This tutorial will show an example how to

- generate Verilog netlists for a FPGA fabric
- generate Verilog testbenches for a RTL design
- run HDL simulation to verify the functional correctness of the implemented FPGA fabric

Note: Before running any design flows, please checkout the tutorial *How to Compile*, to ensure that you have an operating copy of OpenFPGA installed on your computer.

4.2.1 Netlist Generation

We will use the openfpga_flow scripts (see details in *OpenFPGA Task*) to generate the Verilog netlists and testbenches. Here, we consider a representative but fairly simple FPGA architecture, which is based on 4-input LUTs. We will map a 2-input AND gate to the FPGA fabric, and run a full testbench (see details in *Testbench*)

We will simply execute the following openfpga task-run by

```
python3 openfpga_flow/scripts/run_fpga_task.py basic_tests/full_testbench/configuration_
↪chain
```

Detailed settings, such as architecture XML files and RTL designs, can be found at `${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/config/task.conf`.

Note: `${OPENFPGA_PATH}` is the root directory of OpenFPGA

After this task-run, you can find all the generated netlists and testbenches at

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/
```

Note: See *Fabric Netlists* and *Testbench* for the netlist details.

4.2.2 Run icarus iVerilog Simulation

Through OpenFPGA Scripts

By default, the `configuration_chain` task-run will execute iVerilog simulation automatically. The simulation results are logged in

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/vvp_sim_output.txt
```

If the verification passed, you should be able to see `Simulation Succeed` in the log file.

All the waveforms are stored in the `and2_formal.vcd` file. To visualize the waveforms, you can use the GTKWave.

```
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 &
```

Manual Method

If you want to run iVerilog simulation manually, you can follow these steps:

```
cd ${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH

source iverilog_output.txt

vvp compiled_and2
```

Debugging Tips

If you want to apply full visibility to the signals, you need to change the following line in

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/and2_autocheck_top_tb.v
```

from

```
$dumpvars (1, and2_autocheck_top_tb);
```

to

```
$dumpvars (12, and2_autocheck_top_tb);
```

4.2.3 Run Modelsim Simulation

Alternatively, you can run Modelsim simulations through openfpga_flow scripts or manually.

Note: Before starting, please ensure that Mentor Modelsim has been correctly installed on your local environment.

Through OpenFPGA Scripts

You can simply call the python script in the following line:

```
python3 openfpga_flow/scripts/run_modelsim.py basic_tests/full_testbench/configuration_
↪chain --run_sim
```

The script will automatically create a Modelsim project at

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/MSIM2/
```

and run the simulation.

You may open the project and visualize the simulation results.

Manual Method

Modify the `fpga_defines.v` (see details in *Fabric Netlists*) at

```
${OPENFPGA_PATH}/openfpga_flow/tasks/openfpga_shellfull_testbench//configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/
```

by **deleting** the line

```
`define ICARUS_SIMULATOR 1
```

Create a folder `MSIM` under

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/
```

Under the MSIM folder, create symbolic links to SRC folder and reference benchmarks by

```
ln -s ../SRC ./

ln -s ../and2_output_verilog.v ./
```

Note: Depending on the operating system, you may use other ways to create the symbolic links

Launch ModelSim under the MSIM folder and create a project by following Modelsim user manuals.

Add the following file to your project:

```
${OPENFPGA_PATH}/openfpga_flow/tasks/basic_tests/full_testbench/configuration_chain/
↪latest/k4_N4_tileable_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/and2_include_netlists.v
```

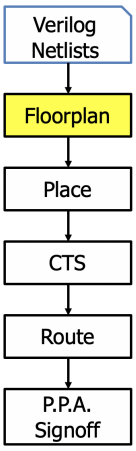
Compile the netlists, create a simulation configuration and specify and2_autocheck_top_tb at the top unit.

Execute simulation with run -all You should see Simulation Succeed in the output log.

4.3 From Verilog to GDSII

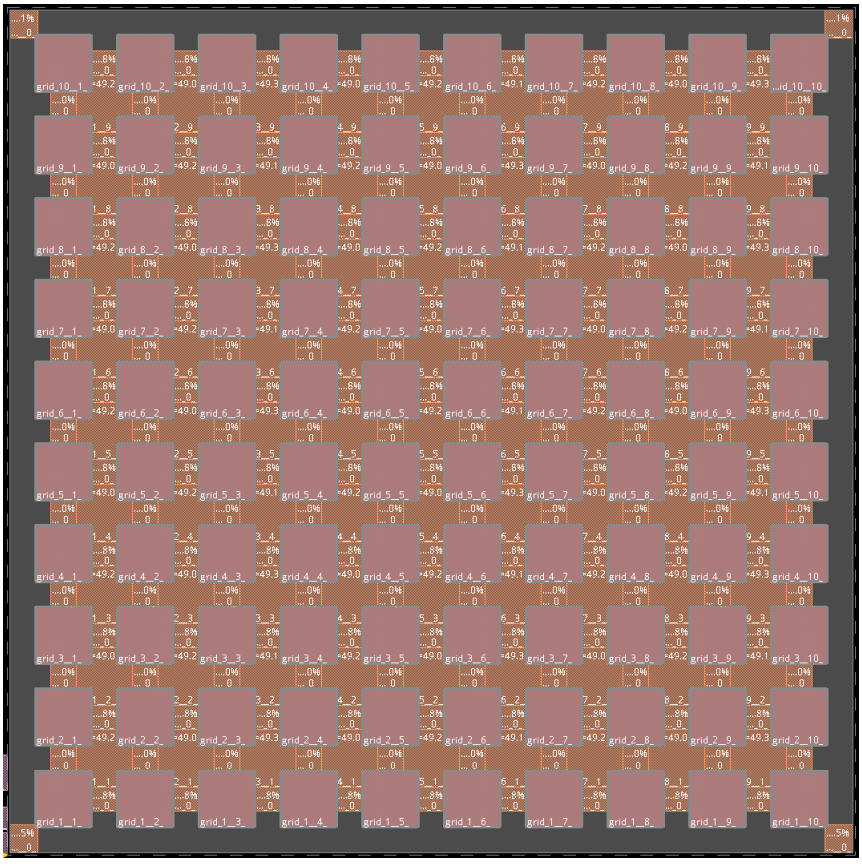
The generated Verilog code can be used through a semi-custom design flow to generate the layout.

Because of the commercial nature of the semi-custom design tools we are using, we cannot share the different scripts that we are using. However, we can show the results to serve as a proof-of-concept and encourage research through it.



Layout_Diagram shows the different steps involved in realizing the layout for any design. CTS stands for Clock Tree Synthesis, and PPA stands for Power-Performance-Area. First, we create the floorplan with the different tiles involved in the FPGA, i.e., the CLBs and place them. Then the clock tree is generated. Finally, the design is routed, and the PPA signoff is realized. Coupled with FPGA-SPICE, we get silicon level analysis on the design.

In Layout_Floorplan, we show the result we get from the floorplanning we get through Cadence Innovus.



CHAPTER

FIVE

ARCHITECTURE MODELING

5.1 A Quick Start

In this tutorial, we will consider a simple but representative FPGA architecture to show you how to

- Adapt a VPR architecture XML file to OpenFPGA acceptable format
- Create an OpenFPGA architecture XML file to customize the primitive circuits
- Create a simulation setting XML file to specify the simulation settings

Through this quick example, we will introduce the key steps to build your own FPGA based on a VPR architecture template.

Note: These tips are generic and fundamental to build any architecture file for OpenFPGA.

5.1.1 Adapt VPR Architecture

We start with the VPR architecture template. This file models a homogeneous FPGA, as illustrated in Fig. 5.1.

A summary of the architectural features is as follows:

- An array of tiles surrounded by a ring of I/O blocks
- K4N4 Configurable Logic Block (CLB), which consists of four Basic Logic Elements (BLEs) and a fully-connected crossbar. Each BLE contains a 4-input Look-Up Table (LUT), a Flip-Flop (FF) and a 2:1 routing multiplexer
- Length-1 routing wires interconnected by Wilton-Style Switch Block (SB)

The VPR architecture description is designed for EDA needs mainly, which lacks the details physical modelingrequired by OpenFPGA. Here, we show a step-by-step adaption on the architecture template.

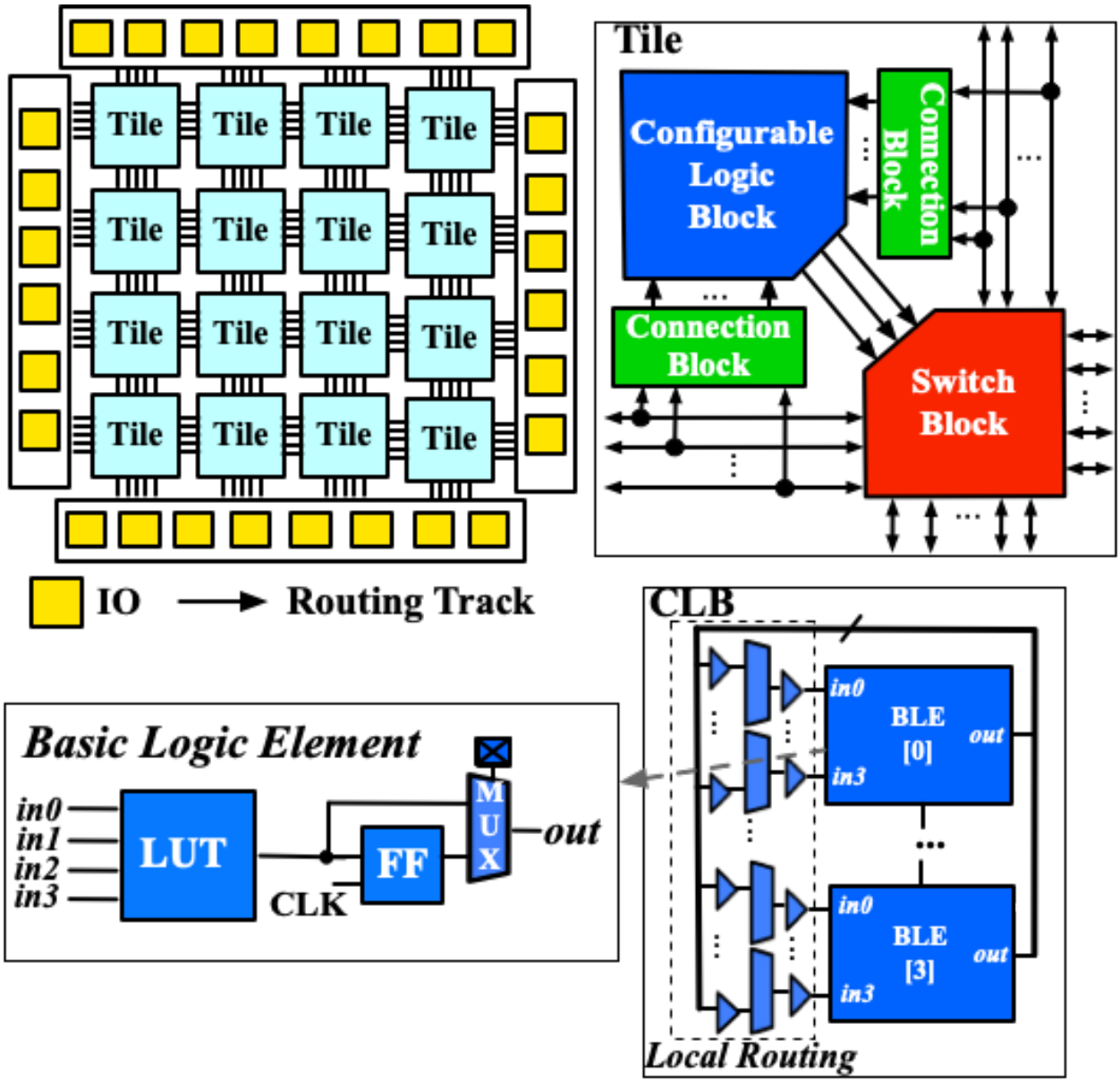


Fig. 5.1: K4N4 FPGA architecture

Physical I/O Modeling

OpenFPGA requires a physical I/O block rather the abstract I/O modeling of VPR. The <pb_type name="io"> under the <complexblocklist> should be adapted to the following:

```
<!-- Define I/O pads begin -->
<pb_type name="io">
  <input name="outpad" num_pins="1"/>
  <output name="inpad" num_pins="1"/>

  <!-- A mode denotes the physical implementation of an I/O
       This mode will not be used by packer but is mainly used for fabric verilog␣
↪generation
    -->
  <mode name="physical" packable="false">
    <pb_type name="iopad" blif_model=".subckt io" num_pb="1">
      <input name="outpad" num_pins="1"/>
      <output name="inpad" num_pins="1"/>
    </pb_type>
    <interconnect>
      <direct name="outpad" input="io.outpad" output="iopad.outpad">
        <delay_constant max="1.394e-11" in_port="io.outpad" out_port="iopad.outpad"/>
      </direct>
      <direct name="inpad" input="iopad.inpad" output="io.inpad">
        <delay_constant max="4.243e-11" in_port="iopad.inpad" out_port="io.inpad"/>
      </direct>
    </interconnect>
  </mode>

  <!-- Operating modes of I/O used by VPR
       IOs can operate as either inputs or outputs. -->
  <mode name="inpad">
    <pb_type name="inpad" blif_model=".input" num_pb="1">
      <output name="inpad" num_pins="1"/>
    </pb_type>
    <interconnect>
      <direct name="inpad" input="inpad.inpad" output="io.inpad">
        <delay_constant max="9.492000e-11" in_port="inpad.inpad" out_port="io.inpad"/>
      </direct>
    </interconnect>
  </mode>
  <mode name="outpad">
    <pb_type name="outpad" blif_model=".output" num_pb="1">
      <input name="outpad" num_pins="1"/>
    </pb_type>
    <interconnect>
      <direct name="outpad" input="io.outpad" output="outpad.outpad">
        <delay_constant max="2.675000e-11" in_port="io.outpad" out_port="outpad.outpad"/
↪>
      </direct>
    </interconnect>
  </mode>
</pb_type>
```

Note that, there are several major changes in the above codes, when compared to the original code.

- We added a physical `mode` of I/O in addition to the original VPR I/O modeling, which is close to the physical implementation of an I/O cell. OpenFPGA will output fabric netlists base on the physical implementation rather than the operating modes.
- We remove the `clock` port of I/O is actually a dangling port.
- We specify that the phyical `mode` to be disabled for VPR packer by using `packable=false`. This can help reduce packer's runtime.

Since, we have added a new BLIF model `subckt io` to the architecture modeling, we should update the `<models>` XML node by adding a new I/O model.

```
<models>
  <!-- A virtual model for I/O to be used in the physical mode of io block -->
  <model name="io">
    <input_ports>
      <port name="outpad"/>
    </input_ports>
    <output_ports>
      <port name="inpad"/>
    </output_ports>
  </model>
</models>
```

Tileable Architecture

OpenFPGA does support fine-grained tile-based architecture as shown in Fig. 5.1. The tileable architecture leads to fast netlist generation as well as enables highly optimized physical designs through backend flow. To turn on the tileable architecture, the `tileable` property should be added to `<layout>` node.

```
<layout tileable="true">
```

By enabling this, all the Switch Blocks and Connection Blocks will be generated as identical as possible. As a result, for any FPGA array size, there are only 9 unique tiles to be generated in netlists. See details in [TGAG19].

Detailed guidelines can be found at *Additional Syntax to Original VPR XML*.

5.1.2 Craft OpenFPGA Architecture

OpenFPGA needs another XML file which contains detailed modeling on the physical design of FPGA architecture. This is designed to minimize the modification on the original VPR architecture file, so that it can be reused. You may create an XML file *k4_n4_openfpga_arch.xml* and then add contents shown as follows.

Overview on the Structure

An OpenFPGA architecture including the following parts.

- Architecture modeling with a focus on circuit-level description
- Configuration protocol definition
- Annotation on the VPR architecture modules

These parts are organized as follows in the XML file.

```
<openfpga_architecture>
  <!-- Technology-related (device/transistor-level) information
  <technology_library>
    ...
  </technology_library>

  <!-- Circuit-level description -->
  <circuit_library>
    ...
  </circuit_library>

  <!-- Configuration protocol definition -->
  <configuration_protocol>
    ...
  </configuration_protocol>

  <!-- Annotation on VPR architecture modules -->
  <connection_block>
    ...
  </connection_block>
  <switch_block>
    ...
  </switch_block>
  <routing_segment>
    ...
  </routing_segment>
  <pb_type_annotations>
    ...
  </pb_type_annotations>
</openfpga_architecture>
```

Technology Library Definition

Technology information are all stored under the <technology_library> node, which contains transistor-level information to build the FPGA. Here, we bind to the open-source ASU Predictive Technology Modeling (PTM) 45nm process library. See details in *Technology library*.

```
<technology_library>
  <device_library>
    <device_model name="logic" type="transistor">
      <lib type="industry" corner="TOP_TT" ref="M" path="${OPENFPGA_PATH}/openfpga_flow/
↪tech/PTM_45nm/45nm.pm"/>
```

(continues on next page)

(continued from previous page)

```
      <design vdd="0.9" pn_ratio="2"/>
      <pmos name="pch" chan_length="40e-9" min_width="140e-9" variation="logic_
↪transistor_var"/>
      <nmos name="nch" chan_length="40e-9" min_width="140e-9" variation="logic_
↪transistor_var"/>
    </device_model>
    <device_model name="io" type="transistor">
      <lib type="academia" ref="M" path="${OPENFPGA_PATH}/openfpga_flow/tech/PTM_45nm/
↪45nm.pm"/>
      <design vdd="2.5" pn_ratio="3"/>
      <pmos name="pch_25" chan_length="270e-9" min_width="320e-9" variation="io_
↪transistor_var"/>
      <nmos name="nch_25" chan_length="270e-9" min_width="320e-9" variation="io_
↪transistor_var"/>
    </device_model>
  </device_library>
  <variation_library>
    <variation name="logic_transistor_var" abs_deviation="0.1" num_sigma="3"/>
    <variation name="io_transistor_var" abs_deviation="0.1" num_sigma="3"/>
  </variation_library>
</technology_library>
```

Note: These information are important for FPGA-SPICE to correctly generate netlists. If you are not using FPGA-SPICE, you may provide a dummy technology library.

Circuit Library Definition

Circuit library is the crucial component of the architecture description, which contains a list of <circuit_model>, each of which describes how a circuit is implemented for a FPGA component.

Typically, we will defined a few atom <circuit_model> which are used to build primitive <circuit_model>.

```
<circuit_library>
  <!-- Atom circuit models begin-->
  <circuit_model>
    ...
  </circuit_model>
  <!-- Atom circuit models end-->

  <!-- Primitive circuit models begin -->
  <circuit_model>
    ...
  </circuit_model>
  <!-- Primitive circuit models end -->
</circuit_library>
```

Note: Primitive <circuit_model> are the circuits which are directly used to build a FPGA component, such as Look-Up Table (LUT). Atom <circuit_model> are the circuits which are only used inside primitive <circuit_model>.

In this tutorial, we need the following atom <circuit_model>, which are inverters, buffers and pass-gate logics.

```
<!-- Atom circuit models begin-->
<circuit_model type="inv_buf" name="INVTX1" prefix="INVTX1" is_default="true">
  <design_technology type="cmos" topology="inverter" size="1"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <delay_matrix type="rise" in_port="in" out_port="out">
    10e-12
  </delay_matrix>
  <delay_matrix type="fall" in_port="in" out_port="out">
    10e-12
  </delay_matrix>
</circuit_model>
<circuit_model type="inv_buf" name="buf4" prefix="buf4" is_default="false">
  <design_technology type="cmos" topology="buffer" size="1" num_level="2" f_per_stage="4
↪"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <delay_matrix type="rise" in_port="in" out_port="out">
    10e-12
  </delay_matrix>
  <delay_matrix type="fall" in_port="in" out_port="out">
    10e-12
  </delay_matrix>
</circuit_model>
<circuit_model type="inv_buf" name="tap_buf4" prefix="tap_buf4" is_default="false">
  <design_technology type="cmos" topology="buffer" size="1" num_level="3" f_per_stage="4
↪"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <delay_matrix type="rise" in_port="in" out_port="out">
    10e-12
  </delay_matrix>
  <delay_matrix type="fall" in_port="in" out_port="out">
    10e-12
  </delay_matrix>
</circuit_model>
<circuit_model type="pass_gate" name="TGATE" prefix="TGATE" is_default="true">
  <design_technology type="cmos" topology="transmission_gate" nmos_size="1" pmos_size="2
↪"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="input" prefix="in" size="1"/>
  <port type="input" prefix="sel" size="1"/>
  <port type="input" prefix="selb" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <delay_matrix type="rise" in_port="in sel selb" out_port="out">
    10e-12 5e-12 5e-12
  </delay_matrix>
  <delay_matrix type="fall" in_port="in sel selb" out_port="out">
    10e-12 5e-12 5e-12
  </delay_matrix>
```

(continues on next page)

(continued from previous page)

```
</circuit_model>
<circuit_model type="chan_wire" name="chan_segment" prefix="track_seg" is_default="true">
  <design_technology type="cmos"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <wire_param model_type="pi" R="101" C="22.5e-15" num_level="1"/> <!-- model_type could␣
↪be T, res_val and cap_val DON'T CARE -->
</circuit_model>
<circuit_model type="wire" name="direct_interc" prefix="direct_interc" is_default="true">
  <design_technology type="cmos"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <wire_param model_type="pi" R="0" C="0" num_level="1"/> <!-- model_type could be T,␣
↪res_val cap_val should be defined -->
</circuit_model>
<!-- Atom circuit models end-->
```

In this tutorial, we require the following primitive <circuit_model>, which are routing multiplexers, Look-Up Tables, I/O cells in FPGA architecture (see Fig. 5.1).

Note: We use different routing multiplexer circuits to maximum the performance by considering it fan-in and fan-out in the architecture context.

Note: We specify that external Verilog netlists will be used for the circuits of Flip-Flops (FFs) `static_dff` and `sc_dff_compact`, as well as the circuit of I/O cell `iopad`. Other circuit models will be auto-generated by OpenFPGA.

```
<!-- Primitive circuit models begin -->
<circuit_model type="mux" name="mux_2level" prefix="mux_2level" dump_structural_verilog=
↪"true">
  <design_technology type="cmos" structure="multi_level" num_level="2" add_const_input=
↪"true" const_input_val="1"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
  <output_buffer exist="true" circuit_model_name="INVTX1"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="1"/>
</circuit_model>
<circuit_model type="mux" name="mux_2level_tapbuf" prefix="mux_2level_tapbuf" dump_
↪structural_verilog="true">
  <design_technology type="cmos" structure="multi_level" num_level="2" add_const_input=
↪"true" const_input_val="1"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
  <output_buffer exist="true" circuit_model_name="tap_buf4"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
```

(continues on next page)

(continued from previous page)

```
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="1"/>
</circuit_model>
<circuit_model type="mux" name="mux_1level_tapbuf" prefix="mux_1level_tapbuf" is_default=
↪"true" dump_structural_verilog="true">
  <design_technology type="cmos" structure="one_level" add_const_input="true" const_
↪input_val="1"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
  <output_buffer exist="true" circuit_model_name="tap_buf4"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="1"/>
</circuit_model>
<!--DFF subckt ports should be defined as <D> <Q> <CLK> <RESET> <SET>  -->
<circuit_model type="ff" name="static_dff" prefix="dff" spice_netlist="${OPENFPGA_PATH}/
↪openfpga_flow/SpiceNetlists/ff.sp" verilog_netlist="${OPENFPGA_PATH}/openfpga_flow/
↪VerilogNetlists/ff.v">
   <design_technology type="cmos"/>
   <input_buffer exist="true" circuit_model_name="INVTX1"/>
   <output_buffer exist="true" circuit_model_name="INVTX1"/>
   <port type="input" prefix="D" size="1"/>
   <port type="input" prefix="set" size="1" is_global="true" default_val="0" is_set="true
↪"/>
   <port type="input" prefix="reset" size="1" is_global="true" default_val="0" is_reset=
↪"true"/>
   <port type="output" prefix="Q" size="1"/>
   <port type="clock" prefix="clk" size="1" is_global="true" default_val="0" />
</circuit_model>
<circuit_model type="lut" name="lut4" prefix="lut4" dump_structural_verilog="true">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
  <output_buffer exist="true" circuit_model_name="INVTX1"/>
  <lut_input_inverter exist="true" circuit_model_name="INVTX1"/>
  <lut_input_buffer exist="true" circuit_model_name="buf4"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="4"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="16"/>
</circuit_model>
<!--Scan-chain DFF subckt ports should be defined as <D> <Q> <Qb> <CLK> <RESET> <SET>  --
↪>
<circuit_model type="ccff" name="sc_dff_compact" prefix="scff" spice_netlist="${OPENFPGA_
↪PATH}/openfpga_flow/SpiceNetlists/ff.sp" verilog_netlist="${OPENFPGA_PATH}/openfpga_
↪flow/VerilogNetlists/ff.v">
   <design_technology type="cmos"/>
   <input_buffer exist="true" circuit_model_name="INVTX1"/>
   <output_buffer exist="true" circuit_model_name="INVTX1"/>
   <port type="input" prefix="pReset" lib_name="reset" size="1" is_global="true" default_
↪val="0" is_reset="true" is_prog="true"/>
   <port type="input" prefix="D" size="1"/>
```

(continues on next page)

(continued from previous page)

```
    <port type="output" prefix="Q" size="1"/>
    <port type="output" prefix="Qb" size="1"/>
    <port type="clock" prefix="prog_clk" lib_name="clk" size="1" is_global="true" default_
↪val="0" is_prog="true"/>
</circuit_model>
<circuit_model type="iopad" name="iopad" prefix="iopad" spice_netlist="${OPENFPGA_PATH}/
↪openfpga_flow/SpiceNetlists/io.sp" verilog_netlist="${OPENFPGA_PATH}/openfpga_flow/
↪VerilogNetlists/io.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
  <output_buffer exist="true" circuit_model_name="INVTX1"/>
  <port type="inout" prefix="pad" size="1" is_global="true" is_io="true"/>
  <port type="sram" prefix="en" size="1" mode_select="true" circuit_model_name="sc_dff_
↪compact" default_val="1"/>
  <port type="input" prefix="outpad" size="1"/>
  <port type="output" prefix="inpad" size="1"/>
</circuit_model>
<!-- Primitive circuit models end -->
```

See details in *Circuit Library* and *Circuit model examples*.

Annotation on VPR Architecture

In this part, we bind the <circuit_model> defined in the circuit library to each FPGA component.

We specify that the FPGA fabric will be configured through a chain of Flip-Flops (FFs), which is built with the <circuit_model name=sc_dff_compact>.

```
<configuration_protocol>
  <organization type="scan_chain" circuit_model_name="sc_dff_compact"/>
</configuration_protocol>
```

For the routing architecture, we specify the circuit_model to be used as routing multiplexers inside Connection Blocks (CBs), Switch Blocks (SBs), and also the routing wires.

```
<connection_block>
  <switch name="ipin_cblock" circuit_model_name="mux_2level_tapbuf"/>
</connection_block>
<switch_block>
  <switch name="0" circuit_model_name="mux_2level_tapbuf"/>
</switch_block>
<routing_segment>
  <segment name="L4" circuit_model_name="chan_segment"/>
</routing_segment>
```

Note: For a correct binding, the name of connection block, switch block and routing segment should match the name definition in your VPR architecture description!

For each <pb_type> defined in the <complexblocklist> of VPR architecture, we need to specify

- The physical mode for any <pb_type> that contains multiple <mode>. The name of the physical mode should match a mode name that is defined in the VPR architecture. For example:

```
<pb_type name="io" physical_mode_name="physical"/>
```

- The circuit model used to implement any primitive <pb_type> in physical modes. It is required to provide full hierarchy of the pb_type. For example:

```
<pb_type name="io[physical].iopad" circuit_model_name="iopad" mode_bits="1"/>
```

Note: Mode-selection bits should be provided as the default configuration for a configurable resource. In this example, an I/O cell has a configuration bit, as defined in the <circuit_model name="iopad">. We specify that by default, the configuration memory will be set to logic 1.

- The physical <pb_type> for any <pb_type> in the operating modes (mode other than the physical mode). This is required to translate mapping results from operating modes to their physical modes, in order to generate bitstreams. It is required to provide full hierarchy of the pb_type. For example,

```
<pb_type name="io[inpad].inpad" physical_pb_type_name="io[physical].iopad" mode_bits="1"/
↪>
```

Note: Mode-selection bits should be provided so as to configure the circuits to be functional as required by the operating mode. In this example, an I/O cell will be configured with a logic 1 when operating as an input pad.

- The circuit model used to implement interconnecting modules. The interconnect name should match the definition in the VPR architecture file. For example,

```
<interconnect name="crossbar" circuit_model_name="mux_2level"/>
```

Note: If not specified, each interconnect will be binded to its default circuit_model. For example, the crossbar will be binded to the default multiplexer <circuit_model name="mux_1level_tapbuf">, if not specified here.

Note: OpenFPGA automatically infers the type of circuit model required by each interconnect.

The complete annotation is shown as follows:

```
<pb_type_annotations>
  <!-- physical pb_type binding in complex block IO -->
  <pb_type name="io" physical_mode_name="physical"/>
  <pb_type name="io[physical].iopad" circuit_model_name="iopad" mode_bits="1"/>
  <pb_type name="io[inpad].inpad" physical_pb_type_name="io[physical].iopad" mode_bits="1
↪"/>
  <pb_type name="io[outpad].outpad" physical_pb_type_name="io[physical].iopad" mode_bits=
↪"0"/>
  <!-- End physical pb_type binding in complex block IO -->

  <!-- physical pb_type binding in complex block CLB -->
  <!-- physical mode will be the default mode if not specified -->
  <pb_type name="clb">
    <!-- Binding interconnect to circuit models as their physical implementation, if not␣
```

(continues on next page)

(continued from previous page)

```
↪defined, we use the default model -->
    <interconnect name="crossbar" circuit_model_name="mux_2level"/>
  </pb_type>
  <pb_type name="clb.fle[n1_lut4].ble4.lut4" circuit_model_name="lut4"/>
  <pb_type name="clb.fle[n1_lut4].ble4.ff" circuit_model_name="static_dff"/>
  <!-- End physical pb_type binding in complex block IO -->
</pb_type_annotations>
```

See details in *Bind circuit modules to VPR architecture*.

5.1.3 Simulation Settings

OpenFPGA needs an XML file where detailed simulation settings are defined. The simulation settings contain critical parameters to build testbenches for verify the FPGA fabric.

You may create an XML file *k4_n4_openfpga_simulation.xml* and then add contents shown as follows.

The complete annotation is shown as follows:

```
<openfpga_simulation_setting>
  <clock_setting>
    <operating frequency="auto" num_cycles="auto" slack="0.2"/>
    <programming frequency="100e6"/>
  </clock_setting>
  <simulator_option>
    <operating_condition temperature="25"/>
    <output_log verbose="false" captab="false"/>
    <accuracy type="abs" value="1e-13"/>
    <runtime fast_simulation="true"/>
  </simulator_option>
  <monte_carlo num_simulation_points="2"/>
  <measurement_setting>
    <slew>
      <rise upper_thres_pct="0.95" lower_thres_pct="0.05"/>
      <fall upper_thres_pct="0.05" lower_thres_pct="0.95"/>
    </slew>
    <delay>
      <rise input_thres_pct="0.5" output_thres_pct="0.5"/>
      <fall input_thres_pct="0.5" output_thres_pct="0.5"/>
    </delay>
  </measurement_setting>
  <stimulus>
    <clock>
      <rise slew_type="abs" slew_time="20e-12" />
      <fall slew_type="abs" slew_time="20e-12" />
    </clock>
    <input>
      <rise slew_type="abs" slew_time="25e-12" />
      <fall slew_type="abs" slew_time="25e-12" />
    </input>
  </stimulus>
</openfpga_simulation_setting>
```

The <clock_setting> is crucial to create clock signals in testbenches.

Note: FPGA has two types of clocks, one is the operating clock which controls applications that mapped to FPGA fabric, while the other is the programming clock which controls the configuration protocol.

In this example, we specify

- the operating clock will follow the maximum frequency achieved by VPR routing results
- the number of operating clock cycles to be used will follow the average signal activities of the RTL design that is mapped to the FPGA fabric.
- the actual operating clock frequency will be relaxed (reduced) by 20% by considering the errors between VPR results and physical designs.
- the programming clock frequency is fixed at 200MHz

The <simulator_option> are the options for SPICE simulator. Here we specify

- SPICE simulations will consider a 25 $^{\circ}C$ temperature.
- SPICE simulation will output results in a compact way without details on node capacitances.
- SPICE simulation will use `0.1ps` as the minimum time step.
- SPICE simulation will consider fast algorithms to speed up runtime.

The <monte_carlo num_simulation_points="2"/> are the options for SPICE simulator. Here we specify that for each testbench, we will consider two Monte-Carlo simulations to evaluate the impact of process variations.

The <measurement_setting> specify how the output signals will be measured for delay and power evaluation. Here we specify that

- for slew calculation (used in power estimation), we consider from the 5% of the `VDD` to the 95% of the `VDD` for both rising and falling edges.
- for delay calculation, we consider from the 50% of the `VDD` of input signal to the 50% of the `VDD` of output signals for both rising and falling edges.

In the <stimulus>, we specify that `20ps` slew time will be applied to built clock waverforms in SPICE simulations.

See details in *Simulation settings*.

5.2 Integrating Custom Verilog Modules with user_defined_template.v

5.2.1 Introduction and Setup

In this tutorial, we will

- Provide the motivation for generating the user_defined_template.v verilog file
- Go through a generated user_defined_template.v file to demonstrate how to use it

Through this tutorial, we will show how and when to use the *user_defined_template.v* file.

To begin the tutorial, we start with a modified version of the hard adder task that comes with OpenFPGA. To follow along, go to the root directory of OpenFPGA and enter:

```
vi openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml
```

Go to **LINE187** and replace **LINE187** with:

```
<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="">
```

5.2.2 Motivation

From the OpenFPGA root directory, run the command:

```
python3 openfpga_flow/scripts_run_fpga_task.py fpga_verilog/adder/hard_adder --debug --
↪show_thread_logs
```

Running this command should fail and produce the following errors:

```
ERROR - iverilog_verification run failed with returncode 21
ERROR - command iverilog -o compiled_and2 ./SRC/and2_include_netlists.v -s and2_top_
↪formal_verification_random_tb
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>././SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
```

(continues on next page)

(continued from previous page)

```
ERROR - -->>./SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>./SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>./SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>./SRC/lb/logical_tile_clb_mode_default__fle_mode_physical__fabric_mode_
↪default__adder.v:50: error: Unknown module type: ADDF
ERROR - -->>21 error(s) during elaboration.
ERROR - Current working directory :    /research/ece/lnis/USERS/leaptrot/OpenFPGA/
↪openfpga_flow/tasks/fpga_verilog/adder/hard_adder/run019/k6_frac_N10_tileable_adder_
↪chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH
ERROR - Failed to run iverilog_verification task
ERROR - Exiting . . . . . .
```

This error log can also be found by running the following command from the root directory:

```
cat openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/00_and2_MIN_ROUTE_CHAN_
↪WIDTH_out.log
```

This command failed during the verification step because the path to the module definition for **ADDF** is missing. In our architecture file, user-defined verilog modules are those <circuit_model> with the key term *verilog_netlist*. The user_defined_template.v file provides a module template for incorporating Hard IPs without external library into the architecture.

5.2.3 Fixing the Error

This error can be resolved by replacing the **LINE187** of k6_frac_N10_adder_chain_40nm_openfpga.xml with the following:

```
<circuit_model type="hard_logic" name="ADDF" prefix="ADDF" is_default="true" spice_
↪netlist="${OPENFPGA_PATH}/openfpga_flow/openfpga_cell_library/spice/adder.sp"    ␣
↪verilog_netlist="${OPENFPGA_PATH}/openfpga_flow/openfpga_cell_library/verilog/adder.v">
```

The above line provides a path to generate the *user_defined_template.v* file. Now we can return to the root directory and run this command again:

```
python3 openfpga_flow/scripts_run_fpga_task.py fpga_verilog/adder/hard_adder --debug --
↪show_thread_logs
```

The task should now complete without any errors.

5.2.4 Fixing the Error with user_defined_template.v

The *user_defined_template.v* file can be found starting from the root directory and entering:

```
vi openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_adder_
↪chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/sub_module/user_defined_template.v
```

Note: The user_defined_template.v file contains user-defined verilog modules that are found in the openfpga_cell_library with ports declaration (compatible with other netlists that are auto-generated by OpenFPGA) but

without functionality. `user_defined_template.v` is used as a reference for engineers to check what is the port sequence required by top-level verilog netlists. `user_defined_template.v` can be included in simulation only if there are modifications to the `user_defined_template.v`.

To implement our own **ADDF** module, we need to remove all other module definitions (they are already defined elsewhere and will cause an error if left in). Replace the `user_defined_template.v` file with the following:

```
//-----------------------------------------
//      FPGA Synthesizable Verilog Netlist
//      Description: Template for user-defined Verilog modules
//      Author: Xifan TANG
//      Organization: University of Utah
//      Date: Fri Mar 19 10:05:32 2021
//-----------------------------------------
//----- Time scale -----
`timescale 1ns / 1ps



// ----- Template Verilog module for ADDF -----
//----- Default net type -----
`default_nettype none

// ----- Verilog module for ADDF -----
module ADDF(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
// ----- END Verilog module for ADDF -----
```

We can now link this `user_defined_template.v` into `k6_frac_N10_adder_chain_40nm_openfpga.xml`.

Note: Be sure to select the run where you modified the `user_defined_template.v`!

From the OpenFPGA root directory, run:

```
vi openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml
```

At **LINE187** in verilog_netlist, put in:

```
${OPENFPGA_PATH}/openfpga_flow/tasks/fpga_verilog/adder/hard_adder/**YOUR_RUN_NUMBER**/
↪k6_frac_N10_tileable_adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/sub_module/user_
↪defined_template.v
```

Finally, rerun this command from the OpenFPGA root directory to ensure it is working:

```
python3 openfpga_flow/scripts_run_fpga_task.py fpga_verilog/adder/hard_adder --debug --
↪show_thread_logs
```

5.3 Build an FPGA fabric using Standard Cell Libraries

5.3.1 Introduction

In this tutorial, we will

- Showcase how to create an architecture description based on standard cells, using OpenFPGA's circuit modeling language
- Use Skywater's Process Design Kit (PDK) cell library to create an OR Gate circuit model for OpenFPGA
- Verify that the standard cell library file was correctly bound into the selected architecture file by looking at auto-generated OpenFPGA files and checking simulation waveforms in GTKWave

Through this example, we will show how to bind standard cell library files with OpenFPGA Architectures.

Note: We showcase the methodology by considering the open-source Skywater 130nm PDK so that users can easily reproduce the results.

5.3.2 Create and Verify the OpenFPGA Circuit Model

Note: In this tutorial, we focus on binding a 2-input **OR** gate from a standard cell library to a circuit model in OpenFPGA's architecture description file. Note that the approach can be generalized to any circuit model.

For this tutorial, we start with an example where the HDL netlist of an 2-input **OR** gate that is auto-generated by OpenFPGA. After updating the architecture file, the auto-generated HDL netlist created by OpenFPGA will directly instantiate a standard cell from the open-source Skywater 130nm PDK library. To follow along, go to the root directory of OpenFPGA and enter:

```
python3 openfpga_flow/scripts_run_fpga_task.py fpga_verilog/adder/hard_adder --debug --
↪show_thread_logs
```

This will run a prebuilt task with OpenFPGA cell libraries. When the task is finished, there will be many auto-generated files to look through. For this tutorial, we are interested in the `luts.v` and `and2_formal.vcd` files. The **OR2** gate is used as a control circuit in the **lut6** circuit model, and the `and2_formal.vcd` file will have the resulting waveforms from the simulation run by the task. To open the `luts.v` file, run the following command:

```
vi 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
```

Note: Users can find full details about netlist organization in our documentation: *Fabric Netlists*

The `luts.v` file represents a Look Up Table within the OpenFPGA architecture. The important lines of this file for the tutorial are highlighted below. These lines show the instantiation of OpenFPGA's **OR2** cell library.

```
//-----------------------------------------
//   FPGA Synthesizable Verilog Netlist
//   Description: Look-Up Tables
//   Author: Xifan TANG
//   Organization: University of Utah
//   Date: Tue Mar 30 15:25:03 2021
//-----------------------------------------
//----- Time scale -----
`timescale 1ns / 1ps

//----- Default net type -----
`default_nettype none

// ----- Verilog module for frac_lut6 -----
module frac_lut6(in,
                 sram,
                 sram_inv,
                 mode,
                 mode_inv,
                 lut4_out,
                 lut5_out,
                 lut6_out);
//----- INPUT PORTS -----
input [0:5] in;
//----- INPUT PORTS -----
input [0:63] sram;
//----- INPUT PORTS -----
input [0:63] sram_inv;
//----- INPUT PORTS -----
input [0:1] mode;
//----- INPUT PORTS -----
input [0:1] mode_inv;
//----- OUTPUT PORTS -----
output [0:3] lut4_out;
//----- OUTPUT PORTS -----
output [0:1] lut5_out;
```

(continues on next page)

(continued from previous page)

```
//----- OUTPUT PORTS -----
output [0:0] lut6_out;

//----- BEGIN wire-connection ports -----
wire [0:5] in;
wire [0:3] lut4_out;
wire [0:1] lut5_out;
wire [0:0] lut6_out;
//----- END wire-connection ports -----


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


wire [0:0] INVTX1_0_out;
wire [0:0] INVTX1_1_out;
wire [0:0] INVTX1_2_out;
wire [0:0] INVTX1_3_out;
wire [0:0] INVTX1_4_out;
wire [0:0] INVTX1_5_out;
wire [0:0] OR2_0_out;
wire [0:0] OR2_1_out;
wire [0:0] buf4_0_out;
wire [0:0] buf4_1_out;
wire [0:0] buf4_2_out;
wire [0:0] buf4_3_out;
wire [0:0] buf4_4_out;
wire [0:0] buf4_5_out;

// ----- BEGIN Local short connections -----
// ----- END Local short connections -----
// ----- BEGIN Local output short connections -----
// ----- END Local output short connections -----

     OR2 OR2_0_ (
             .a(mode[0:0]),
             .b(in[4]),
             .out(OR2_0_out));

     OR2 OR2_1_ (
             .a(mode[1]),
             .b(in[5]),
             .out(OR2_1_out));

     INVTX1 INVTX1_0_ (
             .in(in[0:0]),
             .out(INVTX1_0_out));

     INVTX1 INVTX1_1_ (
             .in(in[1]),
             .out(INVTX1_1_out));
```

(continues on next page)

(continued from previous page)

```
    INVTX1 INVTX1_2_ (
            .in(in[2]),
            .out(INVTX1_2_out));

    INVTX1 INVTX1_3_ (
            .in(in[3]),
            .out(INVTX1_3_out));

    INVTX1 INVTX1_4_ (
            .in(OR2_0_out),
            .out(INVTX1_4_out));

    INVTX1 INVTX1_5_ (
            .in(OR2_1_out),
            .out(INVTX1_5_out));

    buf4 buf4_0_ (
            .in(in[0:0]),
            .out(buf4_0_out));

    buf4 buf4_1_ (
            .in(in[1]),
            .out(buf4_1_out));

    buf4 buf4_2_ (
            .in(in[2]),
            .out(buf4_2_out));

    buf4 buf4_3_ (
            .in(in[3]),
            .out(buf4_3_out));

    buf4 buf4_4_ (
            .in(OR2_0_out),
            .out(buf4_4_out));

    buf4 buf4_5_ (
            .in(OR2_1_out),
            .out(buf4_5_out));

    frac_lut6_mux frac_lut6_mux_0_ (
            .in(sram[0:63]),
            .sram({buf4_0_out, buf4_1_out, buf4_2_out, buf4_3_out, buf4_4_out, buf4_5_
↪out}),
            .sram_inv({INVTX1_0_out, INVTX1_1_out, INVTX1_2_out, INVTX1_3_out, INVTX1_
↪4_out, INVTX1_5_out}),
            .lut4_out(lut4_out[0:3]),
            .lut5_out(lut5_out[0:1]),
            .lut6_out(lut6_out));

endmodule
```

(continues on next page)

(continued from previous page)

```
// ----- END Verilog module for frac_lut6 -----

//----- Default net type -----
`default_nettype none
```

We will also need to look at the control's simulation waveforms. Viewing the waveforms is done through GTKWave with the following command:

```
gtkwave openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_
↪adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/and2_formal.vcd &
```

The simulation waveforms should look similar to the following Fig. 5.2:

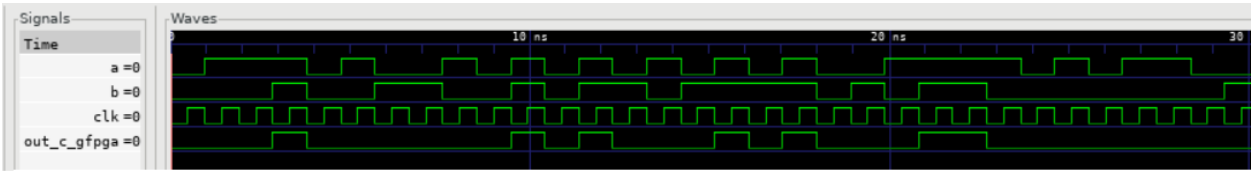


Fig. 5.2: Simulation Waveforms with OpenFPGA Circuit Model

Note: The waveform inputs do not need to exactly match because the testbench provides input in random intervals.

We have now finished creating the control and viewing the important sections for this tutorial. We can now incorporate Skywater's cell library to create a new circuit model.

5.3.3 Clone Skywater PDK into OpenFPGA

We will be using the open-source Skywater PDK to create our circuit model. We start by cloning the Skywater PDK github repository into the OpenFPGA root directory. Run the following command in the root directory of OpenFPGA:

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

Once the repository has been cloned, we need to build the cell libraries by running the following command in the Skywater PDK root directory:

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

This will take some time to complete due to the size of the libraries. Once the libraries are made, creating the circuit model can begin.

5.3.4 Create and Verify the Standard Cell Library Circuit Model

To create the circuit model, we will modify the `k6_frac_N10_adder_chain_40nm_openfpga.xml` OpenFPGA architecture file by removing the circuit model for OpenFPGA's **OR2** gate, replacing the circuit model with one referencing the Skywater cell library, and modifying the LUT that references the old **OR2** circuit model to reference our new circuit model. We begin by running the following command in the root directory:

```
vi openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml
```

We continue the circuit model creation process by replacing **LINE67** to **LINE81** with the following:

```
<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>
```

Note: The name of the circuit model must be consistent with the standard cell!

The most significant differences from the OpenFPGA Circuit Model in this section are:

- Change the `name` and `prefix` to match the module name from Skywater's cell library
- Include a path to the verilog file using `verilog_netlist`.

The second change to `k6_frac_N10_adder_chain_40nm_openfpga.xml` is at **LINE160**, where we will be replacing the line with the following:

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

This change replaces the input of the LUT with our new circuit model. Everything is in place to begin verification.

Verification begins by running the following command:

```
python3 openfpga_flow/scripts_run_fpga_task.py fpga_verilog/adder/hard_adder --debug --
↪show_thread_logs
```

The task may output this error:

```
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - iverilog_verification run failed with returncode 1
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - command iverilog -o compiled_and2 ./SRC/and2_
↪include_netlists.v -s and2_top_formal_verification_random_tb
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - -->>error: Unable to find the root module "and2_
↪top_formal_verification_random_tb" in the Verilog source.
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - -->>1 error(s) during elaboration.
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - Current working directory : OpenFPGA/openfpga_
↪flow/tasks/fpga_verilog/adder/hard_adder/run057/k6_frac_N10_tileable_adder_chain_40nm/
↪and2/MIN_ROUTE_CHAN_WIDTH
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - Failed to run iverilog_verification task
```

(continues on next page)

(continued from previous page)

```
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - Exiting . . . . . .
ERROR (00_and2_MIN_ROUTE_CHAN_WIDTH) - Failed to execute openfpga flow - 00_and2_MIN_
↪ROUTE_CHAN_WIDTH
```

This error has occurred because IVerilog could not find the path to the Skywater PDK Cell Library we have selected. To fix this, we need to go to the `iverilog_output.txt` file found here:

```
emacs openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_
↪adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/iverilog_output.txt
```

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

```
iverilog -o compiled_and2 ./SRC/and2_include_netlists.v -s and2_top_formal_verification_
↪random_tb -I ${OPENFPGA_PATH}/skywater-pdk/libraries/sky130_fd_sc_ls/latest/cells/or2
```

We can now manually rerun IVerilog, a tutorial on manually running IVerilog can be found at our *From Verilog to Verification* tutorial. From the root directory, run the following commands:

```
cd openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_adder_
↪chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/

source iverilog_output.txt

vvp compiled_and2
```

With IVerilog complete, we can verify that the cell library has been bound correctly by viewing the `luts.v` file and the waveforms with GTKWave.

From the root directory, view the `luts.v` file with this command:

```
vi 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
```

Scrolling through `luts.v`, this should be present in the file:

```
//-------------------------------------------
//   FPGA Synthesizable Verilog Netlist
//   Description: Look-Up Tables
//   Author: Xifan TANG
//   Organization: University of Utah
//   Date: Tue Mar 30 20:25:06 2021
//-------------------------------------------
//----- Time scale -----
`timescale 1ns / 1ps

//----- Default net type -----
`default_nettype none

// ----- Verilog module for frac_lut6 -----
module frac_lut6(in,
                 sram,
                 sram_inv,
                 mode,
```

(continues on next page)

(continued from previous page)

```
                mode_inv,
                lut4_out,
                lut5_out,
                lut6_out);
//----- INPUT PORTS -----
input [0:5] in;
//----- INPUT PORTS -----
input [0:63] sram;
//----- INPUT PORTS -----
input [0:63] sram_inv;
//----- INPUT PORTS -----
input [0:1] mode;
//----- INPUT PORTS -----
input [0:1] mode_inv;
//----- OUTPUT PORTS -----
output [0:3] lut4_out;
//----- OUTPUT PORTS -----
output [0:1] lut5_out;
//----- OUTPUT PORTS -----
output [0:0] lut6_out;

//----- BEGIN wire-connection ports -----
wire [0:5] in;
wire [0:3] lut4_out;
wire [0:1] lut5_out;
wire [0:0] lut6_out;
//----- END wire-connection ports -----


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


wire [0:0] INVTX1_0_out;
wire [0:0] INVTX1_1_out;
wire [0:0] INVTX1_2_out;
wire [0:0] INVTX1_3_out;
wire [0:0] INVTX1_4_out;
wire [0:0] INVTX1_5_out;
wire [0:0] buf4_0_out;
wire [0:0] buf4_1_out;
wire [0:0] buf4_2_out;
wire [0:0] buf4_3_out;
wire [0:0] buf4_4_out;
wire [0:0] buf4_5_out;
wire [0:0] sky130_fd_sc_ls__or2_1_0_X;
wire [0:0] sky130_fd_sc_ls__or2_1_1_X;

// ----- BEGIN Local short connections -----
// ----- END Local short connections -----
// ----- BEGIN Local output short connections -----
// ----- END Local output short connections -----
```

(continues on next page)

(continued from previous page)

```
    sky130_fd_sc_ls__or2_1 sky130_fd_sc_ls__or2_1_0_ (
            .A(mode[0:0]),
            .B(in[4]),
            .X(sky130_fd_sc_ls__or2_1_0_X));

    sky130_fd_sc_ls__or2_1 sky130_fd_sc_ls__or2_1_1_ (
            .A(mode[1]),
            .B(in[5]),
            .X(sky130_fd_sc_ls__or2_1_1_X));

    INVTX1 INVTX1_0_ (
            .in(in[0:0]),
            .out(INVTX1_0_out));

    INVTX1 INVTX1_1_ (
            .in(in[1]),
            .out(INVTX1_1_out));

    INVTX1 INVTX1_2_ (
            .in(in[2]),
            .out(INVTX1_2_out));

    INVTX1 INVTX1_3_ (
            .in(in[3]),
            .out(INVTX1_3_out));

    INVTX1 INVTX1_4_ (
            .in(sky130_fd_sc_ls__or2_1_0_X),
            .out(INVTX1_4_out));

    INVTX1 INVTX1_5_ (
            .in(sky130_fd_sc_ls__or2_1_1_X),
            .out(INVTX1_5_out));

    buf4 buf4_0_ (
            .in(in[0:0]),
            .out(buf4_0_out));

    buf4 buf4_1_ (
            .in(in[1]),
            .out(buf4_1_out));

    buf4 buf4_2_ (
            .in(in[2]),
            .out(buf4_2_out));

    buf4 buf4_3_ (
            .in(in[3]),
            .out(buf4_3_out));

    buf4 buf4_4_ (
```

(continues on next page)

(continued from previous page)

```
            .in(sky130_fd_sc_ls__or2_1_0_X),
            .out(buf4_4_out));

    buf4 buf4_5_ (
            .in(sky130_fd_sc_ls__or2_1_1_X),
            .out(buf4_5_out));

    frac_lut6_mux frac_lut6_mux_0_ (
            .in(sram[0:63]),
            .sram({buf4_0_out, buf4_1_out, buf4_2_out, buf4_3_out, buf4_4_out, buf4_5_
↪out}),
            .sram_inv({INVTX1_0_out, INVTX1_1_out, INVTX1_2_out, INVTX1_3_out, INVTX1_
↪4_out, INVTX1_5_out}),
            .lut4_out(lut4_out[0:3]),
            .lut5_out(lut5_out[0:1]),
            .lut6_out(lut6_out));

endmodule
// ----- END Verilog module for frac_lut6 -----

//----- Default net type -----
`default_nettype none
```

We can check the waveforms as well to see if they are similar with the command:

```
gtkwave openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_
↪adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/and2_formal.vcd &
```

The simulation waveforms should look similar to the following Fig. 5.3:

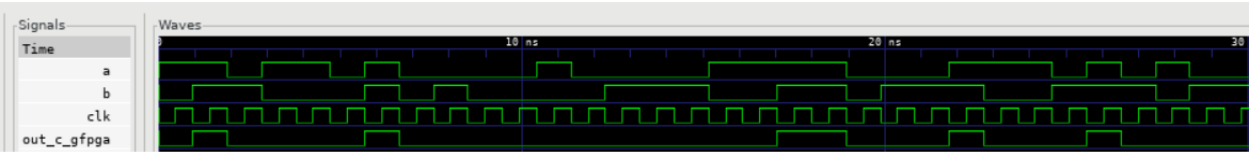


Fig. 5.3: Simulation Waveforms with Skywater PDK Circuit Model

We have now verified that the Skywater PDK Cell Library has been instantiated and bound to the OpenFPGA architecture file. If you have any problems, please *Contact* us.

5.4 Creating Spypads Using XML Syntax

5.4.1 Introduction

In this tutorial, we will

- Show the XML syntax for global outputs
- Showcase an example with spypads
- Modify an existing architecture to incorporate spypads

- Verify correctness through GTKWave

Through this tutorial, we will show how to create spypads in OpenFPGA.

Spypads are physical output pins on a FPGA chip through which you can read out internal signals when doing silicon-level debugging. The XML syntax for spypads and other global signals can be found on our *Circuit Library* documentation page.

To create a spypad, the `port type` needs to be set to **output** and `is_global` and `is_io` need to be set to **true**:

```
<port type="output" is_global="true" is_io="true"/>
```

When the port is syntactically correct, the outputs are independently wired from different instances to separated FPGA outputs and would physically look like *General-purpose outputs as separated FPGA I/Os*

5.4.2 Pre-Built Spypads

An OpenFPGA architecture file that contains spypads and has a task that references it is the k6_frac_N10_adder_register_scan_chain_depop50_spypad_40nm_openfpga.xml file. We can view `k6_frac_N10_adder_register_scan_chain_depop50_spypad_40nm_openfpga.xml` by entering the following command at the root directory of OpenFPGA:

```
emacs openfpga_flow/openfpga_arch/k6_frac_N10_adder_register_scan_chain_depop50_spypad_
↪40nm_openfpga.xml
```

In this architecture file, the output ports of a 6-input Look-Up Table (LUT) are defined as spypads using the XML syntax `is_global` and `is_io`. As a result, all of the outputs from the 6-input LUT will be visible in the top-level module. The output ports to the 6-input LUT are declared from **LINE181** to **LINE183** and belong to the `frac_lut6_spypad` `circuit_model` that begins at **LINE172**.

```
<circuit_model type="lut" name="frac_lut6_spypad" prefix="frac_lut6_spypad" dump_
↪structural_verilog="true">
  <design_technology type="cmos" fracturable_lut="true"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
  <output_buffer exist="true" circuit_model_name="INVTX1"/>
  <lut_input_inverter exist="true" circuit_model_name="INVTX1"/>
  <lut_input_buffer exist="true" circuit_model_name="buf4"/>
  <lut_intermediate_buffer exist="true" circuit_model_name="buf4" location_map="-1-1-"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="6" tri_state_map="----11" circuit_model_name="OR2
↪"/>
  LINE181 <port type="output" prefix="lut4_out" size="4" lut_frac_level="4" lut_output_
↪mask="0,1,2,3" is_global="true" is_io="true"/>
  LINE182 <port type="output" prefix="lut5_out" size="2" lut_frac_level="5" lut_output_
↪mask="0,1" is_global="true" is_io="true"/>
  LINE183 <port type="output" prefix="lut6_out" size="1" lut_output_mask="0" is_global=
↪"true" is_io="true"/>
  <port type="sram" prefix="sram" size="64"/>
  <port type="sram" prefix="mode" size="2" mode_select="true" circuit_model_name="DFFR"␣
↪default_val="1"/>
</circuit_model>
```

The spypads are instantiated in the top-level verilog module `fpga_top.v`. `fpga_top.v` is automatically generated when we run our task from the OpenFPGA root directory. However, we need to modify the task configuration file to run the **full testbench** instead of the **formal testbench** to view the spypads' waveforms in GTKWave.

Note: To read about the differences between the **formal testbench** and the **full testbench**, please visit our page on testbenches: *Testbench*.

To open the task configuration file, run this command from the root directory of OpenFPGA:

```
emacs openfpga_flow/tasks/fpga_verilog/spypad/config/task.conf
```

The last line of the task configuration file (**LINE44**) sets the **formal testbench** to be the desired testbench. To use the **full testbench**, comment out **LINE44**. The file will look like this when finished:

```
1 # = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
2 # Configuration file for running experiments
3 # = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
4 # timeout_each_job : FPGA Task script splits fpga flow into multiple jobs
5 # Each job execute fpga_flow script on combination of architecture & benchmark
6 # timeout_each_job is timeout for each job
7 # = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
8 
9 [GENERAL]
10 run_engine=openfpga_shell
11 power_tech_file = ${PATH:OPENFPGA_PATH}/openfpga_flow/tech/PTM_45nm/45nm.xml
12 power_analysis = true
13 spice_output=false
14 verilog_output=true
15 timeout_each_job = 20*60
16 fpga_flow=vpr_blif
17 
18 [OpenFPGA_SHELL]
19 openfpga_shell_template=${PATH:OPENFPGA_PATH}/openfpga_flow/openfpga_shell_scripts/example_script.openfpga
20 openfpga_arch_file=${PATH:OPENFPGA_PATH}/openfpga_flow/openfpga_arch/k6_frac_N10_adder_register_scan_chain_depop50_spypad_40nm_openfpga.xml
21 openfpga_sim_setting_file=${PATH:OPENFPGA_PATH}/openfpga_flow/openfpga_simulation_settings/auto_sim_openfpga.xml
22 
23 [ARCHITECTURES]
24 arch0=${PATH:OPENFPGA_PATH}/openfpga_flow/vpr_arch/k6_frac_N10_tileable_adder_register_scan_chain_depop50_spypad_40nm.xml
25 
26 [BENCHMARKS]
27 bench0=${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/and2/and2.blif
28 # Cannot pass automatically. Need change in .v file to match ports
29 # When passed, we can replace the and2 benchmark
30 #bench0=${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/test_mode_low/test_mode_low.blif
31 
32 [SYNTHESIS_PARAM]
33 bench0_top = and2
34 bench0_act = ${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/and2/and2.act
35 bench0_verilog = ${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/and2/and2.v
36 
```

(continues on next page)

(continued from previous page)

```
#bench0_top = test_mode_low
#bench0_act = ${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/test_mode_low/test_mode_low.act
#bench0_verilog = ${PATH:OPENFPGA_PATH}/openfpga_flow/benchmarks/micro_benchmark/test_mode_low/test_mode_low.v
bench0_chan_width = 300

[SCRIPT_PARAM_MIN_ROUTE_CHAN_WIDTH]
end_flow_with_test=
#vpr_fpga_verilog_formal_verification_top_netlist=
```

Our OpenFPGA task will now run the full testbench. We run the task with the following command from the root directory of OpenFPGA:

```
python3 openfpga_flow/scripts/run_fpga_task.py fpga_verilog/spypad --debug --show_thread_logs
```

Note: Python 3.8 or later is required to run this task

We can now see the instantiation of these spypads in `fpga_top.v` and `luts.v`. We will start by viewing `luts.v` with the following command:

```
emacs openfpga_flow/tasks/fpga_verilog/spypad/latest/k6_frac_N10_tileable_adder_register_scan_chain_depop50_spypad_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/sub_module/luts.verilog
```

The spypads are coming from the `frac_lut6_spypad` circuit model. In `luts.v`, the `frac_lut6_spypad` module is defined around **LINE150** and looks as follows:

```
module frac_lut6_spypad(in,
sram,
sram_inv,
mode,
mode_inv,
lut4_out,
lut5_out,
lut6_out);
//----- INPUT PORTS -----
input [0:5] in;
//----- INPUT PORTS -----
input [0:63] sram;
//----- INPUT PORTS -----
input [0:63] sram_inv;
//----- INPUT PORTS -----
input [0:1] mode;
//----- INPUT PORTS -----
input [0:1] mode_inv;
//----- OUTPUT PORTS -----
output [0:3] lut4_out;
//----- OUTPUT PORTS -----
output [0:1] lut5_out;
//----- OUTPUT PORTS -----
```

(continues on next page)

(continued from previous page)

```
output [0:0] lut6_out;
```

The fpga_top.v file has some similarities. We can view the fpga_top.v file by running the following command:

```
emacs openfpga_flow/tasks/fpga_verilog/spypad/latest/k6_frac_N10_tileable_adder_register_
↪scan_chain_depop50_spypad_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/fpga_top.v
```

If we look at the module definition and ports of fpga_top.v we should see the following:

```
module fpga_top(pReset,
                prog_clk,
                TESTEN,
                set,
                reset,
                clk,
                gfpga_pad_frac_lut6_spypad_lut4_out,
                gfpga_pad_frac_lut6_spypad_lut5_out,
                gfpga_pad_frac_lut6_spypad_lut6_out,
                gfpga_pad_GPIO_PAD,
                ccff_head,
                ccff_tail);
//----- GLOBAL PORTS -----
input [0:0] pReset;
//----- GLOBAL PORTS -----
input [0:0] prog_clk;
//----- GLOBAL PORTS -----
input [0:0] TESTEN;
//----- GLOBAL PORTS -----
input [0:0] set;
//----- GLOBAL PORTS -----
input [0:0] reset;
//----- GLOBAL PORTS -----
input [0:0] clk;
//----- GPOUT PORTS -----
output [0:3] gfpga_pad_frac_lut6_spypad_lut4_out;
//----- GPOUT PORTS -----
output [0:1] gfpga_pad_frac_lut6_spypad_lut5_out;
//----- GPOUT PORTS -----
output [0:0] gfpga_pad_frac_lut6_spypad_lut6_out;
//----- GPIO PORTS -----
inout [0:7] gfpga_pad_GPIO_PAD;
//----- INPUT PORTS -----
input [0:0] ccff_head;
//----- OUTPUT PORTS -----
output [0:0] ccff_tail;
```

Using *General-purpose outputs as separated FPGA I/Os* as a guide, we can relate our task like Fig. 5.4

We can view testbench waveforms with GTKWave by running the following command from the root directory:

```
gtkwave openfpga_flow/tasks/fpga_verilog/spypad/latest/k6_frac_N10_tileable_adder_
↪register_scan_chain_depop50_spypad_40nm/and2/MIN_ROUTE_CHAN_WIDTH/and2_formal.vcd &
```

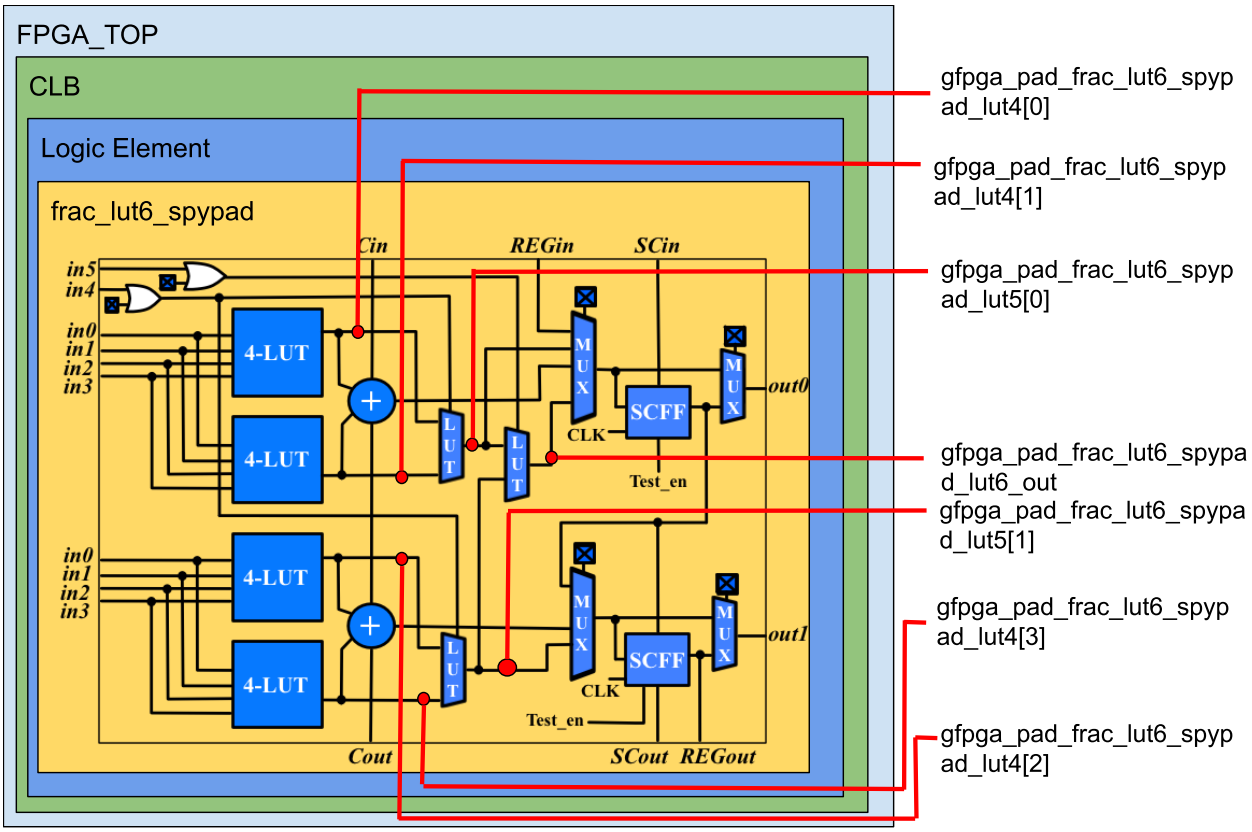


Fig. 5.4: An illustrative example of the `lut6` spypad sourced from inside a logic element.

Note: Information on GTKWave can be found on our documentation page located here: *From Verilog to Verification*

The waveforms will appear similar to Fig. 5.5

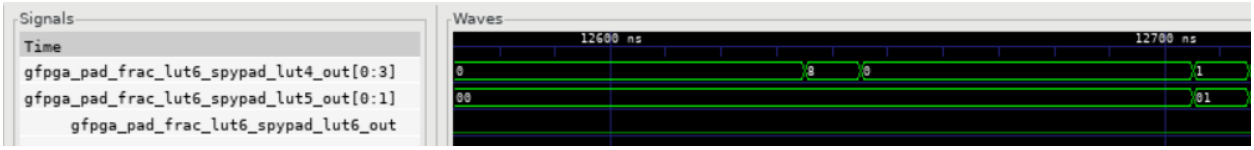


Fig. 5.5: Waveforms of `frac_lut6` spypads

5.4.3 Building Spypads

We will modify the k6_frac_N10_adder_chain_40nm_openfpga.xml file found in OpenFPGA to expose the **sumout** output from the **ADDF** module. We can start modifying the file by running the following command:

```
emacs openfpga_flow/openfpga_arch/k6_frac_N10_adder_chain_40nm_openfpga.xml
```

Replace **LINE214** with the following:

```
<port type="output" prefix="sumout" lib_name="SUM" size="1" is_global="true" is_
↪io="true"/>
```

sumout is now a global output. **sumout** will show up in the `fpga_top.v` file and will have waveforms in GTKWave if we run the **full testbench**. To run the **full testbench**, we have to modify the `hard_adder` configuration file:

```
emacs openfpga_flow/tasks/fpga_verilog/adder/hard_adder/config/task.conf
```

Comment out the last line of the file to run the **full testbench**:

```
#vpr_fpga_verilog_formal_verification_top_netlist=
```

We now run the task to see our changes:

```
python3 openfpga_flow/scripts/run_fpga_task.py fpga_verilog/adder/hard_adder --debug --
↪show_thread_logs
```

We can view the global ports in `fpga_top.v` by running the following command:

```
emacs openfpga_flow/tasks/fpga_verilog/adder/hard_adder/run064/k6_frac_N10_tileable_
↪adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/SRC/fpga_top.v
```

The `fpga_top.v` should have the following in its module definition:

```
module fpga_top(pReset,
                prog_clk,
                set,
                reset,
                clk,
                gfpga_pad_ADDF_sumout,
                gfpga_pad_GPIO_PAD,
```

(continues on next page)

(continued from previous page)

```
            ccff_head,
            ccff_tail);
//----- GLOBAL PORTS -----
input [0:0] pReset;
//----- GLOBAL PORTS -----
input [0:0] prog_clk;
//----- GLOBAL PORTS -----
input [0:0] set;
//----- GLOBAL PORTS -----
input [0:0] reset;
//----- GLOBAL PORTS -----
input [0:0] clk;
//----- GPOUT PORTS -----
output [0:19] gfpga_pad_ADDF_sumout;
```

The architecture will now look like Fig. 5.6

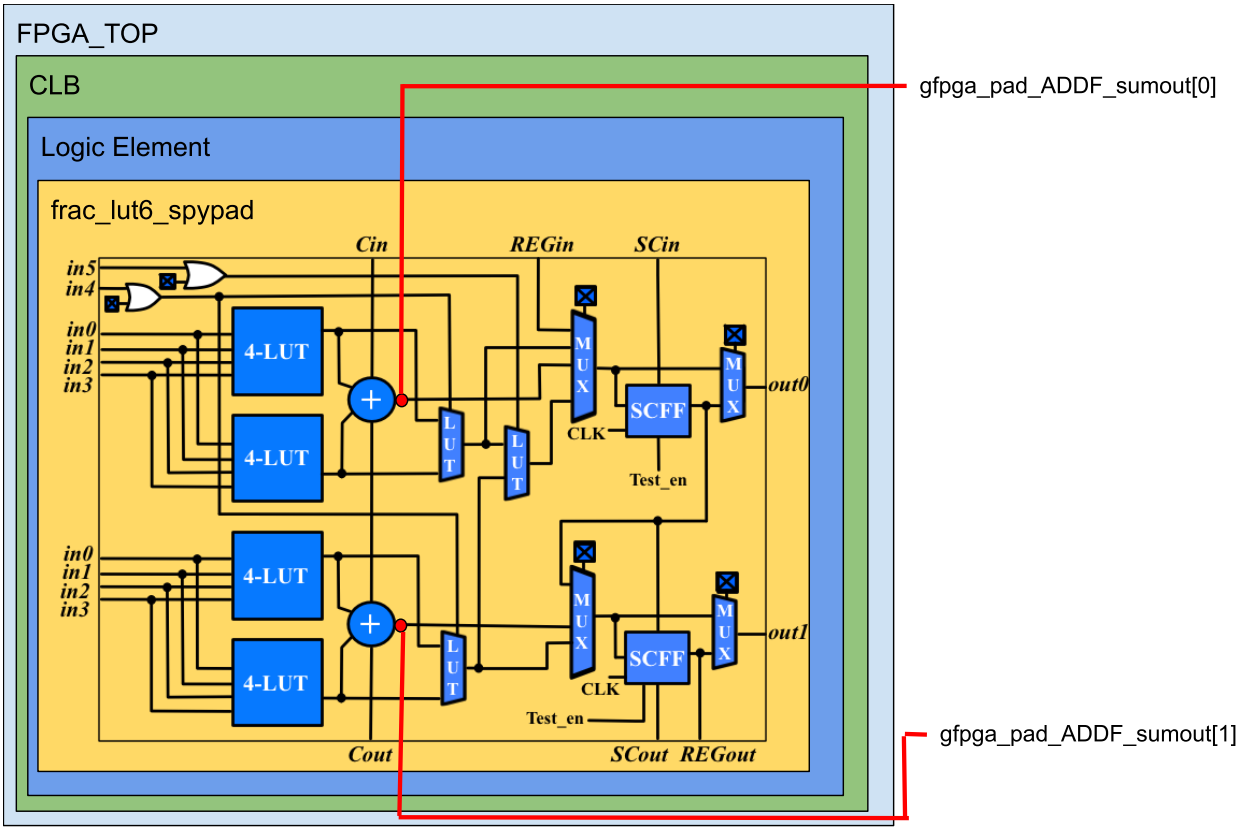


Fig. 5.6: An illustrative example of the sumout spypad sourced from an adder inside a logic element. There are 10 logic elements in a CLB, and we are looking at the 1st logic element.

We can view the waveform by running GTKWave:

```
gtkwave openfpga_flow/tasks/fpga_verilog/adder/hard_adder/latest/k6_frac_N10_tileable_
↪adder_chain_40nm/and2/MIN_ROUTE_CHAN_WIDTH/and2_formal.vcd &
```

The waveform should have some changes to its value. An example of what it may look like is displayed in Fig. 5.7

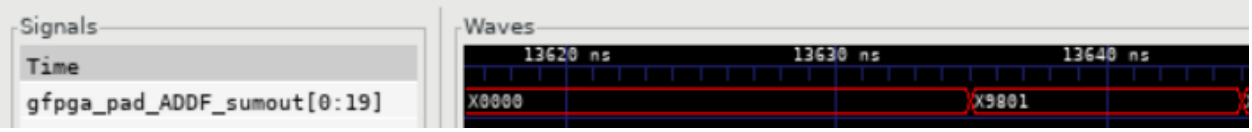


Fig. 5.7: Waveforms of `sumout` spypad

5.4.4 Conclusion

In this tutorial, we have shown how to build spypads into OpenFPGA Architectures using XML Syntax. If you have any issues, feel free to *Contact* us.

CHAPTER

SIX

OPENFPGA FLOW

6.1 OpenFPGA Flow

This python script executes the supported OpenFPGA flow for a single benchmark and architecture file for given script parameters.

The script is located at:

```
${OPENFPGA_PATH}/openfpga_flow/scripts/run_fpga_flow.py
```

6.1.1 Basic Usage

At a minimum `open_fpga_flow.py` requires following command-line arguments:

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

where:

- <architecture_file> is the target FPGA architecture
- <circuit_file> The list of files in the benchmark (Supports ../directory/*.v)
- <top_module_name> The name of the top level module in Verilog project

Note: The script will create a `tmp` run directory in base OpenFPGA path, unless otherwise specified with the *--run_dir* option. All stages of the flow will be run within run directory. Several intermediate files will be generated and maintian in run directory. The path variables declared in architecture XML file will be resolved with absolute path and copied to the `tmp/arch` directory before executing flow. All the benchmark files provided will be copied to `tmp/bench` directory without maintaining any directory structure. **Users should ensure that no important files are kept in this directory as script will clear directory before each execution**

6.1.2 OpenFPGA Variables

Frequently, while running OpenFPGA flow User is suppose to refer external files. To avoid long names and referencing errors user can use following openfpga variables. These variables are resolved with absolute path while execution making each run independent of launch directory.

- <OPENFPGA_PATH> Path to the base OpenFPGA directory
- <OPENFPGA_FLOW_PATH> Path to the run_fpga_flow script directory
- <SPICENETLIST_PATH> Path where spice netlists are saved
- <VERILOG_PATH> Path where Verilog modules are saved
- <TECH_PATH> Path where all characterized XML files are stored

For example in architecture file path vairable can be used as follows:

```
.... lib_path="${TECH_PATH}/PTM_45nm/45nm.pm" ....
```

6.1.3 Output

Based on which flow is executed, resulting in intermediate files are generated in run_directory

The output log of the script provides the status of each stage to the user. If any stage failed to execute, the output log would indicate the stage at which execution failed, and execution traceback.

In case of successful execution, The OpenFPGA flow script will parse parameters listed in configuration from different result files and will create `vpr_stat.txt`, `vpr_stat_power.txt` (optional) file in run_directory.

6.1.4 Advanced Usage

User can pass additional *optional* command arguments to `run_fpga_flow.py` script:

```
run_fpga_flow.py <architecture_file> <benchmark_files> [<options>] [<vpr_options>] [
↪<fpga-verilog_options>] [<fpga-spice_options>] [<fpga-bitstream_options>] [<ace_
↪options>]
```

where:

- <options> are additional arguments passed to `run_fpga_flow.py` (described below),
- <vpr_options> Any argument prefixed with `--vpr-*` will be forwarded to vpr script as it is. The detail of supported vpr argument is available `Add corrrect reference`
- <fpga-verilog_options> are any arguments not recognized by `run_vtr_flow.pl`. These will be forwarded to VPR.
- <ace_options> these arguments will be passed to ACE activity estimator program

For example:

```
run_fpga_flow.py my_circuit.v my_arch.xml -track_memory_usage --pack --place
```

will run the VTR flow to map the circuit `my_circuit.v` onto the architecture `my_arch.xml`; the arguments `--pack` and `--place` will be passed to VPR (since they are unrecognized arguments to `run_vtr_flow.pl`). They will cause VPR to perform only packing and placement.

6.1.5 Detailed Command-line Options

Note: All the commnadline arguments starting with `vpr_*` , `fpga-verilog_*` , `fpga-spice_*` or `fpga-bitstream_*` will be passed to VPR without suffix

General Arguments

--top_module <name>

Provide top module name of the benchmark. Default `top`

--run_dir <directory_path>

Using this option user can provide a custom path as a run directory. Default is `tmp` directory in OpenFPGA root path.

--K <lut_inputs>

This option defines the number of inputs to the LUT. By default, the script parses provided architecture file and finds out inputs to the biggest LUT.

--yosys_tmpl <yosys_template_file>

This option allows the user to provide a custom Yosys template while running a yosys_vpr flow. Default template is stored in a directory `open_fpga_flow\misc\ys_tmpl_yosys_vpr_flow.ys`. Alternately, user can create a copy and modify according to their need. Yosys template script supports `TOP_MODULE` `READ_VERILOG_OPTIONS` `VERILOG_FILES` `LUT_SIZE` & `OUTPUT_BLIF` variables. In case if `--verific` option is provided then `ADD_INCLUDE_DIR`, `ADD_LIBRARY_DIR`, `ADD_BLACKBOX_MODULES`, `READ_HDL_FILE` (should be used instead of `READ_VERILOG_OPTIONS` and `VERILOG_FILES`) and `READ_LIBRARY` additional varialbes are supported. The variables can be used as `${var_name}`.

--ys_rewrite_tmpl <yosys_rewrite_template_file>

This option allows the user to provide an alternate Yosys template to rewrite Verilog netlist while running a yosys_vpr flow. The alternate Yosys template script supports all of the main Yosys template script variables.

--verific

This option specifies to use Verific as a frontend for Yosys while running a yosys_vpr flow. The following standards are used by default for reading input HDL files: * Verilog - `vlog95` * System Verilog - `sv2012` * VHDL - `vhdl2008` The option should be used only with custom Yosys template containing Verific commands.

--debug

To enable detailed log printing.

--flow_config

User can provide option flow configuration file to override some of the default script parameters. for detail information refer *OpenFPGA Flow Configuration*

ACE Arguments

--black_box_ace
Performs ACE simulation on the black box [deprecated]

VPR RUN Arguments

--fix_route_chan_width <channel_number>
Performs VPR implementation for a fixed number of channels defined as the 'channel_number'

--min_route_chan_width <percentage_slack>
Performs VPR implementation to get minimum channel width and then perform fixed channel rerouting with percentage_slack increase in the channel width.

--max_route_width_retry <max_retry_count>
Number of times the channel width should be increased and attempt VPR implementation, while performing min_route_chan_width

--power

--power_tech

blif_vpr_flow Arguments

--activity_file
Activity to be used for the given benchmark while running blif_vpr_flow

--base_verilog
Verilog benchmark file to perform verification while running bliff_vpr_flow

6.1.6 OpenFPGA Flow Configuration file

The OpenFPGA Flow configuration file consists of following sections

- **CAD_TOOLS_PATH**
  Lists executable file path for different CAD tools used in the script
- **FLOW_SCRIPT_CONFIG**
  Lists the supported flows by the script.
- **DEFAULT_PARSE_RESULT_VPR**
  List of default parameters to be parsed from Place, Pack, and Route output
- **DEFAULT_PARSE_RESULT_POWER**
  List of default parameters to be parsed from VPR power analysis output
- **INTERMIDIATE_FILE_PREFIX**
  [Not implemented yet]

Default OpenFPGA_flow Configuration file is located in open_fpga_flow\misc\fpgaflow_default_tool_path.conf. User-supplied configuration file overrides or extends the default configuration.

6.2 OpenFPGA Task

Tasks provide a framework for running the *OpenFPGA Flow* on multiple benchmarks, architectures, and set of OpenFPGA parameters. The structure of the framework is very similar to VTR-Tasks implementation with additional functionality and minor file extension changes.

6.2.1 Task Directory

The tasks are stored in a `TASK_DIRECTORY`, which by default points to `${OPENFPGA_PATH}/openfpga_flow/tasks`. Every directory or sub-directory in task directory consisting of `../config/task.conf` file can be referred to as a task.

To create as task name called `basic_flow` following directory has to exist:

```
${TASK_DIRECTORY}/basic_flow/conf/task.conf
```

Similarly `regression/regression_quick` expect following structure:

```
${TASK_DIRECTORY}/regression/regression_quick/conf/task.conf
```

6.2.2 Running OpenFPGA Task:

At a minimum `open_fpga_flow.py` requires following command-line arguments:

```
open_fpga_flow.py <task1_name> <task2_name> ... [<options>]
```

where:

- <task_name> is the name of the task to run
- <options> Other command line arguments described below

6.2.3 Command-line Options

--maxthreads <number_of_threads>

This option defines the number of threads to run while executing task. Each combination of architecture, benchmark and set of OpenFPGA Flow options runs in a individual thread.

--skip_thread_logs

Passsing this option skips printing logs from each OpenFPGA Flow script run.

--exit_on_fail

Passsing this option exits the OpenFPGA task script with returncode 1, if any threads fail to execute successfully. It is mainly used to while performing regression test.

--test_run

This option allows to debug OpenFPGA Task script by skiping actual execution of OpenFPGA flow . Passing this option prints the list of commnad generated to execute using OpenFPGA flow.

--debug

To enable detailed log printing.

6.2.4 Creating a new OpenFPGA Task

- Create the folder `${TASK_DIRECTORY}/<task_name>`
- Create a file `${TASK_DIRECTORY}/<task_name>/config/task.conf` in it
- Configure the task as explained in *Configuring a new OpenFPGA Task*

6.2.5 Configuring a new OpenFPGA Task

The task configuration file `task.conf` consists of `GENERAL`, `ARCHITECTURES`, `BENCHMARKS`, `SYNTHESIS_PARAM` and `SCRIPT_PARAM_<var_name>` sections. Declaring all the above sections are mandatory.

Note: The configuration file supports all the OpenFPGA Variables refer *OpenFPGA Variables* section to know more. Variable in the configuration file is declared as `${PATH:<variable_name>}`

General Section

fpga_flow=<yosys_vpr|vpr_blif|yosys>

This option defines which OpenFPGA flow to run. By default `yosys_vpr` is executed.

power_analysis=<true|false>

Specifies whether to perform power analysis or not.

power_tech_file=<path_to_tech_XML_file>

Declares which tech XML file to use while performing Power Analysis.

spice_output=<true|false>

Setting up this variable generates Spice Netlist at the end of the flow. Equivalent of passing `--vpr_fpga_spice` command to *OpenFPGA Flow*

verilog_output=<true|false>

Setting up this variable generates Verilog Netlist at the end of the flow. Equivalent of passing `--vpr_fpga_spice` command to *OpenFPGA Flow*

timeout_each_job=<true|false>

Specifies the timeout for each *OpenFPGA Flow* execution. Default is set to `20 min.`

verific=<true|false>

Specifies to use Verific as a frontend for Yosys while running a yosys_vpr flow. The following standards are used by default for reading input HDL files: * Verilog - `vlog95` * System Verilog - `sv2012` * VHDL - `vhdl2008` The option should be used only with custom Yosys template containing Verific commands.

OpenFPGA_SHELL Sections

User can specify OpenFPGA_SHELL options in this section.

Architectures Sections

User can define the list of architecture files in this section.

arch<arch_label>=<xml_architecture_file_path>

The arch_label variable can be any number of string without white-spaces. xml_architecture_file_path is path to the actual XML architecture file

Note: In the final OpenFPGA Task result, the architecture will be referred by its arch_label.

Benchmarks Sections

User can define the list of benchmarks files in this section.

bench<bench_label>=<list_of_files_in_benchmark>

The bench_label variable can be any number of string without white-spaces. list_of_files_in_benchmark is a list of benchmark HDL files paths.

For Example following code shows how to define a benchmarks, with a single file, multiple files and files added from a specific directory.

```
[BENCHMARKS]
# To declare single benchmark file
bench_design1=${BENCH_PATH}/design/top.v

# To declare multiple benchmark file
bench_design2=${BENCH_PATH}/design/top.v,${BENCH_PATH}/design/sub_module.v

# To add all files in specific directory to the benchmark
bench_design3=${BENCH_PATH}/design/top.v,${BENCH_PATH}/design/lib/*.v
```

Note: bench_label is referred again in Synthesis_Param section to provide additional information about benchmark

Synthesis Parameter Sections

User can define extra parameters for each benchmark in the BENCHMARKS sections.

bench<bench_label>_top=<Top_Module_Name>

This option defines the Top Level module name for bench_label benchmark. By default, the top-level module name is considered as a top.

bench<bench_label>_yosys=<yosys_template_file>

This config defines Yosys template script file.

bench<bench_label>_chan_width=<chan_width_to_use>

In case of running fixed channel width routing for each benchmark, this option defines the channel width to be used for bench_label benchmark

bench<bench_label>_act=<activity_file_path>

In case of running `blif_vpr_flow` this option provides the activity files to be used to generate testbench for `bench_label` benchmark

---

Note: This file is required only when the `power_analysis` option in the general section is enabled. Otherwise, it is optional

---

bench<bench_label>_verilog=<source_verilog_file_path>

In case of running `blif_vpr_flow` with verification this option provides the source Verilog design for `bench_label` benchmark to be used while verification.

bench<bench_label>_read_verilog_options=<Options>

This config defines the `read_verilog` command options for `bench_label` benchmark.

bench<bench_label>_yosys_args=<Arguments>

This config defines Yosys arguments to be used in QuickLogic synthesis script for `bench_label` benchmark.

bench<bench_label>_yosys_dff_map_verilog=<dff_technology_file_path>

This config defines DFF technology file to be used in technology mapping for `bench_label` benchmark.

bench<bench_label>_yosys_bram_map_verilog=<bram_technology_file_path>

This config defines BRAM technology file to be used in technology mapping for `bench_label` benchmark.

bench<bench_label>_yosys_bram_map_rules=<bram_technology_rules_file_path>

This config defines BRAM technology rules file to be used in technology mapping for `bench_label` benchmark. This config should be used with `bench<bench_label>_yosys_bram_map_verilog` config.

bench<bench_label>_yosys_dsp_map_verilog=<dsp_technology_file_path>

This config defines DSP technology file to be used in technology mapping for `bench_label` benchmark.

bench<bench_label>_yosys_dsp_map_parameters=<dsp_mapping_parameters>

This config defines DSP technology parameters to be used in technology mapping for `bench_label` benchmark. This config should be used with `bench<bench_label>_yosys_dsp_map_verilog` config.

bench<bench_label>_verific_include_dir=<include_dir_path>

This config defines include directory path for `bench_label` benchmark. Verific will search in this directory to find included files. If there are multiple paths then they can be provided as a comma separated list.

bench<bench_label>_verific_library_dir=<library_dir_path>

This config defines library directory path for `bench_label` benchmark. Verific will search in this directory to find undefined modules. If there are multiple paths then they can be provided as a comma separated list.

bench<bench_label>_verific_verilog_standard=<-vlog95|-vlog2k>

The config specifies Verilog language standard to be used while reading the Verilog files for `bench_label` benchmark.

bench<bench_label>_verific_systemverilog_standard=<-sv2005|-sv2009|-sv2012>

The config specifies SystemVerilog language standard to be used while reading the SystemVerilog files for `bench_label` benchmark.

bench<bench_label>_verific_vhdl_standard=<-vhdl87|-vhdl93|-vhdl2k|-vhdl2008>

The config specifies VHDL language standard to be used while reading the VHDL files for `bench_label` benchmark.

bench<bench_label>_verific_read_lib_name<lib_label>=<lib_name>

The lib_label variable can be any number of string without white-spaces. The config specifies library name for bench_label benchmark where Verilog/SystemVerilog/VHDL files specified by bench<bench_label>_verific_read_lib_src<lib_label> config will be loaded. This config should be used only with bench<bench_label>_verific_read_lib_src<lib_label> config.

bench<bench_label>_verific_read_lib_src<lib_label>=<library_src_files>

The lib_label variable can be any number of string without white-spaces. The config specifies Verilog/SystemVerilog/VHDL files to be loaded into library specified by bench<bench_label>_verific_read_lib_name<lib_label> config for bench_label benchmark. The library_src_files should be the source files names separated by commas. This config should be used only with bench<bench_label>_verific_read_lib_name<lib_label> config.

bench<bench_label>_verific_search_lib=<lib_name>

The config specifies library name for bench_label benchmark from where Verific will look up for external definitions while reading HDL files.

bench<bench_label>_yosys_cell_sim_verilog=<verilog_files>

The config specifies Verilog files for bench_label benchmark which should be separated by comma.

bench<bench_label>_yosys_cell_sim_systemverilog=<systemverilog_files>

The config specifies SystemVerilog files for bench_label benchmark which should be separated by comma.

bench<bench_label>_yosys_cell_sim_vhdl=<vhdl_files>

The config specifies VHDL files for bench_label benchmark which should be separated by comma.

bench<bench_label>_yosys_blackbox_modules=<blackbox_modules>

The config specifies blackbox modules names for bench_label benchmark which should be separated by comma (usually these are the modules defined in files specified with bench<bench_label>_yosys_cell_sim_<verilog/systemverilog/vhdl> option).

Note: The following configs might be common for all benchmarks:

- bench<bench_label>_yosys
- bench<bench_label>_chan_width
- bench<bench_label>_read_verilog_options
- bench<bench_label>_yosys_args
- bench<bench_label>_yosys_bram_map_rules
- bench<bench_label>_yosys_bram_map_verilog
- bench<bench_label>_yosys_cell_sim_verilog
- bench<bench_label>_yosys_cell_sim_systemverilog
- bench<bench_label>_yosys_cell_sim_vhdl
- bench<bench_label>_yosys_blackbox_modules
- bench<bench_label>_yosys_dff_map_verilog
- bench<bench_label>_yosys_dsp_map_parameters
- bench<bench_label>_yosys_dsp_map_verilog
- bench<bench_label>_verific_verilog_standard

- bench<bench_label>_verific_systemverilog_standard
- bench<bench_label>_verific_vhdl_standard
- bench<bench_label>_verific_include_dir
- bench<bench_label>_verific_library_dir
- bench<bench_label>_verific_search_lib

The following syntax should be used to define common config: bench_<config_name>_common

Script Parameter Sections

The script parameter section lists set of commnad line pararmeters to be passed to *OpenFPGA Flow* script. The section name is defines as SCRIPT_PARAM_<parameter_set_label> where *parameter_set_label* can be any word without white spaces. The section is referred with parameter_set_label in the final result file.

For example following code Specifies the two sets (Fixed_Routing_30 and Fixed_Routing_50) of *OpenFPGA Flow* arguments.

```
[SCRIPT_PARAM_Fixed_Routing_30]
# Execute fixed routing with channel with 30
fix_route_chan_width=30

[SCRIPT_PARAM_Fixed_Routing_50]
# Execute fixed routing with channel with 50
fix_route_chan_width=50
```

6.2.6 Example Task Configuration File

```
[GENERAL]
spice_output=false
verilog_output=false
power_analysis = true
power_tech_file = ${PATH:TECH_PATH}/winbond90nm/winbond90nm_power_properties.xml
timeout_each_job = 20*60

[ARCHITECTURES]
arch0=${PATH:ARCH_PATH}/winbond90/k6_N10_rram_memory_bank_SC_winbond90.xml

[BENCHMARKS]
bench0=${PATH:BENCH_PATH}/MCNC_Verilog/s298/s298.v
bench1=${PATH:BENCH_PATH}/MCNC_Verilog/elliptic/elliptic.v

[SYNTHESIS_PARAM]
bench0_top = s298
bench1_top = elliptic

[SCRIPT_PARAM_Slack_30]
min_route_chan_width=1.3

[SCRIPT_PARAM_Slack_80]
min_route_chan_width=1.8
```

CHAPTER

SEVEN

OPENFPGA ARCHITECTURE DESCRIPTION

7.1 General Hierarchy

OpenFPGA uses separated XMLs file other than the VPR8 architecture description file. This is to keep a loose integration to VPR8 so that OpenFPGA can easily integrate any future version of VPR with least engineering effort. However, to implement a physical FPGA, OpenFPGA requires the original VPR XML to include full physical design details. Full syntax can be found in *Additional Syntax to Original VPR XML*.

The OpenFPGA requires two XML files: an architecture description file and a simulation setting description file.

7.1.1 OpenFPGA Architecture Description File

This file contains device-level and circuit-level details as well as annotations to the original VPR architecture. It contains a root node called <openfpga_architecture> under which architecture-level information, such as device-level description, circuit-level and architecture annotations to original VPR architecture XML are defined.

It consists of the following code blocks

- <circuit_library> includes a number of circuit_model, each of which describe a primitive block in FPGA architecture, such as Look-Up Tables and multiplexers. Full syntax can be found in *Circuit Library*.
- <technology_library> includes transistor-level parameters, where users can specify which transistor models are going to be used when building the circuit models. Full syntax can be found in *Technology library*.
- <configuration_protocol> includes detailed description on the configuration protocols to be used in FPGA fabric. Full syntax can be found in *Configuration Protocol*.
- <connection_block> includes annotation on the connection block definition <connection_block> in original VPR XML. Full syntax can be found in *Bind circuit modules to VPR architecture*.
- <switch_block> includes annotation on the switch block definition <switchlist> in original VPR XML. Full syntax can be found in *Bind circuit modules to VPR architecture*.
- <routing_segment> includes annotation on the routing segment definition <segmentlist> in original VPR XML. Full syntax can be found in *Bind circuit modules to VPR architecture*.
- <direct_connection> includes annotation on the inter-tile direct connection definitioin <directlist> in original VPR XML. Full syntax can be found in *Inter-Tile Direct Interconnection extensions*.
- <pb_type_annotation> includes annotation on the programmable block architecture <complexblocklist> in original VPR XML. Full syntax can be found in *Bind circuit modules to VPR architecture*.

Note: <technology_library> will be applied to circuit_model when running FPGA-SPICE. It will not impact FPGA-Verilog, FPGA-Bitstream, FPGA-SDC.

7.1.2 OpenFPGA Simulation Setting File

This file contains parameters required by testbench generators. It contains a root node <openfpga_simulation_setting>, under which all the parameters to be used in generate testbenches in simulation purpose are defined.

It consists of the following code blocks

- <clock_setting> defines the clock-related settings in simulation, such as clock frequency and number of clock cycles to be used.
- <simulator_option> defines universal options available in both HDL and SPICE simulators. This is mainly used by *FPGA-SPICE*.
- <monte_carlo> defines critical parameters to be used in monte-carlo simulations. This is used by *FPGA-SPICE*.
- <measurement_setting> defines the parameters used to measure signal slew and delays. This is used by *FPGA-SPICE*.
- <stimulus> defines the parameters used to generate voltage stimuli in testbenches. This is used by *FPGA-SPICE*.

Full syntax can be found in *Simulation settings*.

Note: the parameters in <clock_setting> will be applied to both FPGA-Verilog and FPGA-SPICE simulations

7.2 Additional Syntax to Original VPR XML

Warning: Note this is only applicable to VPR8!

7.2.1 Models, Complex blocks and Physical Tiles

Each <pb_type> should contain a <mode> that describes the physical implementation of the <pb_type>. Note that this is fully compatible to the VPR architecture XML syntax.

Note: <model> should include the models that describe the primitive <pb_type> in physical mode.

Note: Currently, OpenFPGA only supports 1 <equivalent_sites> to be defined under each <tile>

<mode disable_packing="<bool">/>

OpenFPGA allows users to define it a mode is disabled for VPR packer. By default, the disable_packing is set to false. This is mainly used for the mode that describes the physical implementation, which is typically not packable. Disable it in the packing and signficantly accelerate the packing runtime.

Note: Once a mode is disabled in packing, its child modes will be disabled as well.

Note: The following syntax is only available in OpenFPGA!

We allow more flexible pin location assignment when a <tile> has a capacity > 1. User can specify the location using the index of instance, e.g.,

```
<tile name="io_bottom" capacity="6" area="0">
  <equivalent_sites>
    <site pb_type="io"/>
  </equivalent_sites>
  <input name="outpad" num_pins="1"/>
  <output name="inpad" num_pins="1"/>
  <fc in_type="frac" in_val="0.15" out_type="frac" out_val="0.10"/>
  <pinlocations pattern="custom">
    <loc side="top">io_bottom[0:1].outpad io_bottom[0:3].inpad io_bottom[2:5].outpad io_
↪bottom[4:5].inpad</loc>
  </pinlocations>
</tile>
```

7.2.2 Layout

<layout> may include additioinal syntax to enable tileable routing resource graph generation

tileable="<bool>"

Turn on/off tileable routing resource graph generator.

Tileable routing architecture can minimize the number of unique modules in FPGA fabric to be physically implemented.

Technical details can be found in [TGAG19].

Note: Strongly recommend to enable the tileable routing architecture when you want to PnR large FPGA fabrics, which can effectively reduce the runtime.

---

through_channel="<bool>"

Allow routing channels to pass through multi-width and multi-height programable blocks. This is mainly used in heterogeneous FPGAs to increase routability, as illustrated in Fig. 7.1. By default, it is off.

Warning: Do NOT enable through_channel if you are not using the tileable routing resource graph generator!

Warning: You cannot use spread pin location for the height > 1 or width >1 tiles when using the tileable routing resource graph!!! Otherwise, it will cause undriven pins in your device!!!

A quick example to show tileable routing is enabled and through channels are disabled:

```
<layout tileable="true" through_channel="false">
</layout>
```

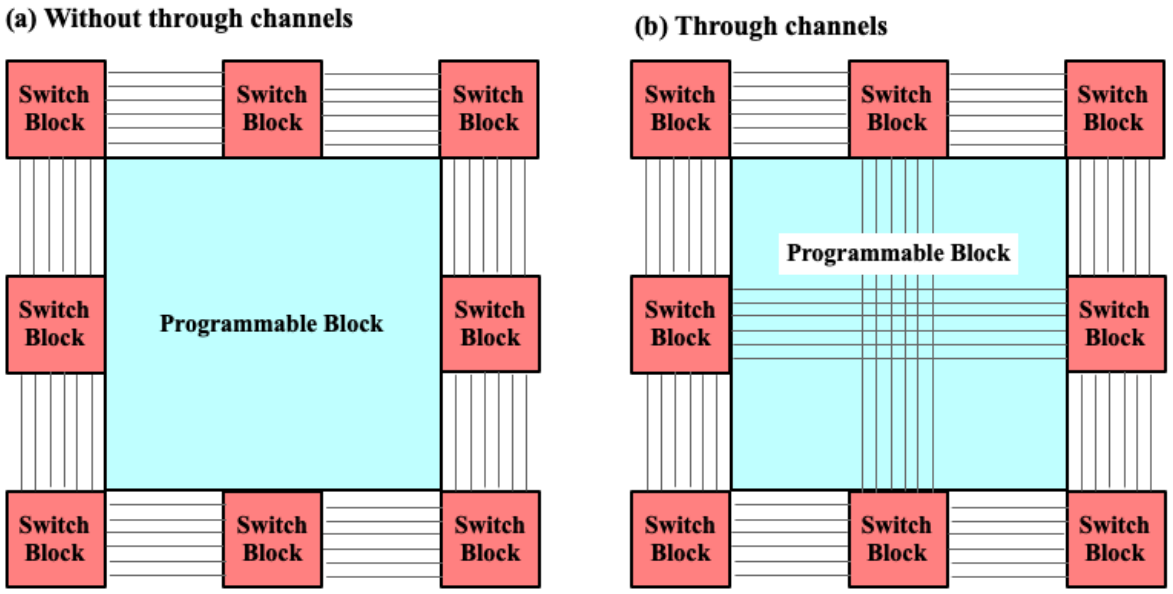


Fig. 7.1: Impact on routing architecture when through channel in multi-width and multi-height programmable blocks: (a) disabled; (b) enabled.

7.2.3 Switch Block

<switch_block> may include addition syntax to enable different connectivity for pass tracks

sub_type="<string>"

Connecting type for pass tracks in each switch block The supported connecting patterns are `subset`, `universal` and `wilton`, being the same as VPR capability If not specified, the pass tracks will the same connecting patterns as start/end tracks, which are defined in `type`

sub_Fs="<int>"

Connectivity parameter for pass tracks in each switch block. Must be a multiple of 3. If not specified, the pass tracks will the same connectivity as start/end tracks, which are defined in `fs`

A quick example which defines a switch block

- Starting/ending routing tracks are connected in the `wilton` pattern
- Each starting/ending routing track can drive 3 other starting/ending routing tracks
- Passing routing tracks are connected in the `subset` pattern
- Each passing routing track can drive 6 other starting/ending routing tracks

```
<device>
  <switch_block type="wilton" fs="3" sub_type="subset" sub_fs="6"/>
</device>
```

7.2.4 Routing Segments

OpenFPGA suggests users to give explicit names for each routing segement in <segmentlist> This is used to link circuit_model to routing segments.

A quick example which defines a length-4 uni-directional routing segment called L4 :

```
<segmentlist>
  <segment name="L4" freq="1" length="4" type="undir"/>
</segmentlist>
```

Note: Currently, OpenFPGA only supports uni-directional routing architectures

7.3 Configuration Protocol

Configuration protocol is the circuitry designed to program an FPGA. As an interface, configuration protocol could be really different in FPGAs, depending on the application context. OpenFPGA supports versatile configuration protocol, providing different trade-offs between speed and area.

7.3.1 Template

```
<configuration_protocol>
  <organization type="<string>" circuit_model_name="<string>" num_regions="<int>"/>
</configuration_protocol>
```

type="scan_chain|memory_bank|standalone|frame_based|ql_memory_bank"

Specify the type of configuration circuits.

OpenFPGA supports different types of configuration protocols to program FPGA fabrics:

- scan_chain: configurable memories are connected in a chain. Bitstream is loaded serially to program a FPGA
- frame_based: configurable memories are organized by frames. Each module of a FPGA fabric, e.g., Configurable Logic Block (CLB), Switch Block (SB) and Connection Block (CB), is considered as a frame of configurable memories. Inside each frame, all the memory banks are accessed through an address decoder. Users can write each memory cell with a specific address. Note that the frame-based memory organization is applid hierarchically. Each frame may consists of a number of sub frames, each of which follows the similar organization.
- memory_bank: configurable memories are organized in an array, where each element can be accessed by an unique address to the BL/WL decoders
- ql_memory_bank: configurable memories are organized in an array, where each element can be accessed by an unique address to the BL/WL decoders. This is a physical design friendly memory bank organization, where BL/WLs are efficiently shared by programmable blocks per column and row
- standalone: configurable memories are directly accessed through ports of FPGA fabrics. In other words, there are no protocol to control the memories. This allows full customization on the configuration protocol for hardware engineers.

Note: Avoid to use `standalone` when designing an FPGA chip. It will causes a huge number of I/Os required, far beyond any package size. It is well applicable to eFPGAs, where designers do need customized protocols between FPGA and processors.

Warning: Currently FPGA-SPICE only supports standalone memory organization.

Warning: Currently RRAM-based FPGA only supports memory-bank organization for Verilog Generator.

`circuit_model_name`="<string>"

Specify the name of circuit model to be used as configurable memory.

- `scan_chain` requires a circuit model type of `ccff`
- `frame_based` requires a circuit model type of `sram`
- `memory_bank` requires a circuit model type of `sram`
- `ql_memory_bank` requires a circuit model type of `sram`
- `standalone` requires a circuit model type of `sram`

`num_regions`="<int>"

Specify the number of configuration regions to be used across the fabrics. By default, it will be only 1 configuration region. Each configuration region contains independent configuration protocols, but the whole fabric should employ the same type of configuration protocols. For example, an FPGA fabric consists of 4 configuration regions, each of which includes a configuration chain. The more configuration chain to be used, the fast configuration runtime will be, but at the cost of more I/Os in the FPGA fabrics. The organization of each configurable region can be customized through the fabric key (see details in *Fabric Key*).

Warning: Currently, multiple configuration regions is not applicable to

- `standalone` configuration protocol.
- `ql_memory_bank` configuration protocol when BL/WL protocol `flatten` is selected

Note: For `ql_memory_bank` configuration protocol when BL/WL protocol `shift_register` is selected, different configuration regions **cannot** share any WLs on the same row! In such case, the default fabric key may not work. Strongly recommend to craft your own fabric key based on your configuration region plannning!

7.3.2 Configuration Chain Example

The following XML code describes a scan-chain circuitry to configure the core logic of FPGA, as illustrated in Fig. 7.2. It will use the circuit model defined in Fig. 7.37.

```
<configuration_protocol>
  <organization type="scan_chain" circuit_model_name="ccff" num_regions="<int>"/>
</configuration_protocol>
```

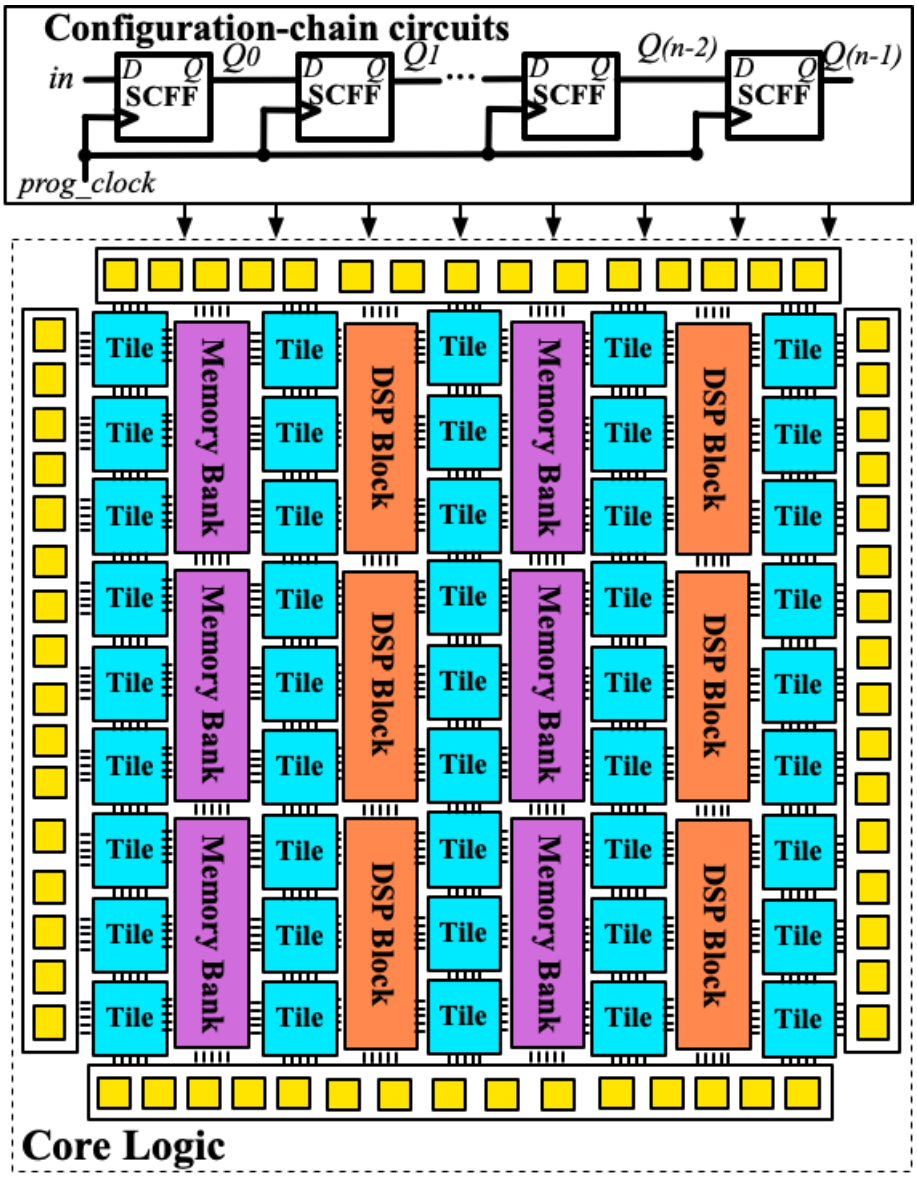


Fig. 7.2: Example of a configuration chain to program core logic of a FPGA

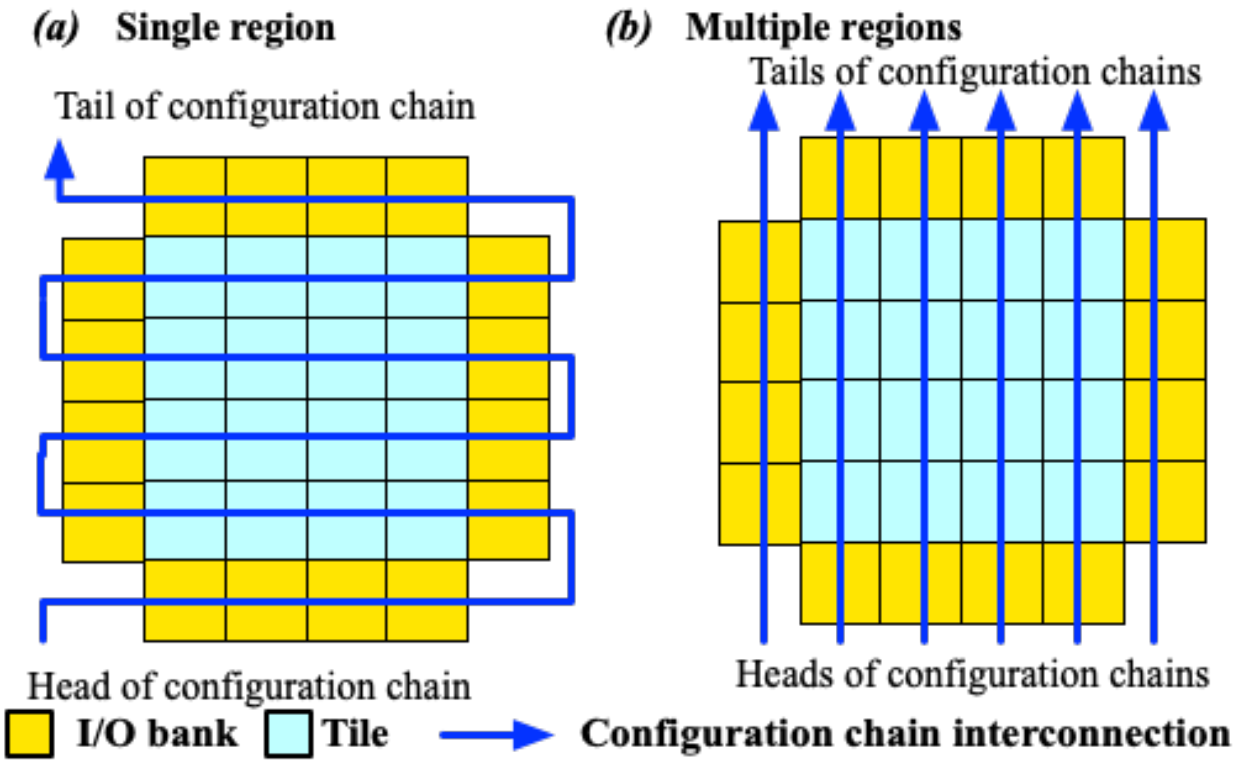


Fig. 7.3: Examples of single- and multiple- region configuration chains

7.3.3 Frame-based Example

The following XML code describes frame-based memory banks to configure the core logic of FPGA. It will use the circuit model defined in Fig. 7.26.

```
<configuration_protocol>
  <organization type="frame_based" circuit_model_name="config_latch"/>
</configuration_protocol>
```

Through frame-based configuration protocol, each memory cell can be accessed with an unique address given to decoders. Fig. 7.4 illustrates an example about how the configurable memories are organizaed inside a Logic Element (LE) shown in Fig. 5.1. The decoder inside the LE will enable the decoders of the Look-Up Table (LUT) and the routing multiplexer, based on the given address at `address[2:2]`. When the decoder of sub block, e.g., the LUT, is enabled, each memory cells can be accessed throught the `address[1:0]` and the data to write is provided at `data_in`.

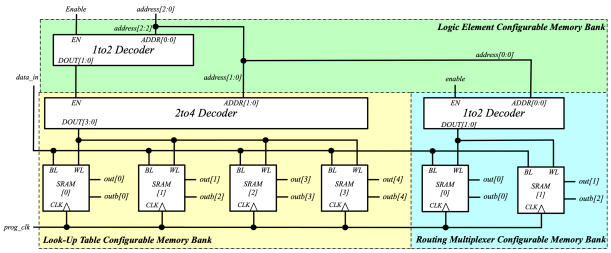


Fig. 7.4: Example of a frame-based memory organization inside a Logic Element

Fig. 7.5 shows a hierarchical view on how the frame-based decoders across a FPGA fabric.

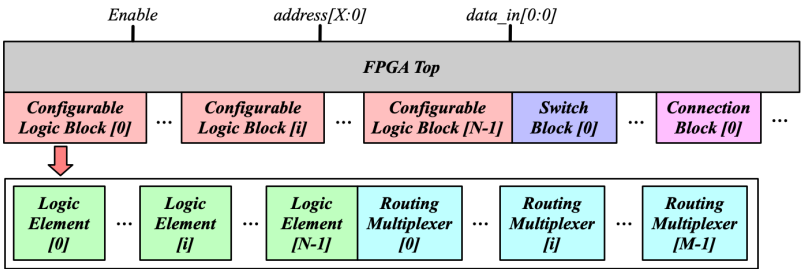


Fig. 7.5: Frame-based memory organization in a hierarchical view

Note: Frame-based decoders does require a memory cell to have

- two outputs (one regular and another inverted)
- a Bit-Line input to load the data
- a Word-Line input to enable data write

Warning: Please do NOT add inverted Bit-Line and Word-Line inputs. It is not supported yet!

When multiple configuration region is applied, the configuration frames will be grouped into different configuration regions. Each region has a separated data input bus and dedicated address decoders. As such, the configuration frame groups can be programmed in parallel.

7.3.4 Memory bank Example

The following XML code describes a memory-bank circuitry to configure the core logic of FPGA, as illustrated in Fig. 7.6. It will use the circuit model defined in Fig. 7.24. Users can customized the number of memory banks to be used across the fabrics. By default, it will be only 1 memory bank. Fig. 7.6 shows an example where 4 memory banks are defined. The more memory bank to be used, the fast configuration runtime will be, but at the cost of more I/Os in the FPGA fabrics. The organization of each configurable region can be customized through the fabric key (see details in *Fabric Key*).

```
<configuration_protocol>
  <organization type="memory_bank" circuit_model_name="sram_blwl"/>
</configuration_protocol>
```

Note: Memory-bank decoders does require a memory cell to have

- two outputs (one regular and another inverted)
- a Bit-Line input to load the data
- a Word-Line input to enable data write

Warning: Please do NOT add inverted Bit-Line and Word-Line inputs. It is not supported yet!

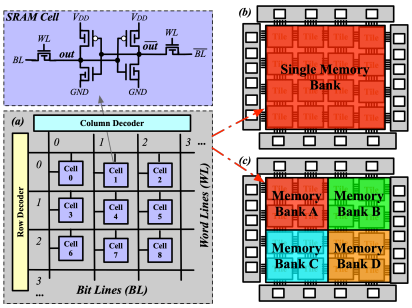


Fig. 7.6: Example of (a) a memory organization using memory decoders; (b) single memory bank across the fabric; and (c) multiple memory banks across the fabric.

7.3.5 QuickLogic Memory bank Example

The following XML code describes a physical design friendly memory-bank circuitry to configure the core logic of FPGA, as illustrated in Fig. 7.6. It will use the circuit model defined in Fig. 7.24.

The BL and WL protocols can be customized through the XML syntax `bl` and `wl`.

Note: If not specified, the BL/WL protocols will use decoders.

```
<configuration_protocol>
  <organization type="ql_memory_bank" circuit_model_name="sram_blwl">
    <bl protocol="<string>" num_banks="<int>"/>
    <wl protocol="<string>" num_banks="<int>"/>
  </organization>
</configuration_protocol>
```

protocol="decoder|flatten|shift_register"

- `decoder`: BLs or WLs are controlled by decoders with address lines. For BLs, the decoder includes an enable signal as well as a data input signal. This is the default option if not specified. See an illustrative example in Fig. 7.7.
- `flatten`: BLs or WLs are directly available at the FPGA fabric. In this way, all the configurable memorys on the same WL can be written through the BL signals in one clock cycle. See an illustrative example in Fig. 7.8.
- `shift_register`: BLs or WLs are controlled by shift register chains. The BL/WLs are programming each time the shift register chains are fully loaded. See an illustrative example in Fig. 7.9.

num_banks="<int>"

Specify the number of shift register banks (i.e., independent shift register chains) to be used in each configuration region. When enabled, the length of each shift register chain will be sized by OpenFPGA automatically based on the number of BL/WLs in each configuration region. OpenFPGA will try to create similar sizes for the shift register chains, in order to minimize the number of HDL modules. If not specified, the default number of banks will be 1.

Note: This is available applicable to shift-register-based BL/WL protocols

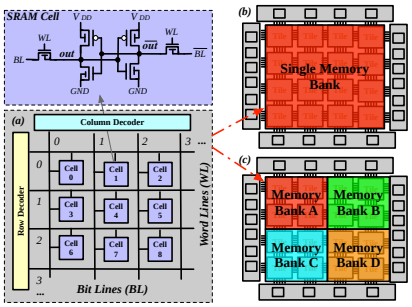


Fig. 7.7: Example of (a) a memory organization using address decoders; (b) single memory bank across the fabric; and (c) multiple memory banks across the fabric.

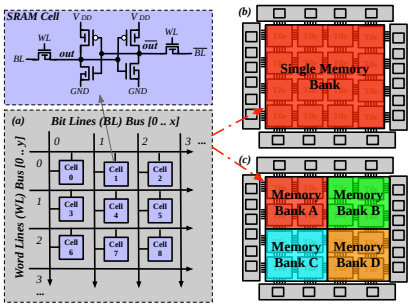


Fig. 7.8: Example of (a) a memory organization with direct access to BL/WL signals; (b) single memory bank across the fabric; and (c) multiple memory banks across the fabric.

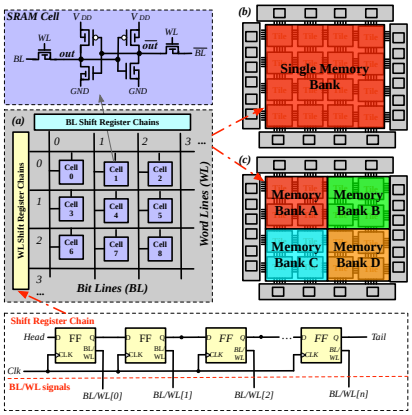


Fig. 7.9: Example of (a) a memory organization using shift register chains to control BL/WLs; (b) single memory bank across the fabric; and (c) multiple memory banks across the fabric.

Note: More customization on the shift register chains can be enabled through *Fabric Key*

Note: The flip-flop for WL shift register requires an enable signal to gate WL signals when loading WL shift registers

Note: Memory-bank decoders does require a memory cell to have

- two outputs (one regular and another inverted)
- a Bit-Line input to load the data
- a Word-Line input to enable data write
- (optional) a Word-Line read input to enabe data readback

Warning: Please do NOT add inverted Bit-Line and Word-Line inputs. It is not supported yet!

7.3.6 Standalone SRAM Example

In the standalone configuration protocol, every memory cell of the core logic of a FPGA fabric can be directly accessed at the top-level module, as illustrated in Fig. 7.10.

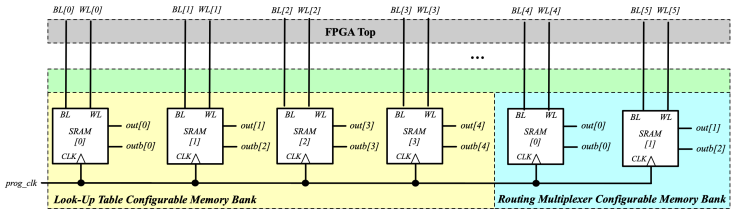


Fig. 7.10: Vanilla (standalone) memory organization in a hierarchical view

The following XML code shows an example where we use the circuit model defined in Fig. 7.24.

```
<configuration_protocol>
  <organization type="standalone" circuit_model_name="sram_blwl"/>
</configuration_protocol>
```

Note: The standalone protocol does require a memory cell to have

- two outputs (one regular and another inverted)
- a Bit-Line input to load the data
- a Word-Line input to enable data write

Warning: Please do NOT add inverted Bit-Line and Word-Line inputs. It is not supported yet!

Warning: This is a vanilla configuration method, which allow users to build their own configuration protocol on top of it.

7.4 Inter-Tile Direct Interconnection extensions

This section introduces extensions on the architecture description file about existing interconnection description.

7.4.1 Directlist

The original direct connections in the directlist section are documented here. Its description is given below:

```
<directlist>
  <direct name="string" from_pin="string" to_pin="string" x_offset="int" y_offset="int"␣
↪z_offset="int" switch_name="string"/>
</directlist>
```

Note: These options are required

Our extension include three more options:

```
<directlist>
  <direct name="string" from_pin="string" to_pin="string" x_offset="int" y_offset="int"␣
↪z_offset="int" switch_name="string" interconnection_type="string" x_dir="string" y_dir=
↪"string"/>
</directlist>
```

Note: these options are optional. However, if *interconnection_type* is set *x_dir* and *y_dir* are required.

interconnection_type="<string>"

the type of interconnection should be a string. Available types are `NONE` | `column` | `row`, specifies if it applies on a column or a row ot if it doesn't apply.

x_dir="<string>"

Available directionalities are `positive` | `negative`, specifies if the next cell to connect has a bigger or lower `x` value. Considering a coordinate system where (0,0) is the origin at the bottom left and `x` and `y` are positives:

- x_dir="positive":
  - interconnection_type="column": a column will be connected to a column on the `right`, if it exists.
  - interconnection_type="row": the most on the `right` cell from a row connection will connect the most on the `left` cell of next row, if it exists.
- x_dir="negative":
  - interconnection_type="column": a column will be connected to a column on the `left`, if it exists.
  - interconnection_type="row": the most on the `left` cell from a row connection will connect the most on the `right` cell of next row, if it exists.

y_dir="<string>"

Available directionalities are `positive` | `negative`, specifies if the next cell to connect has a bigger or lower x value. Considering a coordinate system where (0,0) is the origin at the bottom left and *x* and *y* are positives:

- y_dir="positive":
  - interconnection_type="column": the `bottom` cell of a column will be connected to the next column `top` cell, if it exists.
  - interconnection_type="row": a row will be connected on an `above` row, if it exists.
- y_dir="negative":
  - interconnection_type="column": the `top` cell of a column will be connected to the next column `bottom` cell, if it exists.
  - interconnection_type="row": a row will be connected on a row `below`, if it exists.

7.4.2 Example

For this example, we will study a scan-chain implementation. The description could be:

```
<directlist>
  <direct name="scff_chain" from_pin="clb.sc_out" to_pin="clb.sc_in" x_offset="0" y_
↪offset="-1" z_offset="0" interconnection_type="column" x_dir="positive" y_dir="positive
↪"/>
</directlist>
```

Fig. 7.11 is the graphical representation of the above scan-chain description on a 4x4 FPGA.

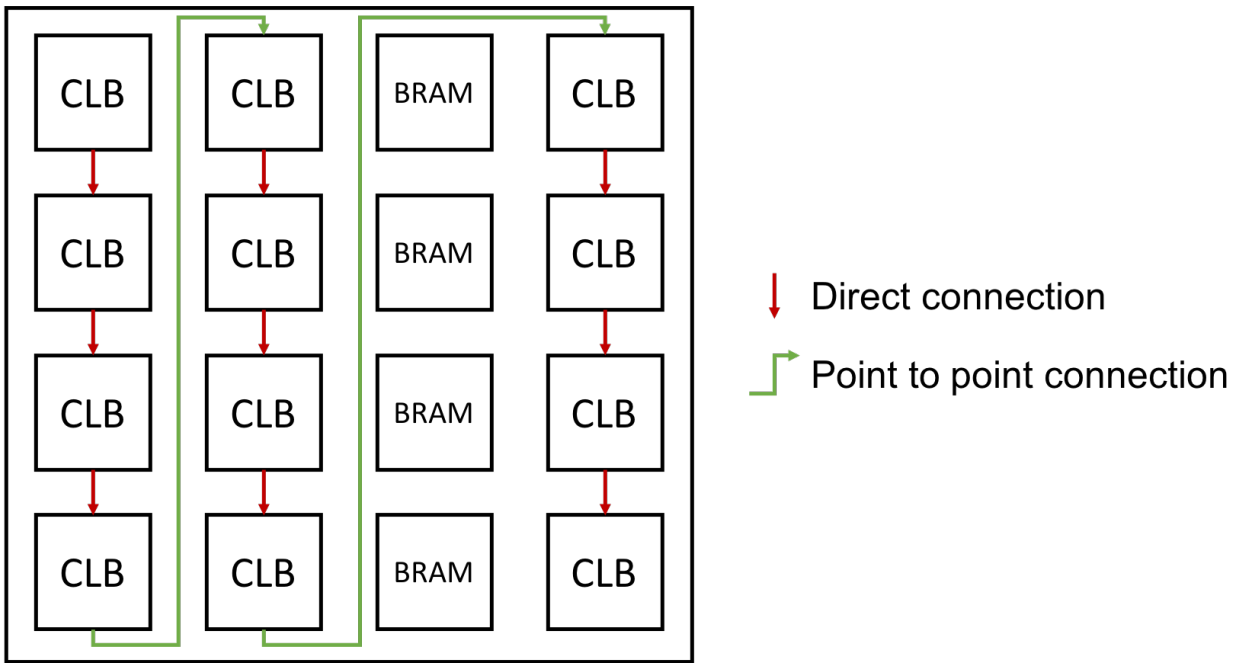


Fig. 7.11: An example of scan-chain implementation

In this figure, the red arrows represent the initial direct connection. The green arrows represent the point to point connection to connect all the columns of CLB.

7.4.3 Truth table

A point to point connection can be applied in different ways than showed in the example section. To help the designer implement his point to point connection, a truth table with our new parameters id provided below.

Fig. 7.12 provides all possible variable combination and the connection it will generate.

Connection	interconnection_type	x_dir	y_dir
	column	positive	positive
	column	positive	negative
	column	negative	positive
	column	negative	negative
	row	positive	positive
	row	positive	negative
	row	negative	positive
	row	negative	negative

Fig. 7.12: Point to point truth table

7.5 Simulation settings

All the simulation settings are stored under the XML node <openfpga_simulation_setting> General organization is as follows

```
<openfpga_simulation_setting>
  <clock_setting>
    <operating frequency="<int>|<string>" num_cycles="<int>|<string>" slack="<float>">
      <clock name="<string>" port="<string>" frequency="<float>"/>
      ...
    </operating>
    <programming frequency="<int>">
      <clock name="<string>" port="<string>" frequency="auto|<float>" is_shift_register="
↪<bool>"/>
      ...
    </programming>
  </clock_setting>
```

(continues on next page)

(continued from previous page)

```
  <simulator_option>
    <operating_condition temperature="<int>"/>
    <output_log verbose="<bool>" captab="<bool>"/>
    <accuracy type="<string>" value="<float>"/>
    <runtime fast_simulation="<bool>"/>
  </simulator_option>
  <monte_carlo num_simulation_points="<int>"/>
  <measurement_setting>
    <slew>
      <rise upper_thres_pct="<float>" lower_thres_pct="<float>"/>
      <fall upper_thres_pct="<float>" lower_thres_pct="<float>"/>
    </slew>
    <delay>
      <rise input_thres_pct="<float>" output_thres_pct="<float>"/>
      <fall input_thres_pct="<float>" output_thres_pct="<float>"/>
    </delay>
  </measurement_setting>
  <stimulus>
    <clock>
      <rise slew_type="<string>" slew_time="<float>"/>
      <fall slew_type="<string>" slew_time="<float>"/>
    </clock>
    <input>
      <rise slew_type="<string>" slew_time="<float>"/>
      <fall slew_type="<string>" slew_time="<float>"/>
    </input>
  </stimulus>
</openfpga_simulation_setting>
```

7.5.1 Clock Setting

Clock setting focuses on defining the clock periods to applied on FPGA fabrics As a programmable device, an FPGA has two types of clocks. The first is the operating clock, which is applied by users' implementations. The second is the programming clock, which is applied on the configuration protocol to load users' implementation to FPGA fabric. OpenFPGA allows users to freely define these clocks as well as the number of clock cycles. We should the full syntax in the code block below and then provide details on each of them.

```
<clock_setting>
  <operating frequency="<float>|<string>" num_cycles="<int>|<string>" slack="<float>">
    <clock name="<string>" port="<string>" frequency="<float>"/>
    ...
  </operating>
  <programming frequency="<float>">
    <clock name="<string>" port="<string>" frequency="auto|<float>" is_shift_register="
↪<bool>"/>
    ...
  </programming>
</clock_setting>
```

Operating clock setting

Operating clocks are defined under the XML node <operating> To support FPGA fabrics with multiple clocks, OpenFPGA allows users to define a default operating clock frequency as well as a set of clock ports using different frequencies.

<operating frequency="<float>|<string>" num_cycles="<int>|<string>" slack="<float>"/>

- frequency="<float|string> Specify frequency of the operating clock. OpenFPGA allows users to specify an absolute value in the unit of [Hz] Alternatively, users can bind the frequency to the maximum clock frequency analyzed by VPR STA engine. This is very useful to validate the maximum operating frequency for users' implementations In such case, the value of this attribute should be a reserved word auto.

Note: The frequency is considered as a default operating clock frequency, which will be used when a clock pin of a multi-clock FPGA fabric lacks explicit clock definition.

- num_cycles="<int>|<string>" can be either auto or an integer. When set to auto, OpenFPGA will infer the number of clock cycles from the average/median of all the signal activities. When set to an integer, OpenFPGA will use the given number of clock cycles in HDL and SPICE simulations.
- slack="<float>" add a margin to the critical path delay in the HDL and SPICE simulations. This parameter is applied to the critical path delay provided by VPR STA engine. So it is only valid when option frequency is set to auto. This aims to compensate any inaccuracy in STA results. Typically, the slack value is between 0 and 1. For example, slack=0.2 implies that the actual clock period in simulations is 120% of the critical path delay reported by VPR.

Note: Only valid when option frequency is set to auto

Warning: Avoid to use a negative slack! This may cause your simulation to fail!

<clock name="<string>" port="<string>" frequency="<float>"/>

- name="<string> Specify a unique name for a clock signal. The name will be used in generating clock stimulus in testbenches.
- port="<string> Specify the clock port which the clock signal should be applied to. The clock port must be a valid clock port defined in OpenFPGA architecture description. Explicit index is required, e.g., clk[1:1]. Otherwise, default index 0 will be considered, e.g., clk will be translated as clk[0:0].

Note: You can define clock ports either through the tile annotation in *Physical Tile Annotation* or *Circuit Port*.

- frequency="<float> Specify frequency of a clock signal in the unit of [Hz]

Warning: Currently, we only allow operating clocks to be overwritten!!!

Programming clock setting

Programming clocks are defined under the XML node <programming>

<programming frequency="<float>"/>

- frequency="<float>" Specify the frequency of the programming clock using an absolute value in the unit of [Hz] This frequency is used in testbenches for programming phase simulation.

<clock name="<string>" port="<string>" frequency="auto|<float>" is_shift_register="<bool>"/
>

- name="<string> Specify a unique name for a clock signal. The name should match a reserved word of programming clock, i.e., bl_sr_clock and wl_sr_clock.

  Note: The bl_sr_clock represents the clock signal driving the BL shift register chains, while the wl_sr_clock represents the clock signal driving the WL shift register chains

- port="<string> Specify the clock port which the clock signal should be applied to. The clock port must be a valid clock port defined in OpenFPGA architecture description. Explicit index is required, e.g., clk[1:1]. Otherwise, default index 0 will be considered, e.g., clk will be translated as clk[0:0].
- frequency="auto|<float> Specify frequency of a clock signal in the unit of [Hz]. If auto is used, the programming clock frequency will be inferred by OpenFPGA.
- is_shift_register="<bool> Specify if this clock signal is used to drive shift register chains in BL/WL protocols

Note: Programming clock frequency is typically much slower than the operating clock and strongly depends on the process technology. Suggest to characterize the speed of your configuration protocols before specifying a value!

7.5.2 Simulator Option

This XML node includes universal options available in both HDL and SPICE simulators.

Note: This is mainly used by FPGA-SPICE

Operating condition

<operating_condition temperature="<int>"/>``

- temperature="<int>" Specify the temperature which will be defined in SPICE netlists. In the top SPICE netlists, it will show as

```
.temp <int>
```

Output logs

<output_log verbose="<bool>" captab="<bool>"/>``

Specify the options in outputting simulation results to log files

- verbose="true|false"

  Specify if the simulation waveforms should be printed out after SPICE simulations. If turned on, it will show in all the SPICE netlists

```
.option POST
```

Note: when the SPICE netlists are large or a long simulation duration is defined, the post option is recommended to be off. If not, huge disk space will be occupied by the waveform files.

- captab="true|false" Specify if the capacitances of all the nodes in the SPICE netlists will be printed out. If turned on, it will show in the top-level SPICE netlists

```
.option CAPTAB
```

Note: When turned on, the SPICE simulation runtime may increase.

Simulation Accuracy

<accuracy type="<string>" value="<float>"/>``

Specify the simulation steps (accuracy) to be used

- type="abs|frac"

  Specify the type of transient step in SPICE simulation.

  - When abs is selected, the accuracy should be the absolute value, such as 1e-12.
  - When frac is selected, the accuracy is the number of simulation points in a clock cycle period, for example, 100.

- value="<float>"

  Specify the transient step in SPICE simulation. Typically, the smaller the step is, the higher the accuracy that can be reached while the long simulation runtime is. The recommended accuracy is between 0.1ps and 0.01ps, which generates good accuracy and runtime is not significantly long.

Simulation Speed

<runtime fast_simulation="<bool>"/>

Specify if any runtime optimization will be applied to the simulator.

- fast_simulation="true|false"

  Specify if fast simulation is turned on for the simulator.

  If turned on, it will show in the top-level SPICE netlists

```
.option fast
```

7.5.3 Monte Carlo Simulation

<monte_carlo num_simulation_points="<int>"/>

Run SPICE simulations in monte carlo mode. This is mainly for FPGA-SPICE When turned on, FPGA-SPICE will apply the device variation defined in *Technology library* to monte carlo simulation

- num_simulation_points="<int>"

  Specify the number of simulation points to be considered in monte carlo. The larger the number is, the longer simulation time will be but more accurate the results will be.

7.5.4 Measurement Setting

- Users can define the parameters in measuring the slew of signals, under XML node <slew>
- Users can define the parameters in measuring the delay of signals, under XML node <delay>

Both delay and slew measurement share the same syntax in defining the upper and lower voltage thresholds.

<rise|fall upper_thres_pct="<float>" lower_thres_pct="<float>"/>

Define the starting and ending point in measuring the slew of a rising or a falling edge of a signal.

- upper_thres_pct="<float>" the ending point in measuring the slew of a rising edge. It is expressed as a percentage of the maximum voltage of a signal. For example, the meaning of upper_thres_pct=0.95 is depicted in Fig. 7.13.
- lower_thres_pct="<float>" the starting point in measuring the slew of a rising edge. It is expressed as a percentage of the maximum voltage of a signal. For example, the meaning of lower_thres_pct=0.05 is depicted in Fig. 7.13.

7.5.5 Stimulus Setting

Users can define the slew time of input and clock signals to be applied to FPGA I/Os in testbenches under XML node <clock> and <input> respectively. This is used by FPGA-SPICE in generating testbenches

<rise|fall slew_type="<string>" slew_time="<float>"/>

Specify the slew rate of an input or clock signal at rising or falling edge

- slew_type="[abs|frac]" specify the type of slew time definition at the rising or falling edge of a lock/input port.
  - The type of abs implies that the slew time is the absolute value. For example, slew_type="abs" slew_time="20e-12" means that the slew of a clock signal is 20ps.
  - The type of frac means that the slew time is related to the period (frequency) of the clock signal. For example, slew_type="frac" slew_time="0.05" means that the slew of a clock signal takes 5% of the period of the clock.
- slew_time="<float>" specify the slew rate of an input or clock signal at the rising/falling edge.

Fig. 7.13 depicts the definition of the slew and delays of signals and the parameters that can be supported by FPGA-SPICE.

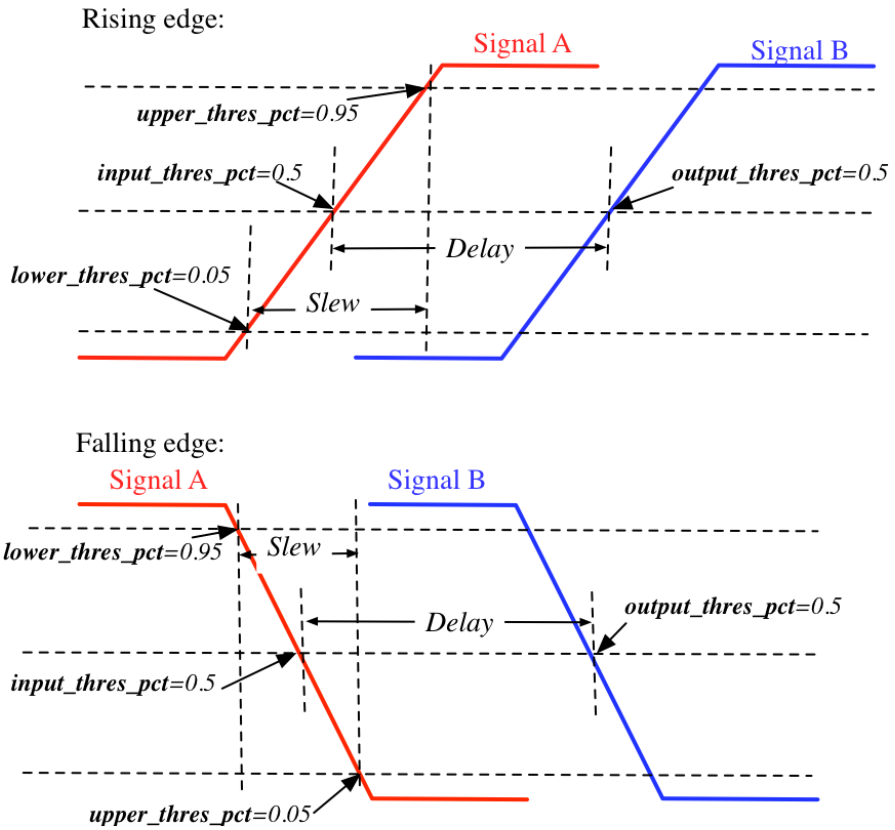


Fig. 7.13: An illustrative example on measuring the slew and delay of signals

7.6 Technology library

Technology library aims to describe transistor-level parameters to be applied to the physical design of FPGAs. In addition to transistor models, technology library also supports the definition of process variations on any transistor models. General organization is as follows.

```
<technology_library>
  <device_library>
    <device_model name="<string>" type="<string>">
      <lib type="<string>" corner="<string>" ref="<string>" path="<string>"/>
      <design vdd="<float>" pn_ratio="<float>"/>
      <pmos name="<string>" chan_length="<float>" min_width="<float>" max_width="<float>
↪" variation="<string>"/>
      <nmos name="<string>" chan_length="<float>" min_width="<float>" max_width="<float>
↪" variation="<string>"/>
      <rram rlrs="<float>" rhrs="<float>" variation="<string>"/>
    </device_model>
  </device_library>
  <variation_library>
    <variation name="<string>" abs_deviation="<float>" num_sigma="<int>"/>
  </variation_library>
</technology_library>
```

7.6.1 Device Library

Device library contains detailed description on device models, such as transistors and Resistive Random Access Memories (RRAMs). A device library may consist of a number of <device_model> and each of them denotes a different transistor model.

A device model represents a transistor/RRAM model available in users' technology library.

<device_model name="<string>" type="<string>">

Specify the name and type of a device model

- name="<string>" is the unique name of the device model in the context of <device_library>.
- type="transistor|rram" is the type of device model in terms of functionality Currently, OpenFPGA supports two types: transistor and RRAM.

Note: the name of <device_model> may not be the name in users' technology library.

<lib type="<string>" corner="<string>" ref="<string>" path="<string>"/>

Specify the technology library that defines the device model

- type="academia|industry" For the industry library, FPGA-SPICE will use .lib <lib_file_path> to include the library file in SPICE netlists. For academia library, FPGA-SPICE will use .include <lib_file_path> to include the library file in SPICE netlists
- corner="<string>" is the process corner name available in technology library. For example, the type of transistors can be TT, SS and FF *etc*.
- ref="<string>" specify the reference of in calling a transistor model. In SPICE netlists, define a transistor follows the convention:

```
<model_ref><trans_name> <ports> <model_name>
```

The reference depends on the technology and the type of library. For example, the PTM bulk model uses "M" as the reference while the PTM FinFET model uses "X" as the reference.

- path="<string>" specify the path of the technology library file. For example:

```
lib_path=/home/tech/45nm.pm.
```

<design vdd="<float>" pn_ratio="<float>"/>

Specify transistor-level design parameters

- vdd="<float>" specify the working voltage for the technology. The voltage will be used as the supply voltage in all the SPICE netlists.
- pn_ratio="<float>" specify the ratio between *p*-type and *n*-type transistors. The ratio will be used when building circuit structures such as inverters, buffers, etc.

<pmos|nmos name="<string>" chan_length="<float>" min_width="<float>" max_width="<float>" variation="<st
>

Specify device-level parameters for transistors

- name="<string>" specify the name of the p/n type transistor, which can be found in the manual of the technology provider.
- chan_length="<float>" specify the channel length of a *p/n* type transistor.
- min_width="<float>" specify the minimum width of a *p/n* type transistor. This parameter will be used in building inverter, buffer, *etc*. as a base number for transistor sizing.
- max_width="<float>" specify the maximum width of a *p/n* type transistor. This parameter will be used in building inverter, buffer, *etc*. as a base number for transistor sizing. If the required transistor width exceeds the maximum width, multiple transistors will be instanciated. Note that for FinFET technology, your max_width should be the same as your min_width.

  Note: The max_width is optional. By default, it will be set to be same as the min_width.

- variation="<string>" specify the variation name defined in the <variation_library>

<rram rlrs="<float>" rhrs="<float>" variation="<string>"/>

Specify device-level parameters for RRAMs

- rlrs="<float>" specify the resistance of Low Resistance State (LRS) of a RRAM device
- rhrs="<float>" specify the resistance of High Resistance State (HRS) of a RRAM device
- variation="<string>" specify the variation name defined in the <variation_library>

7.6.2 Variation Library

Variation library contains detailed description on device variations specified by users. A variation library may consist of a number of <variation> and each of them denotes a different variation parameter.

<variation name="<string>" abs_deviation="<float>" num_sigma="<int>"/>

Specify detail variation parameters

- name="<string>" is the unique name of the device variation in the context of <variation_library>. The name will be used in <device_model> to bind variations
- abs_variation="<float>" is the absolute deviation of a variation
- num_sigma="<int>" is the standard deviation of a variation

7.7 Circuit Library

Circuit design is a dominant factor in Power, Performance, Area (P.P.A.) of FPGA fabrics. Upon practical applications, the hardware engineers may select various circuits to implement their FPGA fabrics. For instance, a ultra-low-power FPGA may be built with ulta-low-power circuit cells while a high-performance FPGA may use absolutely different circuit cells. OpenFPGA provide enriched XML syntax for users to highly customize their circuits in FPGA fabric.

In the XML file, users can define a library of circuits, each of which corresponds to a primitive module required in the FPGA architecture. Users can specify if the Verilog/SPICE netlist of the module is either auto-generated by OpenFPGA or provided by themselves. As such, OpenFPGA can support any circuit design, leading to high flexibility in building FPGA fabrics.

In principle, a circuit library consists of a number of <circuit_model>, each of which correspond to a circuit design. OpenFPGA supports a wide range of circuit designs. The <circuit_model> could be as small as a cornerstone cell, such as inverter, buffer *etc*., or as large as a hardware IP, such as Block RAM.

```
<circuit_library>
  <circuit_model type="<string>" name="<string>">
    <!-- Detailed circuit-level design parameters -->
  </circuit_model>
  <!-- More circuit models -->
</circuit_library>
```

Currently, OpenFPGA supports the following categories of circuits:

- inverters/buffers
- pass-gate logic, including transmission gates and pass transistors
- standard cell logic gates, including AND, OR and MUX2
- metal wires
- multiplexers
- flip-flops
- Look-Up Tables, including single-output and multi-output fracturable LUTs
- Statis Random Access Memory (SRAM)
- scan-chain flip-flops
- I/O pad
- hardware IPs

7.7.1 Circuit Model

As OpenFPGA supports many types of circuit models and their circuit-level implementation could be really different, each type of circuit model has special syntax to customize their designs. However, most circuit models share the common generality in XML language. Here, we focus these common syntax and we will detail special syntax in *Circuit model examples*

```
<circuit_model type="<string>" name="<string>" prefix="<string>" is_default="<bool>"␣
↪spice_netlist="<string>" verilog_netlist="<string>" dump_structural_verilog="<bool>">
  <design_technology type="<string>"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <pass_gate_logic type="<string>" circuit_model_name="<string>"/>
  <port type="<string>" prefix="<string>" lib_name="<string>" size="<int>" default_val="
↪<int>" circuit_model_name="<string>" mode_select="<bool>" is_global="<bool>" is_set="
↪<bool>" is_reset="<bool>" is_config_enable="<bool>"/>
  <!-- more ports -->
</circuit_model>
```

<circuit_model type="<string>" name="<string>" prefix="<string>" is_default="<bool>"
spice_netlist="<string>" verilog_netlist="<string>" dump_structural_verilog="<bool>">

Specify the general attributes for a circuit model

- type="inv_buf|pass_gate|gate|mux|wire|chan_wire|sram|lut|ff|ccff|hard_logic|iopad" Specify the type of circuit model. For the circuit models in the type of mux/wire/chan_wire/lut, FPGA-Verilog/SPICE can auto-generate Verilog/SPICE netlists. For the rest, FPGA-Verilog/SPICE requires a user-defined Verilog/SPICE netlist.
- name="<string>" Specify the name of this circuit model. The name should be unique and will be used to create the Verilog/SPICE module in Verilog/SPICE netlists. Note that for a customized Verilog/SPICE netlist, the name defined here MUST be the name in the customized Verilog/SPICE netlist. FPGA-Verilog/SPICE will check if the given name is conflicted with any reserved words.
- prefix="<string>" Specify the name of the <circuit_model> to shown in the auto-generated Verilog/SPICE netlists. The prefix can be the same as the name defined above. And again, the prefix should be unique
- is_default="true|false" Specify this circuit model is the default one for those in the same types. If a primitive module in VPR architecture is not linked to any circuit model by users, FPGA-Verilog/SPICE will find the default circuit model defined in the same type.
- spice_netlist="<string>" Specify the path and file name of a customized SPICE netlist. For some modules such as SRAMs, FFs, I/O pads, FPGA-SPICE does not support auto-generation of the transistor-level sub-circuits because their circuit design is highly dependent on the technology nodes. These circuit designs should be specified by users. For the other modules that can be auto-generated by FPGA-SPICE, the user can also define a custom netlist.
- verilog_netlist="<string>" Specify the path and file name of a customized Verilog netlist. For some modules such as SRAMs, FFs, I/O pads, FPGA-Verilog does not support auto-generation of the transistor-level sub-circuits because their circuit design is highly dependent on the technology nodes. These circuit designs should be specified by users. For the other modules that can be auto-generated by FPGA-Verilog, the user can also define a custom netlist.
- dump_structural_verilog="true|false" When the value of this keyword is set to be true, Verilog generator will output gate-level netlists of this module, instead of behavior-level. Gate-level netlists bring more opportunities in layout-level optimization while behavior-level is more suitable for high-speed formal verification and easier in debugging with HDL simulators.

Warning: `prefix` may be deprecated soon

Warning: Multiplexers cannot be user-defined.

Warning: For a circuit model type, only one circuit model is allowed to be set as default. If there is only one circuit model defined in a type, it will be considered as the default automatically.

Note: If <spice_netlist> or <verilog_netlist> are not specified, FPGA-Verilog/SPICE auto-generates the Verilog/SPICE netlists for multiplexers, wires, and LUTs.

Note: The user-defined netlists, such as LUTs, the decoding methodology should comply with the auto-generated LUTs!!!

7.7.2 Design Technology

<design_technology type="string"/>

Specify the design technology applied to a <circuit_model>

- type="cmos|rram" Specify the type of design technology of the <circuit_model>. Currently, OpenFPGA supports CMOS and RRAM technology for circuit models. CMOS technology can be applied to any types of <circuit_model>, while RRAM technology is only applicable to multiplexers and SRAMs

Note: Each <circuit_model> may have different technologies

7.7.3 Device Technology

<device_technology device_model_name="<string>"/>

Specify the technology binding between a circuit model and a device model which is defined in the technology library (see details in *Technology library*).

- device_model_name="<string>" Specify the name of device model that the circuit design will use. The device model must be a valid one in the technology library.

Note: Technology binding is only required for primitive circuit models, which are inverters, buffers, logic gates, pass gate logic, and is mandatory only when SPICE netlist generation is required.

7.7.4 Input and Output Buffers

<input_buffer exist="<string>" circuit_model_name="<string>"/>

- exist="true|false" Define the existence of the input buffer. Note that the existence is valid for all the inputs.
- circuit_model_name="<string>" Specify the name of circuit model which is used to implement input buffer, the type of specified circuit model should be inv_buf.

<output_buffer exist="<string>" circuit_model_name="<string>"/>

- exist="true|false" Define the existence of the output buffer. Note that the existence is valid for all the outputs. Note that if users want only part of the inputs (or outputs) to be buffered, this is not supported here. A solution can be building a user-defined Verilog/SPICE netlist.
- circuit_model_name="<string>" Specify the name of circuit model which is used to implement the output buffer, the type of specified circuit model should be inv_buf.

Note: If users want only part of the inputs (or outputs) to be buffered, this is not supported here. A solution can be building a user-defined Verilog/SPICE netlist.

7.7.5 Pass Gate Logic

<pass_gate_logic circuit_model_name="<string>"/>

- circuit_model_name="<string>" Specify the name of the circuit model which is used to implement pass-gate logic, the type of specified circuit model should be pass_gate.

Note: pass-gate logic are used in building multiplexers and LUTs.

7.7.6 Circuit Port

A circuit model may consist of a number of ports. The port list is mandatory in any circuit_model and must be consistent to any user-defined netlists.

<port type="<string>" prefix="<string>" lib_name="<string>" size="<int>"
default_val="<int>" circuit_model_name="<string>" mode_select="<bool>"
is_global="<bool>" is_set="<bool>" is_reset="<bool>"
is_config_enable="<bool>" is_io="<bool>" is_data_io="<bool>"/>

Define the attributes for a port of a circuit model.

- type="input|output|sram|clock" Specify the type of the port, i.e., the directionality and usage. For programmable modules, such as multiplexers and LUTs, SRAM ports MUST be defined. For registers, such as FFs and memory banks, clock ports MUST be defined.

  Note: sram and clock ports are considered as inputs in terms of directionality

  ---

- prefix="<string>" the name of the port to appear in the autogenerated netlists. Each port will be shown as <prefix>[i] in Verilog/SPICE netlists.

  Note: if the circuit model is binded to a pb_type in VPR architecture, prefix must match the port name defined in pb_type

- lib_name="<string>" the name of the port defined in standard cells or customized cells. If not specified, this attribute will be the same as prefix.

  Note: if the circuit model comes from a standard cell library, using lib_name is recommended. This is because - the port names defined in pb_type are very diffrerent from the standard cells - the port sequence is very different

- size="<int>" bandwidth of the port. MUST be larger than zero.
- default_val="<int>" Specify default logic value for a port, which is used as the initial logic value of this port in testbench generation. Can be either 0 or 1. We assume each pin of this port has the same default value.
- circuit_model_name="<string>" Specify the name of the circuit model which is connected to this port.

  Note: circuit_model_name is only valid when the type of this port is sram.

- is_io="true|false" Specify if this port should be treated as an I/O port of an FPGA fabric. When this is enabled, this port of each circuit model instanciated in FPGA will be added as an I/O of an FPGA.

  Note: global output ports must be io ports

- is_data_io="true|false" Specify if this port should be treated as a mappable FPGA I/O port for users' implementation. When this is enabled, I/Os of user's implementation, e.g., .input and .output in .blif netlist, can be mapped to the port through VPR.

  Note: Any I/O model must have at least 1 port that is defined as data I/O!

- mode_select="true|false" Specify if this port controls the mode switching in a configurable logic block. This is due to that a configurable logic block can operate in different modes, which is controlled by SRAM bits.

  Note: mode_select is only valid when the type of this port is sram.

- is_global="true|false" can be either true or false. Specify if this port is a global port, which will be routed globally.

  Note: For input ports, when multiple global input ports are defined with the same name, by default, these global ports will be short-wired together. When io is turned on for this port, these global ports will be independent in the FPGA fabric.

Note: For output ports, the global ports will be independent in the FPGA fabric

- `is_set="true|false"` Specify if this port controls a set signal. All the set ports are connected to global set voltage stimuli in testbenches.
- `is_reset="true|false"` Specify if this port controls a reset signal. All the reset ports are connected to a global reset voltage stimuli in testbenches.
- `is_config_enable="true|false"` Specify if this port controls a configuration-enable signal. Only valid when `is_global` is `true`. This port is only enabled during FPGA configuration, and always disabled during FPGA operation. All the `config_enable` ports are connected to global configuration-enable voltage stimuli in testbenches.

Note: This attribute is used by testbench generators (see *Testbench*)

- In full testbench,
  - There is a `config_done` signal, which stay at logic `0` during bitstream loading phase, and is pulled up to logic `1` during operating phase
  - When `default_value="0"`, the port will be wired to a `config_done` signal.
  - When `default_value="1"`, the port will be wired to an inverted `config_done` signal.
- In preconfigured wrapper, the port will be set to the inversion of `default_value`, as the preconfigured testbenches consider operating phase only.

Note: `is_set`, `is_reset` and `is_config_enable` are only valid when `is_global` is `true`.

Note: Different types of `circuit_model` have different XML syntax, with which users can highly customize their circuit topologies. See refer to examples of :ref:`circuit_model_example` for more details.

Note: Note that we have a list of reserved port names, which indicate the usage of these ports when building FPGA fabrics. Please do not use `mem_out`, `mem_inv`, `bl`, `wl`, `blb`, `wlb`, `wlr`, `ccff_head` and `ccff_tail`.

7.7.7 FPGA I/O Port

The `circuit_model` support not only highly customizable circuit-level modeling but also flexible I/O connection in the FPGA fabric. Typically, circuit ports appear in the primitive modules of a FPGA fabric. However, it is also very common that some circuit ports should be I/O of a FPGA fabric. Using syntax `is_global` and `is_io`, users can freely define how these ports are connected as FPGA I/Os.

In principle, when `is_global` is set `true`, the port will appear as an FPGA I/O. The syntax `is_io` is applicable when `is_global` is `true`. When `is_io` is `true`, the port from different instances will be treated as independent I/Os. When `is_io` is `false`, the port from different instances will be treated as the same I/Os, which are short-wired.

To beef up, the following examples will explain how to use `is_global` and `is_io` to achieve different types of connections to FPGA I/Os.

Global short-wired inputs

```
<port type="input" is_global="true" is_io="false"/>
```

The global inputs are short wired across different instances. These inputs are widely seen in FPGAs, such as clock ports, which are shared between sequential elements.

Fig. 7.14 shows an example on how the global inputs are wired inside FPGA fabric.

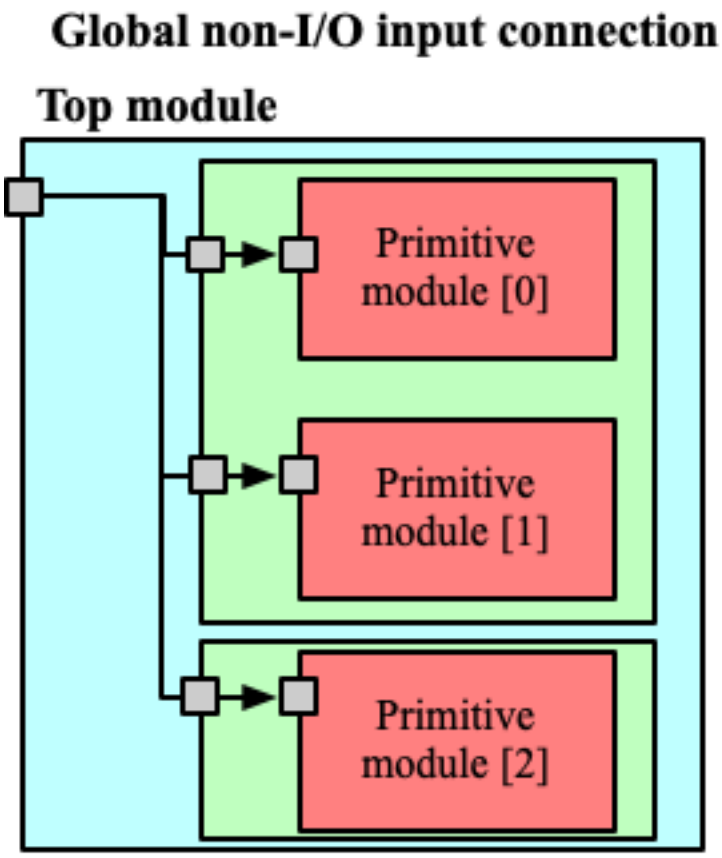


Fig. 7.14: Short-wired global inputs as an FPGA I/O

Global short-wired inouts

```
<port type="inout" is_global="true" is_io="false"/>
```

The global inouts are short wired across different instances.

Fig. 7.15 shows an example on how the global inouts are wired inside FPGA fabric.

General-purpose inputs

```
<port type="input" is_global="true" is_io="true"/>
```

The general-purpose inputs are independent wired from different instances to separated FPGA I/Os. For example, power-gating signals can be applied to each tile of a FPGA.

Fig. 7.16 shows an example on how the general-purpose inputs are wired inside FPGA fabric.

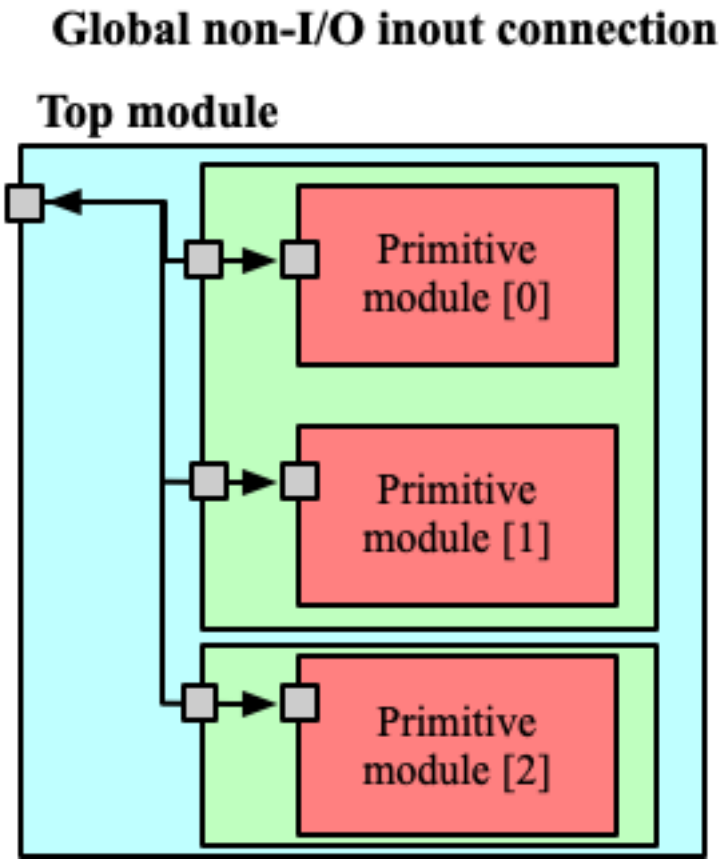


Fig. 7.15: Short-wired global inouts as an FPGA I/O

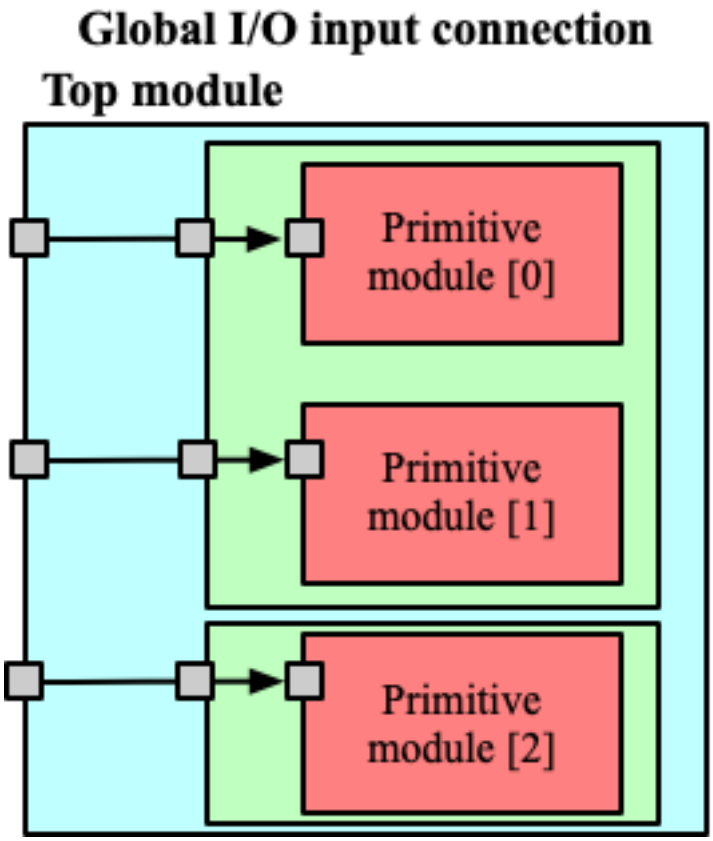


Fig. 7.16: General-purpose inputs as separated FPGA I/Os

General-purpose I/O

```
<port type="inout" is_global="true" is_io="true"/>
```

The general-purpose I/O are independent wired from different instances to separated FPGA I/Os. In practice, inout of GPIO cell is typically wired like this.

Fig. 7.16 shows an example on how the general-purpose inouts are wired inside FPGA fabric.

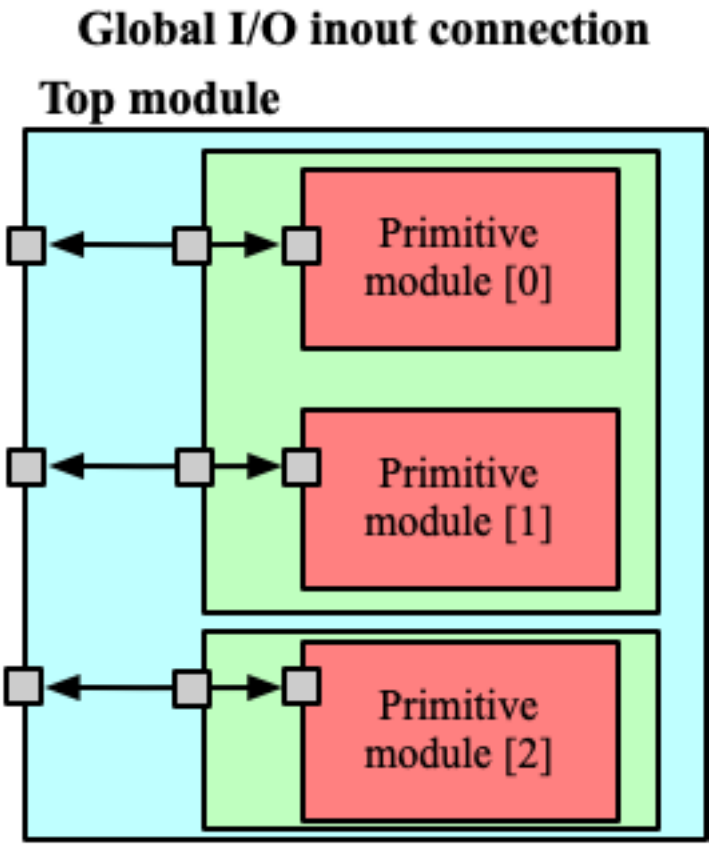


Fig. 7.17: General-purpose inouts as separated FPGA I/Os

General-purpose outputs

```
<port type="output" is_global="true" is_io="true"/>
```

The general-purpose outputs are independent wired from different instances to separated FPGA outputs. In practice, these outputs are typically spypads to probe internal signals of a FPGA.

Fig. 7.18 shows an example on how the general-purpose outputs are wired inside FPGA fabric.

Warning: The general-purpose inputs/inouts/outputs are not applicable to routing multiplexer outputs

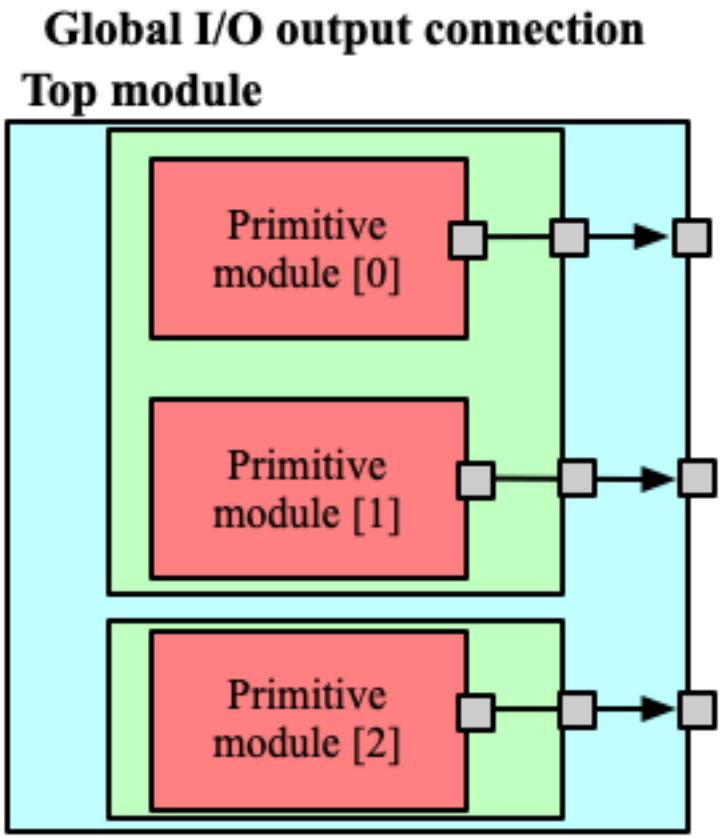


Fig. 7.18: General-purpose outputs as separated FPGA I/Os

7.8 Circuit model examples

As circuit model in different types have various special syntax. Here, we will provide detailed examples on each type of `circuit_model`. These examples may be considered as template for users to craft their own `circuit_model`.

7.8.1 Inverters and Buffers

Template

```
<circuit_model type="inv_buf" name="<string>" prefix="<string>" netlist="<string>" is_
↪default="<int>">
  <design_technology type="cmos" topology="<string>" size="<int>" num_level="<int>" f_
↪per_stage="<float>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
</circuit_model>
```

```
<design_technology type="cmos" topology="<string>" size="<int>" num_level="<int>" f_per_stage="<float>"
>
```

- `topology="inverter|buffer"` Specify the type of this component, can be either an inverter or a buffer.
- `size="<int>"` Specify the driving strength of inverter/buffer. For a buffer, the size is the driving strength of the inverter at the second level. Note that we consider a two-level structure for a buffer here.
- `num_level="<int>"` Define the number of levels of a tapered inverter/buffer. This is required when users need an inverter or a buffer consisting of >2 stages
- `f_per_stage="<float>"` Define the ratio of driving strength between the levels of a tapered inverter/buffer. Default value is 4.

Inverter 1x Example

Fig. 7.19 is the inverter symbol depicted in this example.

Fig. 7.19: Classical inverter 1x symbol.

The XML code describing this inverter is:

```
<circuit_model type="inv_buf" name="inv1x" prefix="inv1x">
  <design_technology type="cmos" topology="inverter" size="1"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit_model>
```

This example shows:

- The topology chosen as inverter
- Size of 1 for the output strength
- The tapered parameter is not declared and is `false` by default

Power-gated Inverter 1x example

The XML code describing an inverter which can be power-gated by the control signals `EN` and `ENB` :

```
<circuit_model type="inv_buf" name="INVTX1" prefix="INVTX1">
  <design_technology type="cmos" topology="inverter" size="3" power_gated="true"/>
  <port type="input" prefix="in" size="1" lib_name="I"/>
  <port type="input" prefix="EN" size="1" lib_name="EN" is_global="true" default_val="0"␣
↪is_config_enable="true"/>
  <port type="input" prefix="ENB" size="1" lib_name="ENB" is_global="true" default_val="1
↪" is_config_enable="true"/>
  <port type="output" prefix="out" size="1" lib_name="Z"/>
</circuit_model>
```

Note: For power-gated inverters: all the control signals must be set as `config_enable` so that the testbench generation will generate testing waveforms. If the power-gated inverters are auto-generated, all the `config_enable` signals must be `global` signals as well. If the power-gated inverters come from user-defined netlists, restrictions on `global` signals are free.

Buffer 2x example

Fig. 7.20 is the buffer symbol depicted in this example.

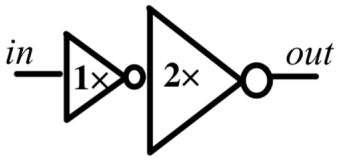


Fig. 7.20: Buffer made by two inverter, with an output strength of 2.

The XML code describing this buffer is:

```
<circuit_model type="inv_buf" name="buf2" prefix="buf2">
  <design_technology type="cmos" topology="buffer" size="2"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit_model>
```

This example shows:

- The topology chosen as buffer
- Size of 2 for the output strength

- The tapered parameter is not declared and is `false` by default

Power-gated Buffer 4x example

The XML code describing a buffer which can be power-gated by the control signals `EN` and `ENB` :

```
<circuit_model type="inv_buf" name="buf_4x" prefix="buf_4x">
  <design_technology type="cmos" topology="buffer" size="4" power_gated="true"/>
  <port type="input" prefix="in" size="1" lib_name="I"/>
  <port type="input" prefix="EN" size="1" lib_name="EN" is_global="true" default_val="0"␣
↪is_config_enable="true"/>
  <port type="input" prefix="ENB" size="1" lib_name="ENB" is_global="true" default_val="1
↪" is_config_enable="true"/>
  <port type="output" prefix="out" size="1" lib_name="Z"/>
</circuit_model>
```

Note: For power-gated buffers: all the control signals must be set as `config_enable` so that the testbench generation will generate testing waveforms. If the power-gated buffers are auto-generated, all the `config_enable` signals must be `global` signals as well. If the power-gated buffers come from user-defined netlists, restrictions on `global` signals are free.

Tapered inverter 16x example

Fig. 7.21 is the tapered inverter symbol depicted this example.

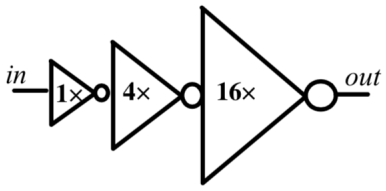


Fig. 7.21: Inverter with high output strength made by 3 stage of inverter.

The XML code describing this inverter is:

```
<circuit_model type="inv_buf" name="tapdrive4" prefix="tapdrive4">
  <design_technology type="cmos" topology="inverter" size="1" num_level="3" f_per_stage=
↪"4"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit_model>
```

This example shows:

- The topology chosen as inverter
- Size of 1 for the first stage output strength
- The number of stage is set to 3 by
- f_per_stage is set to 4. As a result, 2nd stage output strength is 4x, and the 3rd stage output strength is 16x.

Tapered buffer 64x example

The XML code describing a 4-stage buffer is:

```
<circuit_model type="inv_buf" name="tapbuf_16x" prefix="tapbuf_16x">
  <design_technology type="cmos" topology="buffer" size="1" num_level="4" f_per_stage="4
↪"/>
  <port type="input" prefix="in" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit_model>
```

This example shows:

- The topology chosen as buffer
- Size of 1 for the first stage output strength
- The number of stage is set to 4 by
- f_per_stage is set to 2. As a result, 2nd stage output strength is 4*, the 3rd stage output strength is 16*, and the 4th stage output strength is 64x.

7.8.2 Pass-gate Logic

Template

```
<circuit_model type="pass_gate" name="<string>" prefix="<string>" netlist="<string>" is_
↪default="<int>">
  <design_technology type="cmos" topology="<string>" nmos_size="<float>" pmos_size="
↪<float>"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
</circuit_model>
```

Note: The port sequence really matters! And all the input ports must have an input size of 1!

- The first input must be the datapath input, e.g., `in`.
- The second input must be the select input, e.g., `sel`.
- The third input (if applicable) must be the inverted select input, e.g., `selb`.

Warning: Please do **NOT** add input and output buffers to pass-gate logic.

`<design_technology type="cmos" topology="<string>" nmos_size="<float>" pmos_size="<float>"/>`

- `topology="transmission_gate|pass_transistor"` Specify the circuit topology for the pass-gate logic. A transmission gate consists of a *n*-type transistor and a *p*-type transistor. The pass transistor consists of only a *n*-type transistor.

- `nmos_size="<float>"` the size of *n*-type transistor in a transmission gate or pass_transistor, expressed in terms of the minimum width `min_width` defined in the transistor model in *Technology library*.
- `pmos_size="<float>"` the size of *p*-type transistor in a transmission gate, expressed in terms of the minimum width `min_width` defined in the transistor model in *Technology library*.

Note: `nmos_size` and `pmos_size` are required for FPGA-SPICE

Transmission-gate Example

Fig. 7.22 is the pass-gate symbol depicted in this example.

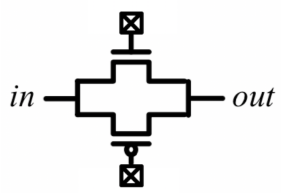


Fig. 7.22: Pass-gate made by a *p*-type and a *n*-type transistors.

The XML code describing this pass-gate is:

```
<circuit_model type="pass_gate" name="tgate" prefix="tgate">
  <design_technology type="cmos" topology="transmission_gate" nmos_size="1" pmos_size="2
↪"/>
  <port type="input" prefix="in" size="1"/>
  <port type="input" prefix="sram" size="1"/>
  <port type="input" prefix="sramb" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit_model>
```

This example shows:

- A `transmission_gate` built with a *n*-type transistor in the size of 1 and a *p*-type transistor in the size of 2.
- 3 inputs considered, 1 for datapath signal and 2 to turn on/off the transistors gates

Pass-transistor Example

Fig. 7.23 is the pass-gate symbol depicted in this example.

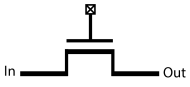


Fig. 7.23: Pass-gate made by a nmos transistor.

The XML code describing this pass-gate is:

```
<circuit_model type="pass_gate" name="t_pass" prefix="t_pass">
  <design_technology type="cmos" topology="pass_transistor"/>
  <port type="input" prefix="in" size="1"/>
  <port type="input" prefix="sram" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit_model>
```

This example shows:

- A `pass_transistor` build with a *n*-type transistor in the size of 1
- 2 inputs considered, 1 for datapath signal and 1 to turn on/off the transistor gate

7.8.3 SRAMs

Note: OpenFPGA does not auto-generate any netlist for SRAM cells. Users should define the HDL modeling in external netlists and ensure consistency to physical designs.

Template

```
<circuit_model type="sram" name="<string>" prefix="<string>" verilog_netlist="<string>"␣
↪spice_netlist="<string>"/>
  <design_technology type="cmos"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
</circuit_model>
```

Note: The circuit designs of SRAMs are highly dependent on the technology node and well optimized by engineers. Therefore, FPGA-Verilog/SPICE requires users to provide their customized SRAM Verilog/SPICE/Verilog netlists. A sample Verilog/SPICE netlist of SRAM can be found in the directory SpiceNetlists in the released package. FPGA-Verilog/SPICE assumes that all the LUTs and MUXes employ the SRAM circuit design. Therefore, currently only one SRAM type is allowed to be defined.

Note: The information of input and output buffer should be clearly specified according to the customized Verilog/SPICE netlist! The existence of input/output buffers will influence the decision in creating testbenches, which may leads to larger errors in power analysis.

SRAM with BL/WL

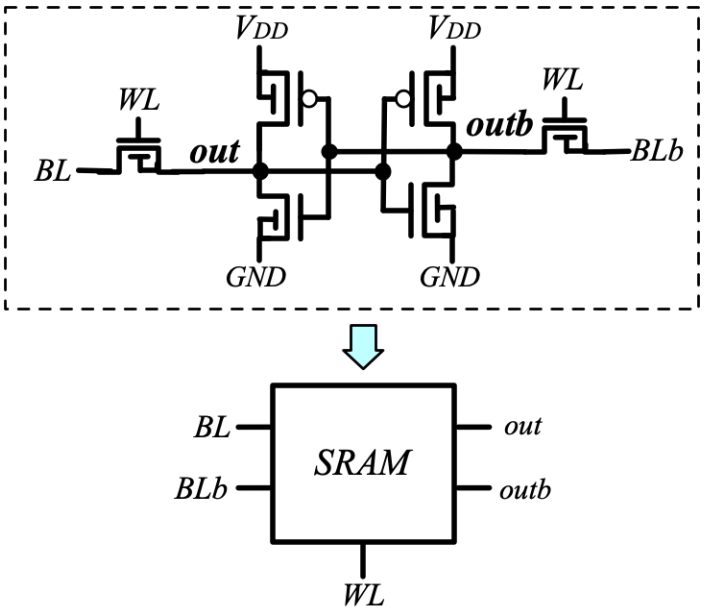


Fig. 7.24: An example of a SRAM with Bit-Line (BL) and Word-Line (WL) control signals

The following XML codes describes the SRAM cell shown in Fig. 7.24.

```
<circuit_model type="sram" name="sram_blwl" prefix="sram_blwl" verilog_netlist="sram.v"␣
↪spice_netlist="sram.sp"/>
  <design_technology type="cmos"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="bl" prefix="bl" size="1"/>
  <port type="blb" prefix="blb" size="1"/>
  <port type="wl" prefix="wl" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <port type="output" prefix="outb" size="1"/>
</circuit_model>
```

Note: OpenFPGA always assume that a `WL` port should be the write/read enable signal, while a `BL` port is the data input.

Note: When the `memory_bank` type of configuration procotol is specified, SRAM modules should have a BL and a WL.

SRAM with BL/WL/WLR

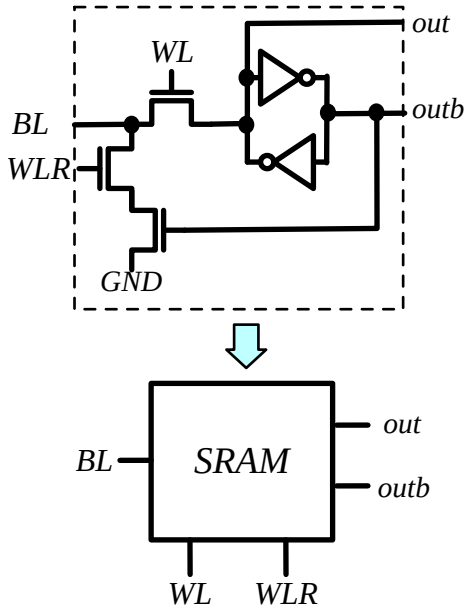


Fig. 7.25: An example of a SRAM with Bit-Line (BL), Word-Line (WL) and WL read control signals

The following XML codes describes the SRAM cell shown in Fig. 7.25.

```
<circuit_model type="sram" name="sram_blwlr" prefix="sram_blwlr" verilog_netlist="sram.v
↪" spice_netlist="sram.sp"/>
  <design_technology type="cmos"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="bl" prefix="bl" size="1"/>
  <port type="wl" prefix="wl" size="1"/>
  <port type="wlr" prefix="wlr" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <port type="output" prefix="outb" size="1"/>
</circuit_model>
```

Note: OpenFPGA always assume that a `WL` port should be the write enable signal, a `WLR` port should be the read enable signal, while a `BL` port is the data input.

Note: When the `memory_bank` type of configuration procotol is specified, SRAM modules should have a BL and a WL. WLR is optional

Configurable Latch

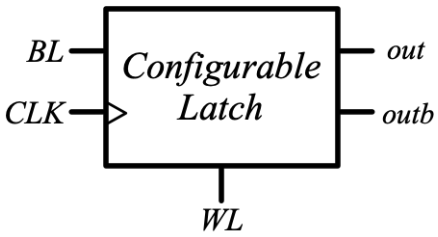


Fig. 7.26: An example of a SRAM-based configurable latch with Bit-Line (BL) and Word-Line (WL) control signals

The following XML codes describes the configurable latch shown in Fig. 7.26.

```
<circuit_model type="sram" name="config_latch" prefix="config_latch" verilog_netlist=
↪"sram.v" spice_netlist="sram.sp"/>
  <design_technology type="cmos"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="clock" prefix="clk" size="1"/>
  <port type="bl" prefix="bl" size="1"/>
  <port type="wl" prefix="wl" size="1"/>
  <port type="output" prefix="out" size="1"/>
  <port type="output" prefix="outb" size="1"/>
</circuit_model>
```

Note: OpenFPGA always assume that a `WL` port should be the write/read enable signal, while a `BL` port is the data input.

Note: When the `frame_based` type of configuration procotol is specified, the configurable latch or a SRAM with `BL` and `WL` should be specified.

7.8.4 Logic gates

The circuit model in the type of `gate` aims to support direct mapping to standard cells or customized cells provided by technology vendors or users.

Template

```
<circuit_model type="gate" name="<string>" prefix="<string>" spice_netlist="<string>"␣
↪verilog_netlist="<string>"/>
  <design_technology type="cmos" topology="<string>"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" lib_name="<string>" size="<int>"/>
  <port type="output" prefix="<string>" lib_name="<string>" size="<int>"/>
</circuit_model>
```

`<design_technology type="cmos" topology="<string>"/>`

- `topology="AND|OR|MUX2"` Specify the logic functionality of a gate. As for standard cells, the size of each port is limited to 1. Currently, only 2-input and single-output logic gates are supported.

Note: The port sequence really matters for MUX2 logic gates!

- The first two inputs must be the datapath inputs, e.g., `in0` and `in1`.
- The third input must be the select input, e.g., `sel`.

2-input AND Gate

```
<circuit_model type="gate" name="AND2" prefix="AND2" is_default="true">
  <design_technology type="cmos" topology="AND"/>
  <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="out" size="1"/>
  <delay_matrix type="rise" in_port="a b" out_port="out">
    10e-12 8e-12
  </delay_matrix>
  <delay_matrix type="fall" in_port="a b" out_port="out">
    10e-12 7e-12
  </delay_matrix>
</circuit_model>
```

This example shows:

- A 2-input AND gate without any input and output buffers
- Propagation delay from input `a` to `out` is 10ps in rising edge and and 8ps in falling edge
- Propagation delay from input `b` to `out` is 10ps in rising edge and 7ps in falling edge

2-input OR Gate

```
<circuit_model type="gate" name="OR2" prefix="OR2" is_default="true">
  <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="out" size="1"/>
  <delay_matrix type="rise" in_port="a b" out_port="out">
    10e-12 8e-12
  </delay_matrix>
  <delay_matrix type="fall" in_port="a b" out_port="out">
    10e-12 7e-12
  </delay_matrix>
</circuit_model>
```

This example shows:

- A 2-input OR gate without any input and output buffers
- Propagation delay from input `a` to `out` is 10ps in rising edge and and 8ps in falling edge
- Propagation delay from input `b` to `out` is 10ps in rising edge and 7ps in falling edge

MUX2 Gate

```
<circuit_model type="gate" name="MUX2" prefix="MUX2" is_default="true" verilog_netlist=
↪"sc_mux.v">
  <design_technology type="cmos" topology="MUX2"/>
  <input_buffer exist="false"/>
  <output_buffer exist="false"/>
  <port type="input" prefix="in0" lib_name="B" size="1"/>
  <port type="input" prefix="in1" lib_name="A" size="1"/>
  <port type="input" prefix="sel" lib_name="S" size="1"/>
  <port type="output" prefix="out" lib_name="Y" size="1"/>
</circuit_model>
```

This example shows:

- A 2-input MUX gate with two inputs `in0` and `in1`, a select port `sel` and an output port `out`
- The Verilog of MUX2 gate is provided by the user in the netlist `sc_mux.v`
- The use of `lib_name` to bind to a Verilog module with different port names.
- When binding to the Verilog module, the inputs will be swapped. In other words, `in0` of the circuit model will be wired to the input `B` of the MUX2 cell, while `in1` of the circuit model will be wired to the input `A` of the MUX2 cell.

Note: OpenFPGA requires a fixed truth table for the `MUX2` gate. When the select signal sel is enabled, the first input, i.e., `in0`, will be propagated to the output, i.e., `out`. If your standard cell provider does not offer the exact truth table, you can simply swap the inputs as shown in the example.

7.8.5 Multiplexers

Template

```
<circuit_model type="mux" name="<string>" prefix="<string>">
  <design_technology type="<string>" structure="<string>" num_level="<int>" add_const_
↪input="<bool>" const_input_val="<int>" local_encoder="<bool>"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <pass_gate_logic type="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
  <port type="sram" prefix="<string>" size="<int>"/>
</circuit_model>
```

Note: user-defined Verilog/SPICE netlists are not currently supported for multiplexers.

<design_technology type="<string>" structure="<string>" num_level="<int>" add_const_input="<bool>" const
>

- structure="tree|multi-level|one-level" Specify the multiplexer structure for a multiplexer. The structure option is only valid for SRAM-based multiplexers. For RRAM-based multiplexers, currently we only support the one-level structure
- num_level="<int>" Specify the number of levels when multi-level structure is selected.
- add_const_input="true|false" Specify if an extra input should be added to the multiplexer circuits. For example, an 4-input multiplexer will be turned to a 5-input multiplexer. The extra input will be wired to a constant value, which can be specified through the XML syntax const_input_val.

  Note: Adding an extra constant input will help reducing the leakage power of FPGA and parasitic signal activities, with a limited area overhead.

- const_input_val="0|1" Specify the constant value, to which the extra input will be connected. By default it is 0. This syntax is only valid when the add_const_input is set to true.
- local_encoder="true|false". Specify if a local encoder should be added to the multiplexer circuits. The local encoder will interface the SRAM inputs of multiplexing structure and SRAMs. It can encode the one-hot codes (that drive the select port of multiplexing structure) to a binary code. For example, 8-bit 00000001 will be encoded to 3-bit 000. This will help reduce the number of SRAM cells used in FPGAs as well as configuration time (especially for scan-chain configuration protocols). But it may cost an area overhead.

  Note: Local encoders are only applicable for one-level and multi-level multiplexers. Tree-like multiplexers are already encoded in their nature.

Note: A multiplexer should have only three types of ports, input, output and sram, which are all mandatory.

Note: For tree-like multiplexers, they can be built with standard cell MUX2. To enable this, users should define a circuit_model, which describes a 2-input multiplexer (See details and examples in how to define a logic gate using circuit_model. In this case, the circuit_model_name in the pass_gate_logic should be the name of MUX2 circuit_model.

Note: When multiplexers are not provided by users, the size of ports do not have to be consistent with actual numbers in the architecture.

One-level Multiplexer

Fig. 7.27 illustrates an example of multiplexer modelling, which consists of input/output buffers and a transmission-gate-based tree structure.

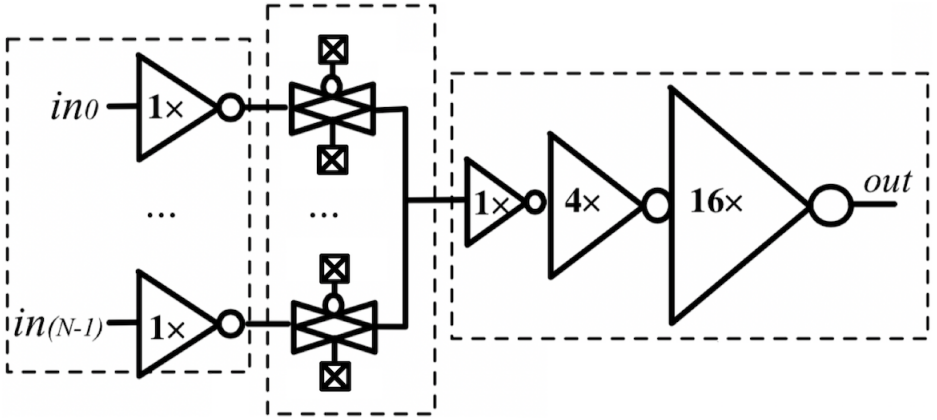


Fig. 7.27: An example of a one level multiplexer with transistor-level design parameters

The code describing this Multiplexer is:

```
<circuit_model type="mux" name="mux_1level" prefix="mux_1level">
  <design_technology type="cmos" structure="one-level"/>
  <input_buffer exist="on" circuit_model_name="inv1x"/>
  <output_buffer exist="on" circuit_model_name="tapbuf4"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="in" size="4"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="4"/>
</circuit_model>
```

This example shows:

- A one-level 4-input CMOS multiplexer
- All the inputs will be buffered using the circuit model `inv1x`
- All the outputs will be buffered using the circuit model `tapbuf4`
- The multiplexer will be built by transmission gate using the circuit model `tgate`
- The multiplexer will have 4 inputs and 4 SRAMs to control which datapath to propagate

Tree-like Multiplexer

Fig. 7.28 illustrates an example of multiplexer modelling, which consists of input/output buffers and a transmission-gate-based tree structure.

If we arbitrarily fix the number of Mux entries at 4, the following code could illustrate (a):

```
<circuit_model type="mux" name="mux_tree" prefix="mux_tree">
  <design_technology type="cmos" structure="tree"/>
  <input_buffer exist="on" circuit_model_name="inv1x"/>
```

(continues on next page)

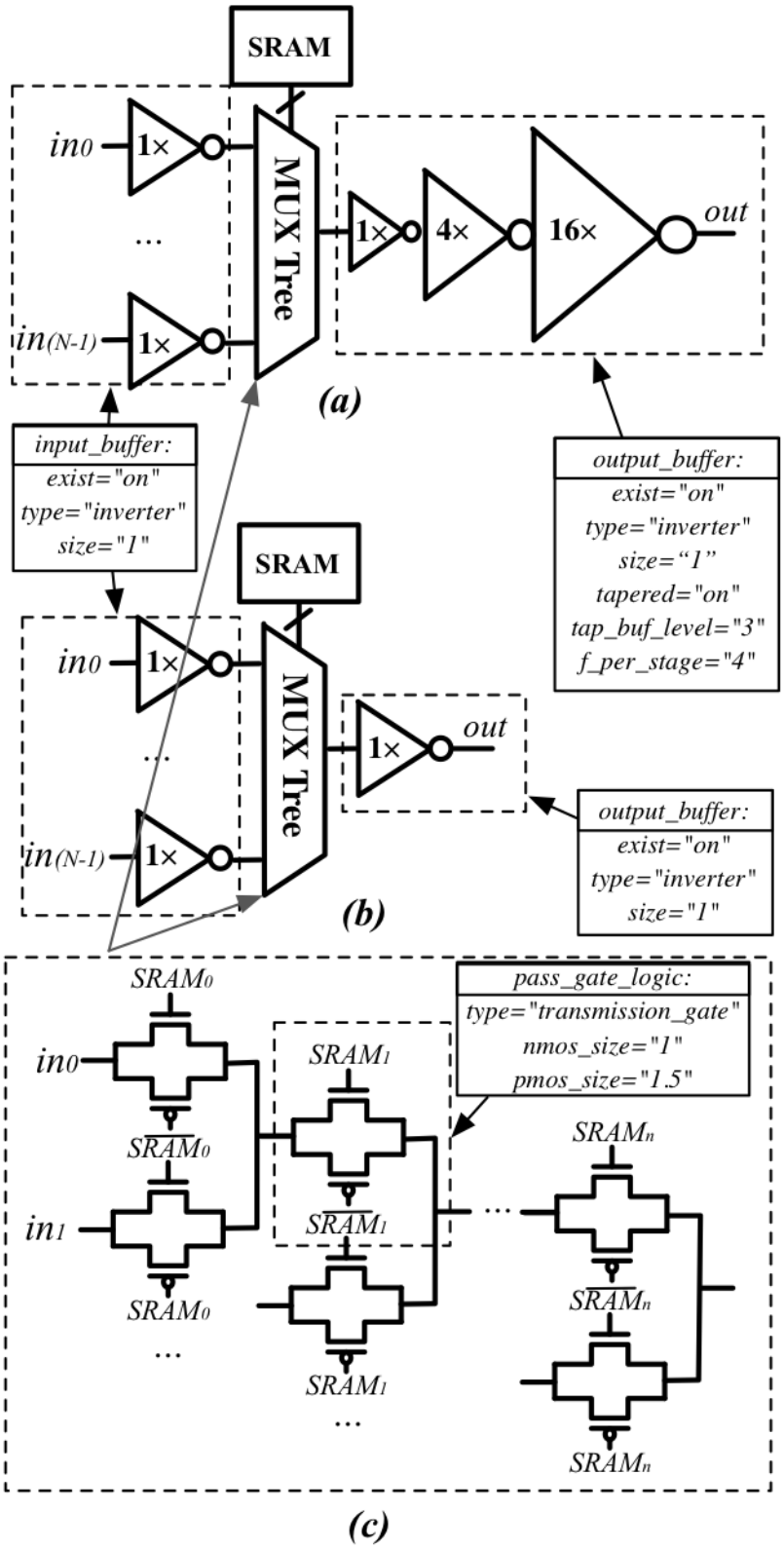


Fig. 7.28: An example of a tree-like multiplexer with transistor-level design parameters

(continued from previous page)

```
  <output_buffer exist="on" circuit_model_name="tapdrive4"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="in" size="4"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="3"/>
</circuit_model>
```

This example shows:

- A tree-like 4-input CMOS multiplexer
- All the inputs will be buffered using the circuit model `inv1x`
- All the outputs will be buffered using the circuit model `tapbuf4`
- The multiplexer will be built by transmission gate using the circuit model `tgate`
- The multiplexer will have 4 inputs and 3 SRAMs to control which datapath to propagate

Standard Cell Multiplexer

```
<circuit_model type="mux" name="mux_stdcell" prefix="mux_stdcell">
  <design_technology type="cmos" structure="tree"/>
  <input_buffer exist="on" circuit_model_name="inv1x"/>
  <output_buffer exist="on" circuit_model_name="tapdrive4"/>
  <pass_gate_logic circuit_model_name="MUX2"/>
  <port type="input" prefix="in" size="4"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="3"/>
</circuit_model>
```

This example shows:

- A tree-like 4-input CMOS multiplexer built by the standard cell `MUX2`
- All the inputs will be buffered using the circuit model `inv1x`
- All the outputs will be buffered using the circuit model `tapbuf4`
- The multiplexer will have 4 inputs and 3 SRAMs to control which datapath to propagate

Multi-level Multiplexer

```
<circuit_model type="mux" name="mux_2level" prefix="mux_stdcell">
  <design_technology type="cmos" structure="multi_level" num_level="2"/>
  <input_buffer exist="on" circuit_model_name="inv1x"/>
  <output_buffer exist="on" circuit_model_name="tapdrive4"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="16"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="8"/>
</circuit_model>
```

This example shows:

- A two-level 16-input CMOS multiplexer built by the transmission gate `TGATE`

- All the inputs will be buffered using the circuit model `inv1x`
- All the outputs will be buffered using the circuit model `tapbuf4`
- The multiplexer will have 16 inputs and 8 SRAMs to control which datapath to propagate

Multiplexer with Local Encoder

```
<circuit_model type="mux" name="mux_2level" prefix="mux_stdcell">
  <design_technology type="cmos" structure="multi_level" num_level="2" local_encoder=
↪"true"/>
  <input_buffer exist="on" circuit_model_name="inv1x"/>
  <output_buffer exist="on" circuit_model_name="tapdrive4"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="16"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="4"/>
</circuit_model>
```

This example shows:

- A two-level 16-input CMOS multiplexer built by the transmission gate `TGATE`
- All the inputs will be buffered using the circuit model `inv1x`
- All the outputs will be buffered using the circuit model `tapbuf4`
- The multiplexer will have 16 inputs and 4 SRAMs to control which datapath to propagate
- Two local encoders are generated between the SRAMs and multiplexing structure to reduce the number of configurable memories required.

Multiplexer with Constant Input

```
<circuit_model type="mux" name="mux_2level" prefix="mux_stdcell">
  <design_technology type="cmos" structure="multi_level" num_level="2" add_const_input=
↪"true" const_input_val="1"/>
  <input_buffer exist="on" circuit_model_name="inv1x"/>
  <output_buffer exist="on" circuit_model_name="tapdrive4"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="input" prefix="in" size="14"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="8"/>
</circuit_model>
```

This example shows:

- A two-level 16-input CMOS multiplexer built by the transmission gate `TGATE`
- All the inputs will be buffered using the circuit model `inv1x`
- All the outputs will be buffered using the circuit model `tapbuf4`
- The multiplexer will have 15 inputs and 8 SRAMs to control which datapath to propagate
- An constant input toggled at logic '1' is added in addition to the 14 regular inputs

7.8.6 Look-Up Tables

Template

```
<circuit_model type="lut" name="<string>" prefix="<string>" spice_netlist="<string>"␣
↪verilog_netlist="<string>"/>
  <design_technology type="cmos" fracturable_lut="<bool>"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <lut_input_buffer exist="<string>" circuit_model_name="<string>"/>
  <lut_input_inverter exist="<string>" circuit_model_name="<string>"/>
  <lut_intermediate_buffer exist="<string>" circuit_model_name="<string>" location_map="
↪<string>"/>
  <pass_gate_logic type="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>" tri_state_map="<string>" circuit_
↪model_name="<string>" is_harden_lut_port="<bool>"/>
  <port type="output" prefix="<string>" size="<int>" lut_frac_level="<int>" lut_output_
↪mask="<int>" is_harden_lut_port="<bool>"/>
  <port type="sram" prefix="<string>" size="<int>" mode_select="<bool>" circuit_model_
↪name="<string>" default_val="<int>"/>
</circuit_model>
```

Note: The Verilog/SPICE netlists of LUT can be auto-generated or customized. The auto-generated LUTs are based on a tree-like multiplexer, whose gates of the transistors are used as the inputs of LUTs and the drains/sources of the transistors are used for configurable memories (SRAMs). The LUT provided in customized Verilog/SPICE netlist should have the same decoding methodology as the traditional LUT.

<lut_input_buffer exist="<string>" circuit_model_name="<string>"/>

Define transistor-level description for the buffer for the inputs of a LUT (gates of the internal multiplexer).

- exist="true|false" Specify if the input buffer should exist for LUT inputs
- circuit_model_name="<string>" Specify the circuit_model that will be used to build the input buffers

Note: In the context of LUT, input_buffer corresponds to the buffer for the datapath inputs of multiplexers inside a LUT. lut_input_buffer corresponds to the buffer at the inputs of a LUT

<lut_input_inverter exist="<string>" circuit_model_name="<string>"/>

Define transistor-level description for the inverter for the inputs of a LUT (gates of the internal multiplexer).

- exist="true|false" Specify if the input buffer should exist for LUT inputs
- circuit_model_name="<string>" Specify the circuit_model that will be used to build the input inverters

<lut_intermediate_buffer exist="<string>" circuit_model_name="<string>" location_map="<string>"/
>

Define transistor-level description for the buffer locating at intermediate stages of internal multiplexer of a LUT.

- exist="true|false" Specify if the input buffer should exist at intermediate stages
- circuit_model_name="<string>" Specify the circuit_model that will be used to build these buffers

- `location_map="[1|-]"` Customize the location of buffers in intermediate stages. Users can define an integer array consisting of '1' and '-'. Take the example in Fig. 7.29, `-1-` indicates buffer inseration to the second stage of the LUT multiplexer tree, considering a 3-input LUT.

Note: For a LUT, three types of ports (`input`, `output` and `sram`) should be defined. If the user provides an customized Verilog/SPICE netlist, the bandwidth of ports should be defined to the same as the Verilog/SPICE netlist. To support customizable LUTs, each type of port contain special keywords.

```
<port type="input" prefix="<string>" size="<int>" tri_state_map="<string>" circuit_model_name="<string>"
>
```

- `tri_state_map="[-|1]"` Customize which inputs are fixed to constant values when the LUT is in fracturable modes. For example, `tri_state_map="----11"` indicates that the last two inputs will be fixed to be logic '1' when a 6-input LUT is in fracturable modes.
- `circuit_model_name="<string>"` Specify the circuit model to build logic gates in order to tri-state the inputs in fracturable LUT modes. It is required to use an `AND` gate to force logic '0' or an `OR` gate to force logic '1' for the input ports.
- `is_harden_lut_port="[true|false]"` Specify if the input drives a harden logic inside a LUT. A harden input is supposed **NOT** to drive any multiplexer input (the internal multiplexer of LUT). As a result, such inputs are not considered to implement any truth table mapped to the LUT. If enabled, the input will **NOT** be considered for wiring to internal multiplexers as well as bitstream generation. By default, an input port is treated **NOT** to be a harden LUT port.

```
<port type="output" prefix="<string>" size="<int>" lut_frac_level="<int>" lut_output_mask="<int>" is_ha
>
```

- `lut_frac_level="<int>"` Specify the level in LUT multiplexer tree where the output port are wired to. For example, `lut_frac_level="4"` in a fracturable LUT6 means that the output are potentially wired to the 4th stage of a LUT multiplexer and it is an output of a LUT4.
- `lut_output_mask="<int>"` Describe which fracturable outputs are used. For instance, in a 6-LUT, there are potentially four LUT4 outputs can be wired out. `lut_output_mask="0,2"` indicates that only the first and the thrid LUT4 outputs will be used in fracturable mode.
- `is_harden_lut_port="[true|false]"` Specify if the output is driven by a harden logic inside a LUT. A harden input is supposed **NOT** to be driven by any multiplexer output (the internal multiplexer of LUT). As a result, such outputs are not considered to implement any truth table mapped to the LUT. If enabled, the output will **NOT** be considered for wiring to internal multiplexers as well as bitstream generation. By default, an output port is treated **NOT** to be a harden LUT port.

Note: The size of the output port should be consistent to the length of `lut_output_mask`.

```
<port type="sram" prefix="<string>" size="<int>" mode_select="<bool>" circuit_model_name="<string>" defa
>
```

- `mode_select="true|false"` Specify if this port is used to switch the LUT between different operating modes, the SRAM bits of a fracturable LUT consists of two parts: configuration memory and mode selecting.
- `circuit_model_name="<string>"` Specify the circuit model to be drive the SRAM port. Typically, the circuit model should be in the type of `ccff` or `sram`.
- `default_val="0|1"` Specify the default value for the SRAM port. The default value will be used in generating testbenches for unused LUTs

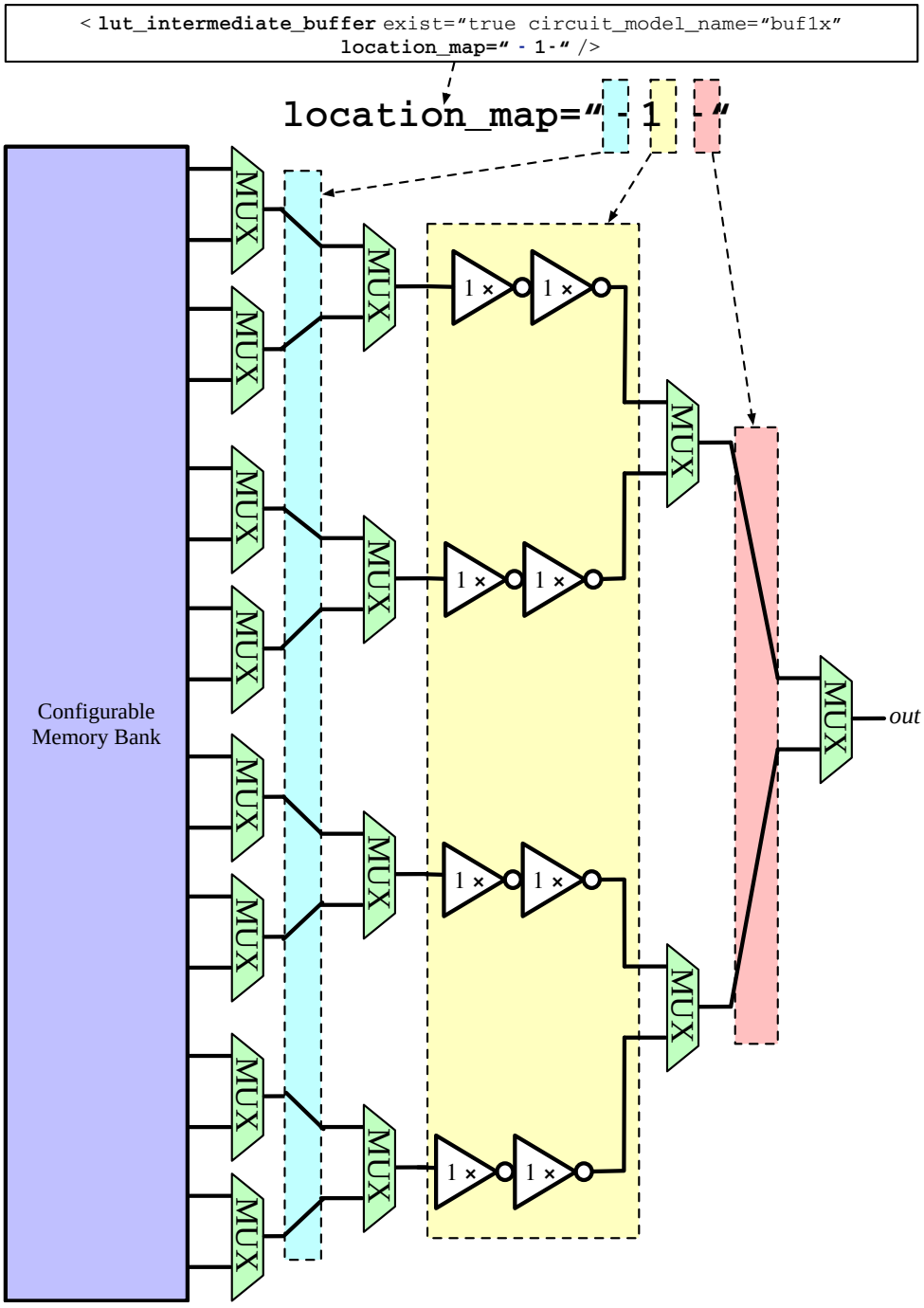


Fig. 7.29: An example of adding intermediate buffers to a 3-input Look-Up Table (LUT).

Note: The size of a mode-selection SRAM port should be consistent to the number of '1s' or '0s' in the `tri_state_map`.

Single-Output LUT

Fig. 7.30 illustrates an example of LUT modeling, which consists of input/output buffers and a transmission-gate-based tree structure.

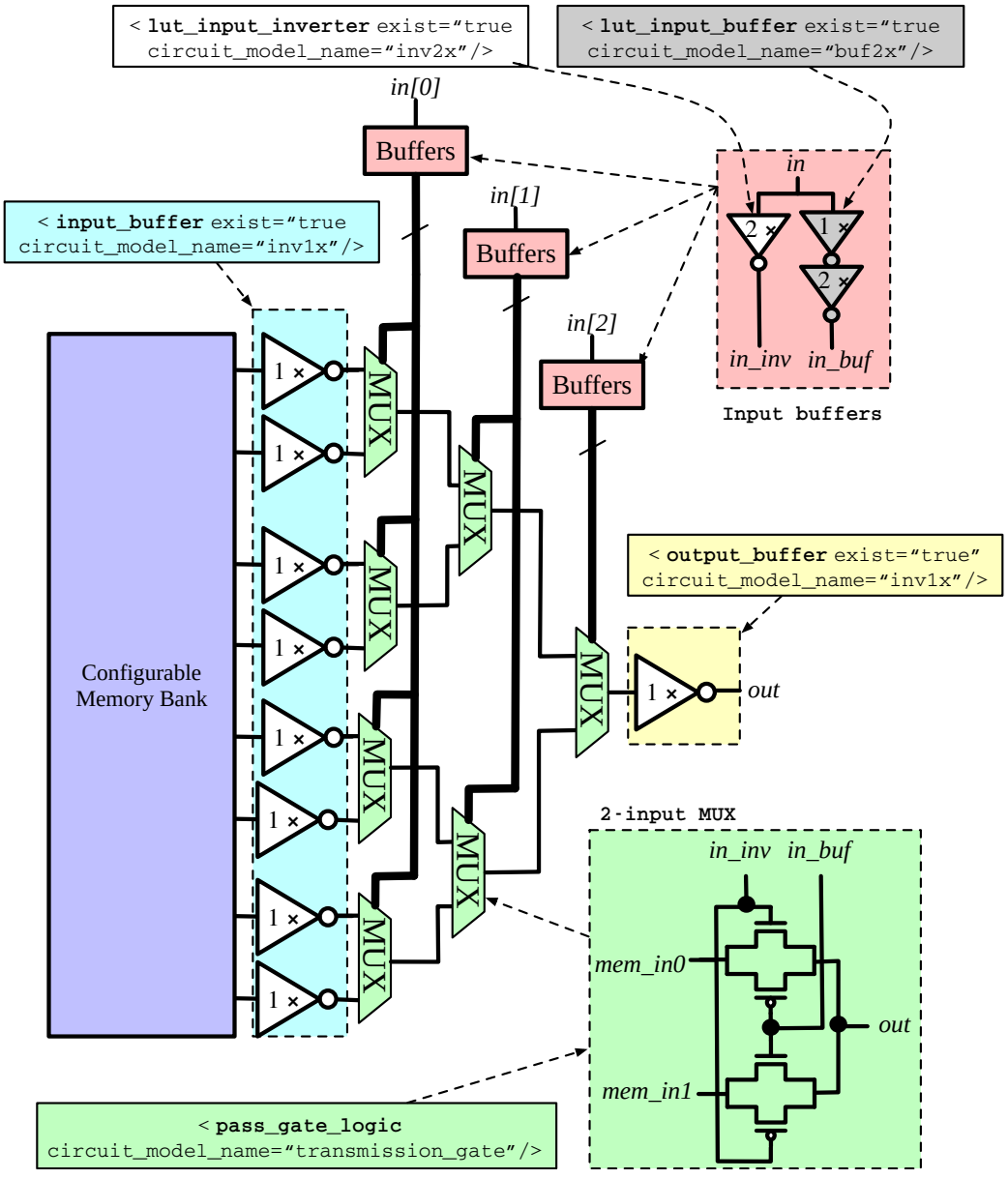


Fig. 7.30: An example of a single-output 3-input LUT.

The code describing this LUT is:

```
<circuit_model type="lut" name="lut3" prefix="lut3">
  <input_buffer exist="on" circuit_model="inv1x"/>
  <output_buffer exist="on" circuit_model_name="inv1x"/>
  <lut_input_buffer exist="on" circuit_model_name="buf2"/>
  <lut_input_inverter exist="on" circuit_model_name="inv1x"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="in" size="3"/>
  <port type="output" prefix="out" size="1"/>
  <port type="sram" prefix="sram" size="8"/>
</circuit_model>
```

This example shows:

- A 3-input LUT which is configurable by 8 SRAM cells.
- The multiplexer inside LUT will be built with transmission gate using circuit model `inv1x`
- There are no internal buffered inserted to any intermediate stage of a LUT

Standard Fracturable LUT

Fig. 7.31 illustrates a typical example of 3-input fracturable LUT modeling, which consists of input/output buffers and a transmission-gate-based tree structure.

The code describing this LUT is:

```
<circuit_model type="lut" name="frac_lut3" prefix="frac_lut3" dump_structural_verilog=
↪"true">
  <design_technology type="cmos" fracturable_lut="true"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <lut_input_inverter exist="true" circuit_model_name="inv1x"/>
  <lut_input_buffer exist="true" circuit_model_name="buf4"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="in" size="3" tri_state_map="--1" circuit_model_name="OR2"/>
  <port type="output" prefix="lut2_out" size="1" lut_frac_level="3" lut_output_mask="0"/>
  <port type="output" prefix="lut3_out" size="1" lut_output_mask="0"/>
  <port type="sram" prefix="sram" size="8"/>
  <port type="sram" prefix="mode" size="1" mode_select="true" circuit_model_name="ccff"␣
↪default_val="0"/>
</circuit_model>
```

This example shows:

- Fracturable 3-input LUT which is configurable by 9 SRAM cells.
- There is a SRAM cell to switch the operating mode of this LUT, configured by a configuration-chain flip-flop `ccff`
- The last input `in[2]` of LUT will be tri-stated in dual-LUT2 mode.
- An 2-input OR gate will be wired to the last input `in[2]` to tri-state the input. The mode-select SRAM will be wired to an input of the OR gate. It means that when the mode-selection bit is '0', the LUT will operate in dual-LUT3 mode.
- There will be two outputs wired to the 2th stage of routing multiplexer (the outputs of dual 2-input LUTs)

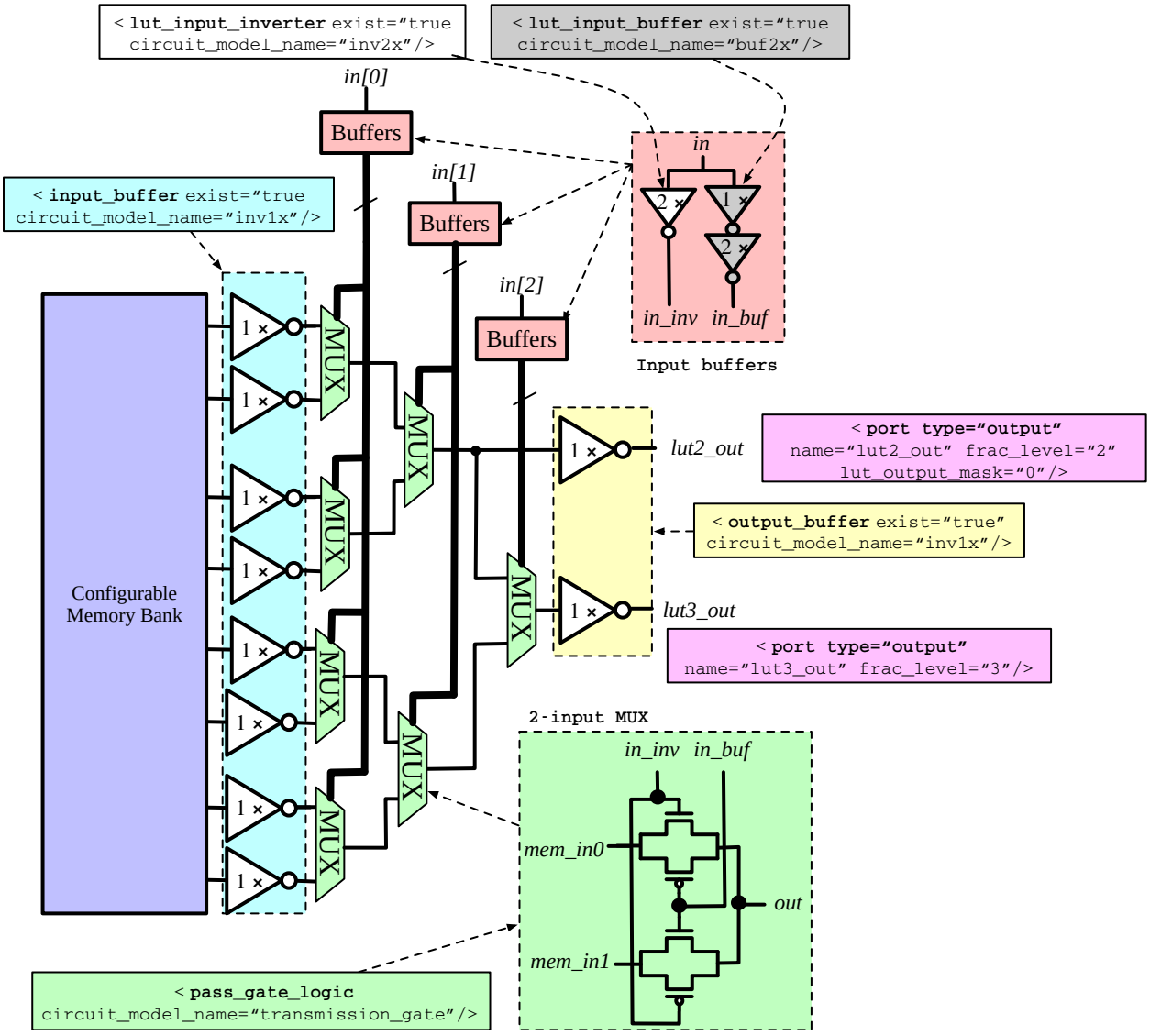


Fig. 7.31: An example of a fracturable 3-input LUT.

- By default, the mode-selection configuration bit will be '0', indicating that by default the LUT will operate in dual-LUT2 mode.

Fig. 7.32 illustrates the detailed schematic of a standard fracturable 6-input LUT, where the 5th and 6th inputs can be pull up/down to a fixed logic value to enable LUT4 and LUT5 outputs.

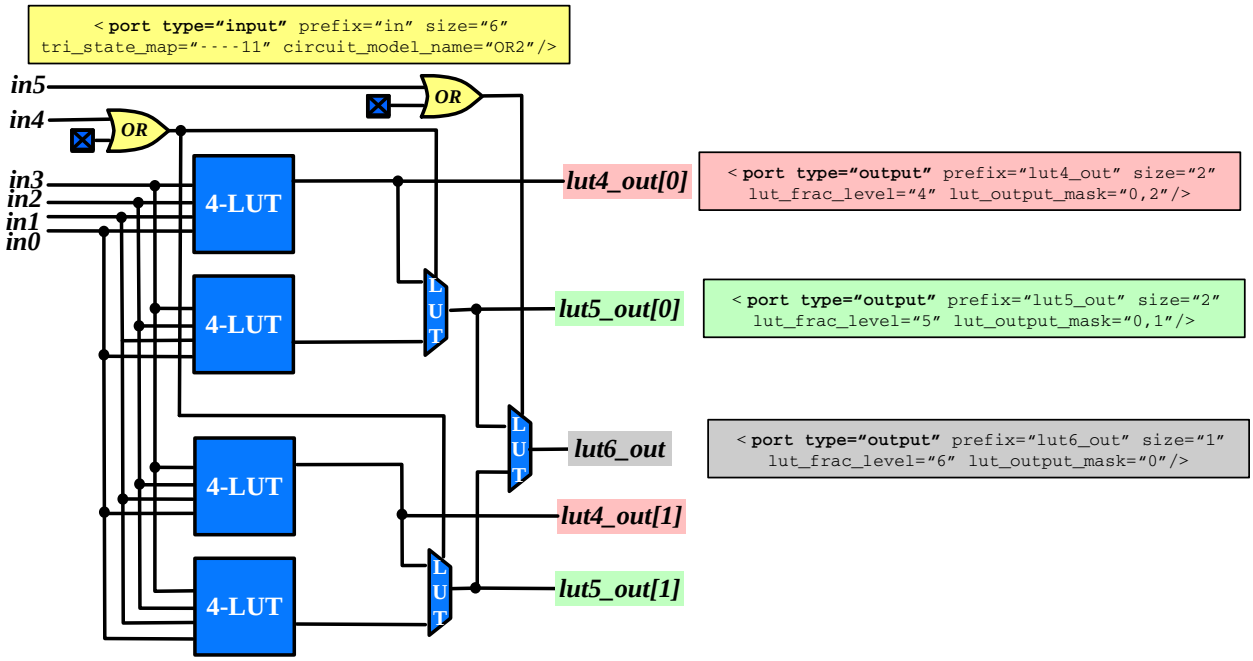


Fig. 7.32: Detailed schematic of a standard fracturable 6-input LUT.

The code describing this LUT is:

```
<circuit_model type="lut" name="frac_lut6" prefix="frac_lut6" dump_structural_verilog=
↪"true">
  <design_technology type="cmos" fracturable_lut="true"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <lut_input_inverter exist="true" circuit_model_name="inv1x"/>
  <lut_input_buffer exist="true" circuit_model_name="buf4"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="in" size="6" tri_state_map="----11" circuit_model_name="OR2
↪"/>
  <port type="output" prefix="lut4_out" size="2" lut_frac_level="4" lut_output_mask="0,2
↪"/>
  <port type="output" prefix="lut5_out" size="2" lut_frac_level="5" lut_output_mask="0,1
↪"/>
  <port type="output" prefix="lut6_out" size="1" lut_output_mask="0"/>
  <port type="sram" prefix="sram" size="64"/>
  <port type="sram" prefix="mode" size="2" mode_select="true" circuit_model_name="ccff"␣
↪default_val="1"/>
</circuit_model>
```

This example shows:

- Fracturable 6-input LUT which is configurable by 66 SRAM cells.

- There are two SRAM cells to switch the operating mode of this LUT, configured by two configuration-chain flip-flops `ccff`
- The inputs `in[4]` and `in[5]` of LUT will be tri-stated in dual-LUT4 and dual-LUT5 modes respectively.
- An 2-input OR gate will be wired to the inputs `in[4]` and `in[5]` to tri-state them. The mode-select SRAM will be wired to an input of the OR gate.
- There will be two outputs wired to the 4th stage of routing multiplexer (the outputs of dual 4-input LUTs)
- There will be two outputs wired to the 5th stage of routing multiplexer (the outputs of dual 5-input LUTs)
- By default, the mode-selection configuration bit will be '11', indicating that by default the LUT will operate in dual-LUT4 mode.

Native Fracturable LUT

Fig. 7.33 illustrates the detailed schematic of a native fracturable 6-input LUT, where LUT4, LUT5 and LUT6 outputs are always active and there are no tri-state buffers.

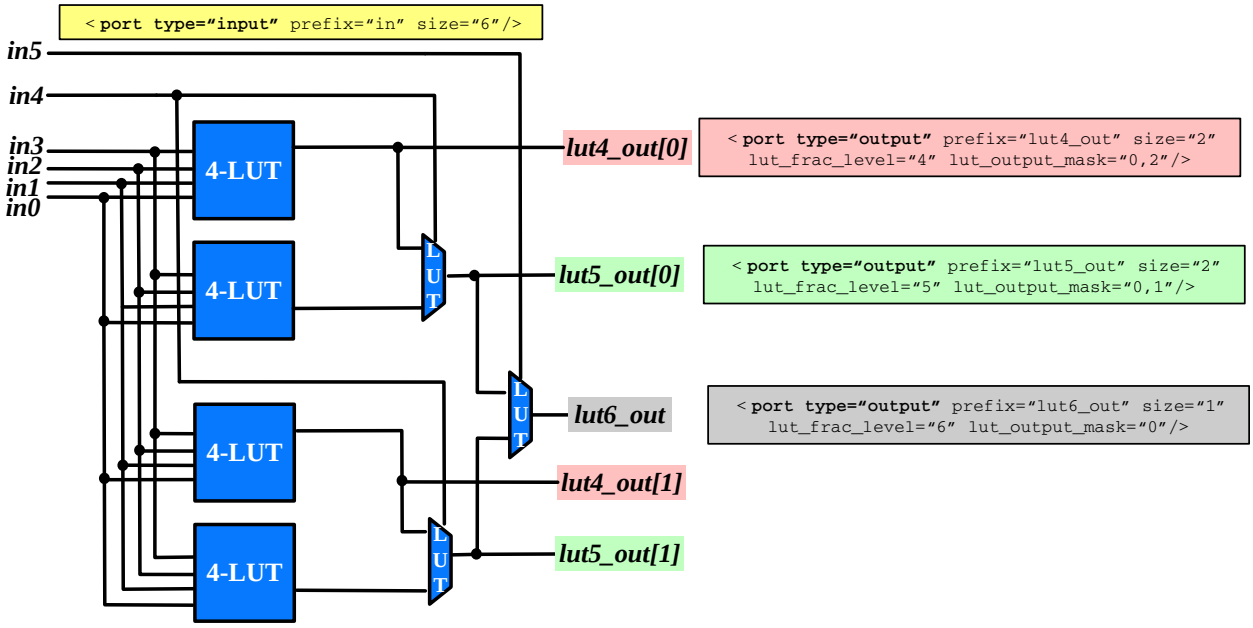


Fig. 7.33: Detailed schematic of a native fracturable 6-input LUT.

The code describing this LUT is:

```
<circuit_model type="lut" name="frac_lut6" prefix="frac_lut6" dump_structural_verilog=
↪"true">
  <design_technology type="cmos" fracturable_lut="true"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <lut_input_inverter exist="true" circuit_model_name="inv1x"/>
  <lut_input_buffer exist="true" circuit_model_name="buf4"/>
  <pass_gate_logic circuit_model_name="tgate"/>
  <port type="input" prefix="in" size="6"/>
  <port type="output" prefix="lut4_out" size="2" lut_frac_level="4" lut_output_mask="0,2
↪"/>
```

(continues on next page)

(continued from previous page)

```
  <port type="output" prefix="lut5_out" size="2" lut_frac_level="5" lut_output_mask="0,1
↪"/>
  <port type="output" prefix="lut6_out" size="1" lut_output_mask="0"/>
  <port type="sram" prefix="sram" size="64"/>
</circuit_model>
```

This example shows:

- Fracturable 6-input LUT which is configurable by 64 SRAM cells.
- There will be two outputs wired to the 4th stage of routing multiplexer (the outputs of dual 4-input LUTs)
- There will be two outputs wired to the 5th stage of routing multiplexer (the outputs of dual 5-input LUTs)

LUT with Harden Logic

Fig. 7.34 illustrates the detailed schematic of a fracturable 4-input LUT coupled with carry logic gates. For fracturable LUT schematic, please refer to Fig. 7.32. This feature allows users to fully customize their LUT circuit implementation while being compatible with OpenFPGA's bitstream generator when mapping truth tables to the LUTs.

Warning: OpenFPGA does **NOT** support netlist autogeneration for the LUT with harden logic. Users should build their own netlist and use `verilog_netlist` syntax of *Circuit Library* to include it.

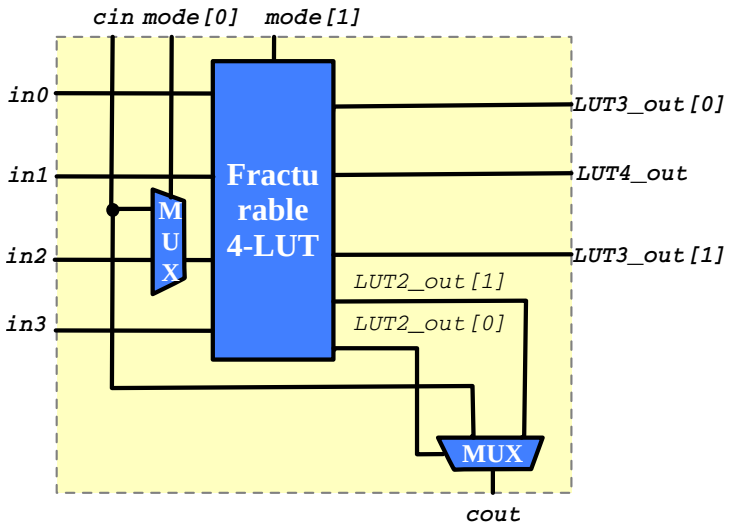


Fig. 7.34: Detailed schematic of a fracturable 4-input LUT with embedded carry logic.

The code describing this LUT is:

```
<circuit_model type="lut" name="frac_lut4_arith" prefix="frac_lut4_arith" dump_
↪structural_verilog="true" verilog_netlist="${OPENFPGA_PATH}/openfpga_flow/openfpga_
↪cell_library/verilog/frac_lut4_arith.v">
  <design_technology type="cmos" fracturable_lut="true"/>
  <input_buffer exist="false"/>
  <output_buffer exist="true" circuit_model_name="sky130_fd_sc_hd__buf_2"/>
```

(continues on next page)

(continued from previous page)

```
  <lut_input_inverter exist="true" circuit_model_name="sky130_fd_sc_hd__inv_1"/>
  <lut_input_buffer exist="true" circuit_model_name="sky130_fd_sc_hd__buf_2"/>
  <lut_intermediate_buffer exist="true" circuit_model_name="sky130_fd_sc_hd__buf_2"␣
↪location_map="-1-"/>
  <pass_gate_logic circuit_model_name="sky130_fd_sc_hd__mux2_1"/>
  <port type="input" prefix="in" size="4" tri_state_map="---1" circuit_model_name=
↪"sky130_fd_sc_hd__or2_1"/>
  <port type="input" prefix="cin" size="1" is_harden_lut_port="true"/>
  <port type="output" prefix="lut3_out" size="2" lut_frac_level="3" lut_output_mask="0,1
↪"/>
  <port type="output" prefix="lut4_out" size="1" lut_output_mask="0"/>
  <port type="output" prefix="cout" size="1" is_harden_lut_port="true"/>
  <port type="sram" prefix="sram" size="16"/>
  <port type="sram" prefix="mode" size="2" mode_select="true" circuit_model_name="DFFRQ"␣
↪default_val="1"/>
</circuit_model>
```

This example shows:

- Fracturable 4-input LUT which is configurable by 16 SRAM cells.
- There are two output wired to the 3th stage of routing multiplexer (the outputs of dual 3-input LUTs)
- There are two outputs wired to the 2th stage of routing multiplexer (the outputs of 2-input LUTs in the in the lower part of SRAM cells). Note that the two outputs drive the embedded carry logic
- There is a harden carry logic, i.e., a 2-input MUX, to implement high-performance carry function.
- There is a mode-switch multiplexer at `cin` port, which is used to switch between arithemetic mode and regular LUT mode.

Note: If the embedded harden logic are driven partially by LUT outputs, users may use the *Bitstream Setting (.xml)* to gaurantee correct bitstream generation for the LUTs.

7.8.7 Datapath Flip-Flops

Note: OpenFPGA does not auto-generate any netlist for datapath flip-flops. Users should define the HDL modeling in external netlists and ensure consistency to physical designs.

Template

```
<circuit_model type="ff" name="<string>" prefix="<string>" spice_netlist="<string>"␣
↪verilog_netlist="<string>"/>
  <design_technology type="cmos"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
```

(continues on next page)

(continued from previous page)

```
  <port type="clock" prefix="<string>" size="<int>"/>
</circuit_model>
```

Note: The circuit designs of flip-flops are highly dependent on the technology node and well optimized by engineers. Therefore, FPGA-Verilog/SPICE requires users to provide their customized FF Verilog/SPICE/Verilog netlists. A sample Verilog/SPICE netlist of FF can be found in the directory SpiceNetlists in the released package.

The information of input and output buffer should be clearly specified according to the customized SPICE netlist! The existence of input/output buffers will influence the decision in creating SPICE testbenches, which may leads to larger errors in power analysis.

Note: FPGA-Verilog/SPICE currently support only one clock domain in the FPGA. Therefore there should be only one clock port to be defined and the size of the clock port should be 1.

type="ff"

`ff` is a regular flip-flop to be used in datapath logic, e.g., a configurable logic block.

Note: A flip-flop should at least have three types of ports, `input`, `output` and `clock`.

Note: If the user provides a customized Verilog/SPICE netlist, the bandwidth of ports should be defined to the same as the Verilog/SPICE netlist.

D-type Flip-Flop

Fig. 7.35 illustrates an example of regular flip-flop.

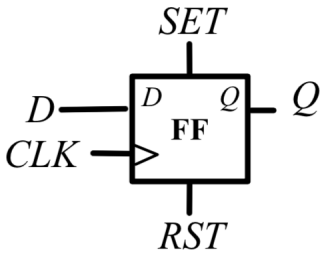


Fig. 7.35: An example of classical Flip-Flop.

The code describing this FF is:

```
<circuit_model type="ff" name="dff" prefix="dff" verilog_netlist="ff.v" spice_netlist=
↪"ff.sp">
  <port type="input" prefix="D" lib_name="D" size="1"/>
  <port type="input" prefix="Set" lib_name="S" size="1" is_global="true"/>
  <port type="input" prefix="Reset" lib_name="R" size="1" is_global="true"/>
  <port type="output" prefix="Q" lib_name="Q" size="1"/>
```

(continues on next page)

(continued from previous page)

```
  <port type="clock" prefix="clk" lib_name="CK" size="1" is_global="true"/>
</circuit_model>
```

This example shows:

- A regular flip-flop which is defined in a Verilog netlist `ff.v` and a SPICE netlist `ff.sp`
- The flip-flop has `set` and `reset` functionalities
- The flip-flop port names defined differently in standard cell library and VPR architecture. The `lib_name` capture the port name defined in standard cells, while `prefix` capture the port name defined in `pb_type` of VPR architecture file

Multi-mode Flip-Flop

Fig. 7.36 illustrates an example of a flip-flop which can be operate in different modes.

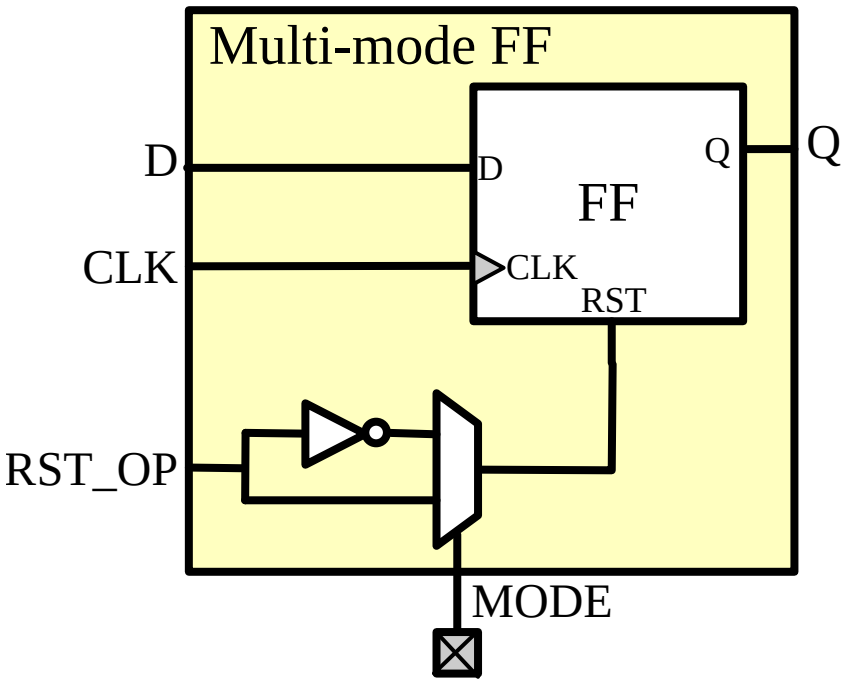


Fig. 7.36: An example of a flip-flop which can be operate in different modes

The code describing this FF is:

```
<circuit_model type="ff" name="frac_ff" prefix="frac_ff" verilog_netlist="frac_ff.v"␣
↪spice_netlist="frac_ff.sp">
  <port type="input" prefix="D" lib_name="D" size="1"/>
  <port type="input" prefix="Reset" lib_name="RST_OP" size="1" is_global="true"/>
  <port type="output" prefix="Q" lib_name="Q" size="1"/>
  <port type="clock" prefix="clock" lib_name="CLK" size="1" is_global="true"/>
  <port type="sram" prefix="MODE" lib_name="MODE" size="1" mode_select="true" circuit_
↪model_name="CCFF" default_value="0"/>
</circuit_model>
```

This example shows:

- A multi-mode flip-flop which is defined in a Verilog netlist `frac_ff.v` and a SPICE netlist `frac_ff.sp`
- The flip-flop has a `reset` pin which can be either active-low or active-high, depending on the mode selection pin `MODE`.
- The mode-selection bit will be generated by a configurable memory outside the flip-flop, which will be implemented by a circuit model `CCFF` defined by users (see an example in *Regular Configuration-chain Flip-flop*).
- The flip-flop port names defined differently in standard cell library and VPR architecture. The `lib_name` capture the port name defined in standard cells, while `prefix` capture the port name defined in `pb_type` of VPR architecture file

7.8.8 Configuration Chain Flip-Flop

Note: OpenFPGA does not auto-generate any netlist for configuration chain flip-flops. Users should define the HDL modeling in external netlists and ensure consistency to physical designs.

Template

```
<circuit_model type="ccff" name="<string>" prefix="<string>" spice_netlist="<string>"␣
↪verilog_netlist="<string>"/>
  <design_technology type="cmos"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
  <port type="clock" prefix="<string>" size="<int>"/>
</circuit_model>
```

Note: The circuit designs of configurable memory elements are highly dependent on the technology node and well optimized by engineers. Therefore, FPGA-Verilog/SPICE requires users to provide their customized FF Verilog/SPICE/Verilog netlists. A sample Verilog/SPICE netlist of FF can be found in the directory SpiceNetlists in the released package.

The information of input and output buffer should be clearly specified according to the customized SPICE netlist! The existence of input/output buffers will influence the decision in creating SPICE testbenches, which may leads to larger errors in power analysis.

Note: FPGA-Verilog/SPICE currently support only one clock domain for any configuration protocols in the FPGA. Therefore there should be only one clock port to be defined and the size of the clock port should be 1.

Note: A flip-flop should at least have three types of ports, `input`, `output` and `clock`.

Note: If the user provides a customized Verilog/SPICE netlist, the bandwidth of ports should be defined to the same as the Verilog/SPICE netlist.

Note: In a valid FPGA architecture, users should provide at least either a `ccff` or `sram` circuit model, so that the configurations can loaded to core logic.

Regular Configuration-chain Flip-flop

Fig. 7.37 illustrates an example of standard flip-flops used to build a configuration chain.

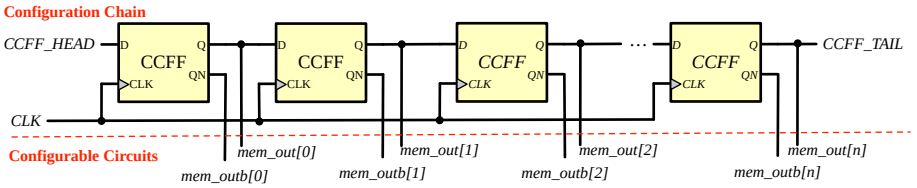


Fig. 7.37: An example of a Flip-Flop organized in a chain.

The code describing this FF is:

```
<circuit_model type="ccff" name="ccff" prefix="ccff" verilog_netlist="ccff.v" spice_
↪netlist="ccff.sp">
  <port type="input" prefix="D" size="1"/>
  <port type="output" prefix="Q" size="1"/>
  <port type="output" prefix="QN" size="1"/>
  <port type="clock" prefix="CK" size="1" is_global="true" is_prog="true" is_clock="true
↪"/>
</circuit_model>
```

This example shows:

- A configuration-chain flip-flop which is defined in a Verilog netlist `ccff.v` and a SPICE netlist `ccff.sp`
- The flip-flop has a global clock port, `CK`, which will be wired a global programming clock

Note:

The output ports of the configuration flip-flop must follow a fixed sequence in definition:

- The first output port **MUST** be the data output port, e.g., `Q`.
- The second output port **MUST** be the **inverted** data output port, e.g., `QN`.

Configuration-chain Flip-flop with Configure Enable Signals

Configuration chain could be built with flip-flops with outputs that are enabled by specific signals. Consider the example in Fig. 7.38, the flip-flop has

- a configure enable signal `CFG_EN` to release the data output `Q` and `QN`
- a pair of data outputs `Q` and `QN` which are controlled by the configure enable signal `CFG_EN`
- a regular data output `SCAN_Q` which outputs registered data

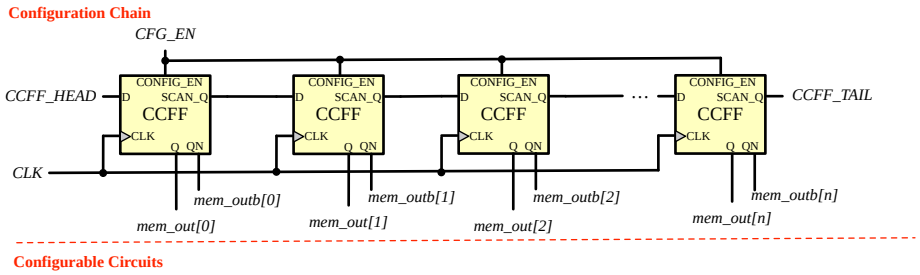


Fig. 7.38: An example of a Flip-Flop with config enable feature organized in a chain.

The code describing this FF is:

```
<circuit_model type="ccff" name="ccff" prefix="ccff" verilog_netlist="ccff.v" spice_
↪netlist="ccff.sp">
  <port type="input" prefix="CFG_EN" size="1" is_global="true" is_config_enable="true"/>
  <port type="input" prefix="D" size="1"/>
  <port type="output" prefix="SCAN_Q" size="1"/>
  <port type="output" prefix="QN" size="1"/>
  <port type="output" prefix="Q" size="1"/>
  <port type="clock" prefix="CK" size="1" is_global="true" is_prog="true" is_clock="true
↪"/>
</circuit_model>
```

Note:

The output ports of the configuration flip-flop must follow a fixed sequence in definition:

- The first output port **MUST** be the regular data output port, e.g., `SCAN_Q`.
- The second output port **MUST** be the **inverted** data output port which is activated by the configure enable signal, e.g., `QN`.
- The second output port **MUST** be the data output port which is activated by the configure enable signal, e.g., `Q`.

Configuration-chain Flip-flop with Scan Input

Configuration chain could be built with flip-flops with a scan chain input . Consider the example in Fig. 7.39, the flip-flop has

- an additional input `SI` to enable scan-chain capabaility
- a configure enable signal `CFG_EN` to release the data output `Q` and `QN`
- a pair of data outputs `Q` and `QN` which are controlled by the configure enable signal `CFG_EN`
- a regular data output `SCAN_Q` which outputs registered data

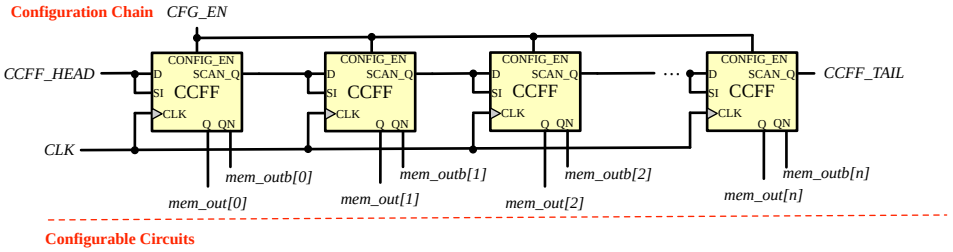


Fig. 7.39: An example of a Flip-Flop with scan input organized in a chain.

The code describing this FF is:

```
<circuit_model type="ccff" name="ccff" prefix="ccff" verilog_netlist="ccff.v" spice_
↪netlist="ccff.sp">
  <port type="input" prefix="CFG_EN" size="1" is_global="true" is_config_enable="true"/>
  <port type="input" prefix="D" size="1"/>
  <port type="input" prefix="SI" size="1"/>
  <port type="output" prefix="SCAN_Q" size="1"/>
  <port type="output" prefix="QN" size="1"/>
  <port type="output" prefix="Q" size="1"/>
  <port type="clock" prefix="CK" size="1" is_global="true" is_prog="true" is_clock="true
↪"/>
</circuit_model>
```

Note:

The input ports of the configuration flip-flop must follow a fixed sequence in definition:

- The first input port **MUST** be the regular data input port, e.g., `D`.
- The second input port **MUST** be the scan input port, e.g., `SI`.

7.8.9 Hard Logics

Note: OpenFPGA does not auto-generate any netlist for the hard logics. Users should define the HDL modeling in external netlists and ensure consistency to physical designs.

Template

```
<circuit_model type="hardlogic" name="<string>" prefix="<string>" verilog_netlist="
↪<string>" spice_netlist="<string>"/>
  <design_technology type="cmos"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
</circuit_model>
```

Note: Hard logics are defined for non-configurable resources in FPGA architectures, such as adders, multipliers and RAM blocks. Their circuit designs are highly dependent on the technology node and well optimized by engineers. As more functional units are included in FPGA architecture, it is impossible to auto-generate these functional units. Therefore, FPGA-Verilog/SPICE requires users to provide their customized Verilog/SPICE netlists.

Note: Examples can be found in hard_logic_example_link

Note: The information of input and output buffer should be clearly specified according to the customized Verilog/SPICE netlist! The existence of input/output buffers will influence the decision in creating SPICE testbenches, which may leads to larger errors in power analysis.

Full Adder

The code describing the 1-bit full adder is:

```
<circuit_model type="hard_logic" name="adder" prefix="adder" spice_netlist="adder.sp"␣
↪verilog_netlist="adder.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <port type="input" prefix="a" size="1"/>
  <port type="input" prefix="b" size="1"/>
  <port type="input" prefix="cin" size="1"/>
  <port type="output" prefix="cout" size="1"/>
  <port type="output" prefix="sumout" size="1"/>
</circuit_model>
```

This example shows:

- A 1-bit full adder which is defined in a Verilog netlist `adder.v` and a SPICE netlist `adder.sp`

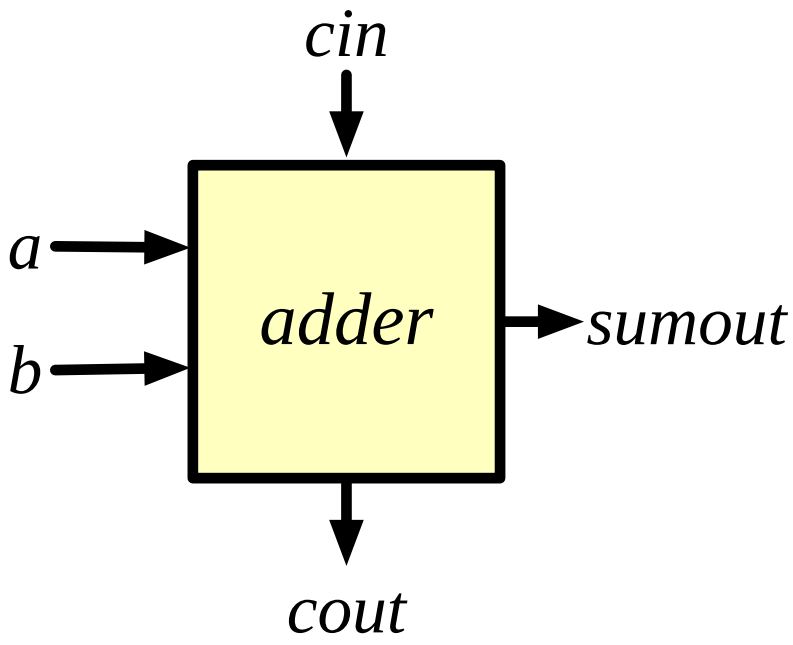


Fig. 7.40: An example of a 1-bit full adder.

- The adder has three 1-bit inputs, i.e., `a`, `b` and `cin`, and two 2-bit outputs, i.e., `cout`, `sumout`.

Multiplier

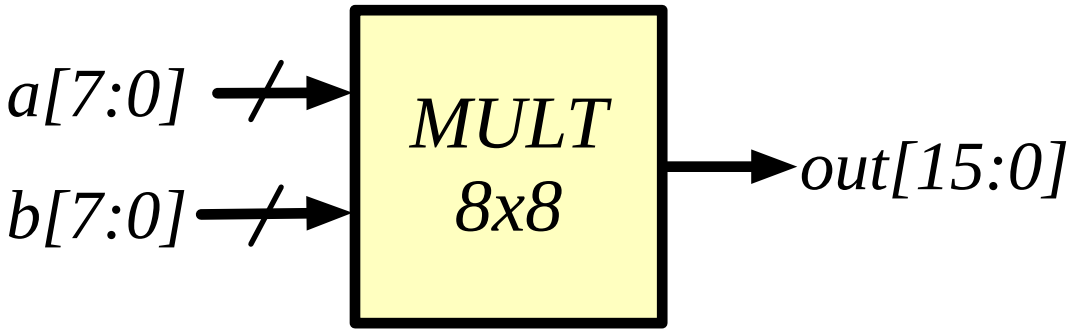


Fig. 7.41: An example of a 8-bit multiplier.

The code describing the multiplier is:

```
<circuit_model type="hard_logic" name="mult8x8" prefix="mult8x8" spice_netlist="dsp.sp"␣
↪verilog_netlist="dsp.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <port type="input" prefix="a" size="8"/>
  <port type="input" prefix="b" size="8"/>
  <port type="output" prefix="out" size="16"/>
</circuit_model>
```

This example shows:

- A 8-bit multiplier which is defined in a Verilog netlist `dsp.v` and a SPICE netlist `dsp.sp`

Multi-mode Multiplier

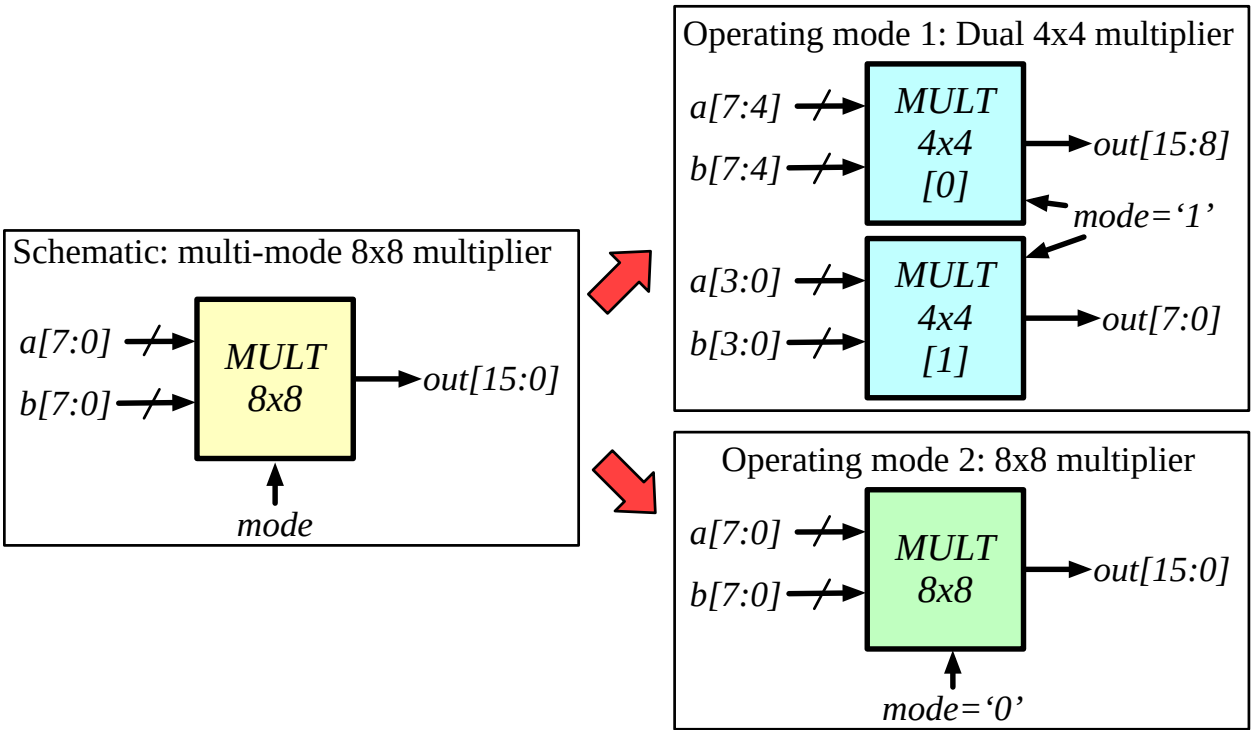


Fig. 7.42: An example of a 8-bit multiplier which can operating in two modes: (1) dual 4-bit multipliers; and (2) 8-bit multiplier

The code describing the multiplier is:

```
<circuit_model type="hard_logic" name="frac_mult8x8" prefix="frac_mult8x8" spice_netlist=
↪"dsp.sp" verilog_netlist="dsp.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <port type="input" prefix="a" size="8"/>
  <port type="input" prefix="b" size="8"/>
  <port type="output" prefix="out" size="16"/>
  <port type="sram" prefix="mode" size="1" mode_select="true" circuit_model_name="CCFF"␣
↪default_value="0"/>
</circuit_model>
```

This example shows:

- A multi-mode 8-bit multiplier which is defined in a Verilog netlist `dsp.v` and a SPICE netlist `dsp.sp`
- The multi-mode multiplier can operating in two modes: (1) dual 4-bit multipliers; and (2) 8-bit multiplier
- The mode-selection bit will be generated by a configurable memory outside the flip-flop, which will be implemented by a circuit model `CCFF` defined by users (see an example in *Regular Configuration-chain Flip-flop*).

Dual Port Block RAM

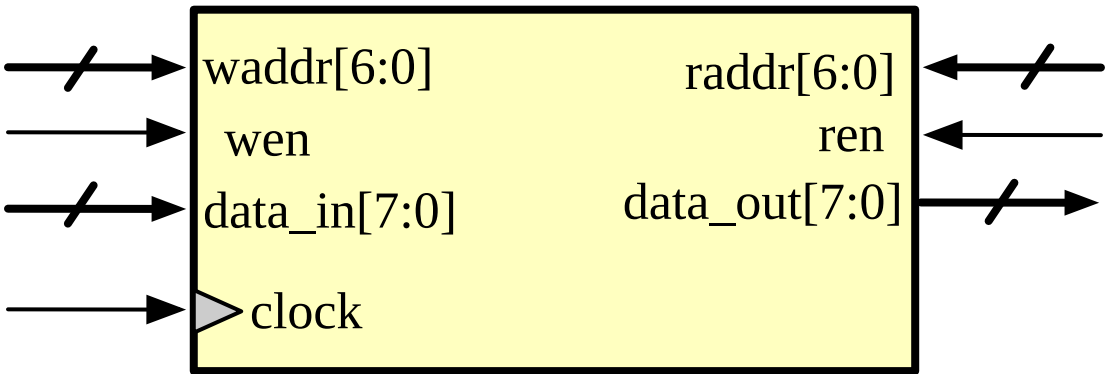


Fig. 7.43: An example of a dual port block RAM with 128 addresses and 8-bit data width.

The code describing this block RAM is:

```
<circuit_model type="hard_logic" name="dpram_128x8" prefix="dpram_128x8" spice_netlist=
↪"dpram.sp" verilog_netlist="dpram.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <port type="input" prefix="waddr" size="7"/>
  <port type="input" prefix="raddr" size="7"/>
  <port type="input" prefix="data_in" size="8"/>
  <port type="input" prefix="wen" size="1"/>
  <port type="input" prefix="ren" size="1"/>
  <port type="output" prefix="data_out" size="8"/>
  <port type="clock" prefix="clock" size="1" is_global="true" default_val="0"/>
</circuit_model>
```

This example shows:

- A 128x8 dual port RAM which is defined in a Verilog netlist `dpram.v` and a SPICE netlist `dpram.sp`
- The clock port of the RAM is controlled by a global signal (see details about global signal definition in *Physical Tile Annotation*).

Multi-mode Dual Port Block RAM

The code describing this block RAM is:

```
<circuit_model type="hard_logic" name="frac_dpram_128x8" prefix="frac_dpram_128x8" spice_
↪netlist="frac_dpram.sp" verilog_netlist="frac_dpram.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="inv1x"/>
  <output_buffer exist="true" circuit_model_name="inv1x"/>
  <port type="input" prefix="waddr" size="8"/>
  <port type="input" prefix="raddr" size="8"/>
  <port type="input" prefix="data_in" size="8"/>
  <port type="input" prefix="wen" size="1"/>
  <port type="input" prefix="ren" size="1"/>
```

(continues on next page)

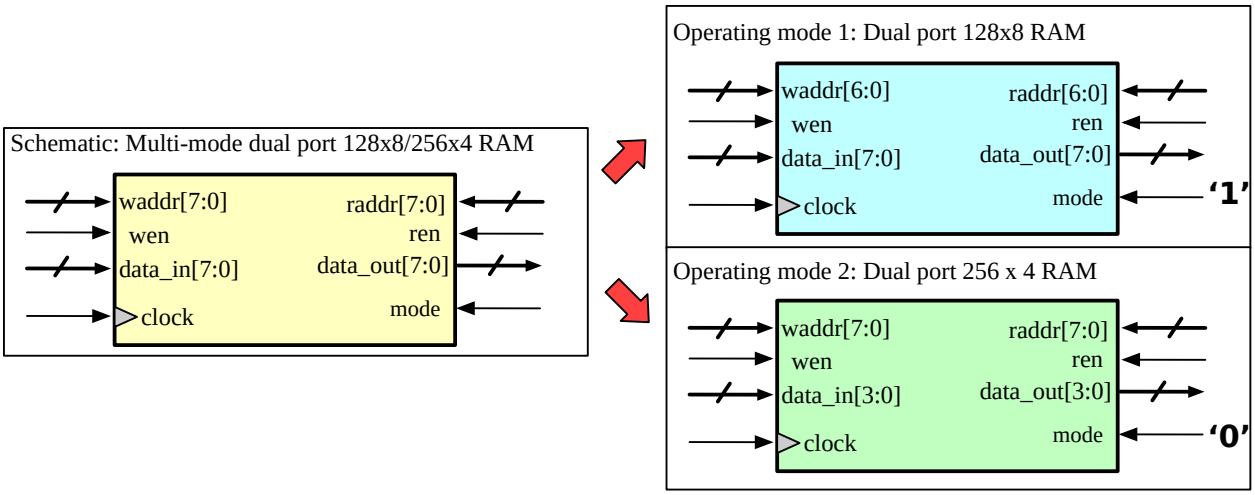


Fig. 7.44: An example of a dual port block RAM which can operate in two modes: 128x8 and 256x4.

(continued from previous page)

```
  <port type="output" prefix="data_out" size="8"/>
  <port type="clock" prefix="clock" size="1" is_global="true" default_val="0"/>
  <port type="sram" prefix="mode" size="1" mode_select="true" circuit_model_name="CCFF"␣
↪default_value="0"/>
</circuit_model>
```

This example shows:

- A fracturable dual port RAM which is defined in a Verilog netlist `frac_dpram.v` and a SPICE netlist `frac_dpram.sp`
- The dual port RAM can operate in two modes: (1) 128 addresses with 8-bit data width; (2) 256 addresses with 4-bit data width
- The clock port of the RAM is controlled by a global signal (see details about global signal definition in *Physical Tile Annotation*).
- The mode-selection bit will be generated by a configurable memory outside the flip-flop, which will be implemented by a circuit model `CCFF` defined by users (see an example in *Regular Configuration-chain Flip-flop*).

7.8.10 Routing Wire Segments

FPGA architecture requires two type of wire segments:

- `wire`, which targets the local wires inside the logic blocks. The wire has one input and one output, directly connecting the output of a driver and the input of the downstream unit, respectively
- `chan_wire`, especially targeting the channel wires. The channel wires have one input and two outputs, one of which is connected to the inputs of Connection Boxes while the other is connected to the inputs of Switch Boxes. Two outputs are created because from the view of layout, the inputs of Connection Boxes are typically connected to the middle point of channel wires, which has less parasitic resistances and capacitances than connected to the ending point.

Template

```
<circuit_model type="wire|cham_wire" name="<string>" prefix="<string>" spice_netlist="
↪<string>" verilog_netlist="<string>"/>
  <design_technology type="cmos"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
  <wire_param model_type="<string>" R="<float>" C="<float>" num_level="<int>"/>
</circuit_model>
```

Note: FPGA-Verilog/SPICE can auto-generate the Verilog/SPICE model for wires while also allows users to provide their customized Verilog/SPICE netlists.

Note: The information of input and output buffer should be clearly specified according to the customized netlist! The existence of input/output buffers will influence the decision in creating testbenches, which may leads to larger errors in power analysis.

`<wire_param model_type="<string>" R="<float>" C="<float>" num_level="<int>"/>`

- `model_type="pi|T"` Specify the type of RC models for this wire segement. Currently, OpenFPGA supports the π-type and T-type multi-level RC models.
- `R="<float>"` Specify the total resistance of the wire
- `C="<float>"` Specify the total capacitance of the wire.
- `num_level="<int>"` Specify the number of levels of the RC wire model.

Note: wire parameters are essential for FPGA-SPICE to accurately model wire parasitics

Routing Track Wire Example

Fig. 7.45 depicts the modeling for a length-2 channel wire.

The code describing this wire is:

```
<circuit_model type="chan_wire" name="segment0" prefix="chan_wire"/>
  <design_technology type="cmos"/>
  <port type="input" prefix="mux_out" size="1"/>
  <port type="output" prefix="cb_sb" size="1"/>
  <wire_param model_type="pi" res_val="103.84" cap_val="13.80e-15" level="1"/>
</circuit_model>
```

This example shows

- A routing track wire has 1 input and output
- The routing wire will be modelled as a 1-level π-type RC wire model with a total resistance of 103.84Ω and a total capacitance of $13.89fF$

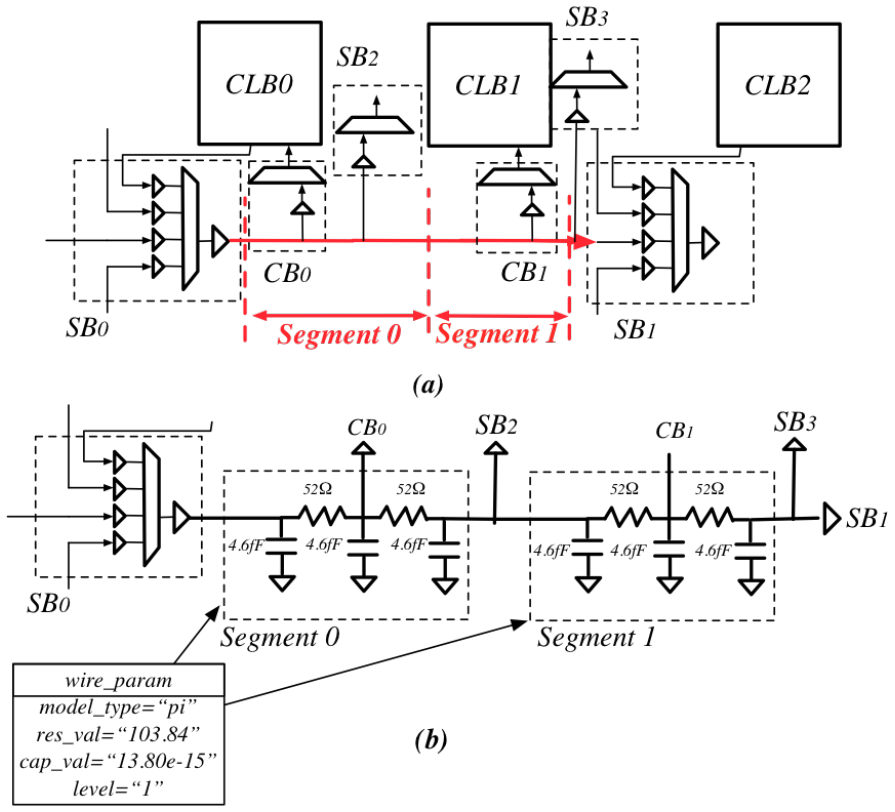


Fig. 7.45: An example of a length-2 channel wire modeling

7.8.11 I/O pads

Note: OpenFPGA does not auto-generate any netlist for I/O cells. Users should define the HDL modeling in external netlists and ensure consistency to physical designs.

Template

```
<circuit_model type="iopad" name="<string>" prefix="<string>" spice_netlist="<string>"␣
↪verilog_netlist="<string>"/>
  <design_technology type="cmos"/>
  <input_buffer exist="<string>" circuit_model_name="<string>"/>
  <output_buffer exist="<string>" circuit_model_name="<string>"/>
  <port type="input" prefix="<string>" size="<int>"/>
  <port type="output" prefix="<string>" size="<int>"/>
  <port type="sram" prefix="<string>" size="<int>" mode_select="<bool>" circuit_model_
↪name="<string>" default_val="<int>"/>
</circuit_model>
```

Note: The circuit designs of I/O pads are highly dependent on the technology node and well optimized by engineers. Therefore, FPGA-Verilog/SPICE requires users to provide their customized Verilog/SPICE/Verilog netlists. A sample Verilog/SPICE netlist of an I/O pad can be found in the directory SpiceNetlists in the released package.

Note: The information of input and output buffer should be clearly specified according to the customized netlist! The existence of input/output buffers will influence the decision in creating testbenches, which may leads to larger errors in power analysis.

General Purpose I/O

Fig. 7.46 depicts a general purpose I/O pad.

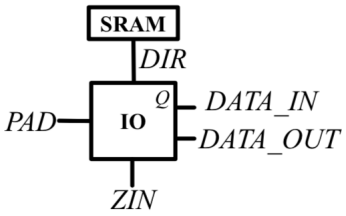


Fig. 7.46: An example of an IO-Pad

The code describing this I/O-Pad is:

```
<circuit_model type="iopad" name="iopad" prefix="iopad" spice_netlist="io.sp" verilog_
↪netlist="io.v">
  <design_technology type="cmos"/>
  <input_buffer exist="true" circuit_model_name="INVTX1"/>
```

(continues on next page)

(continued from previous page)

```
  <output_buffer exist="true" circuit_model_name="INVTX1"/>
  <pass_gate_logic circuit_model_name="TGATE"/>
  <port type="inout" prefix="pad" size="1" is_global="true" is_io="true" is_data_io="true
↪"/>
  <port type="sram" prefix="en" size="1" mode_select="true" circuit_model_name="ccff"␣
↪default_val="1"/>
  <port type="input" prefix="outpad" size="1"/>
  <port type="output" prefix="inpad" size="1"/>
</circuit_model>
```

This example shows

- A general purpose I/O cell defined in Verilog netlist `io.sp` and SPICE netlist `io.sp`
- The I/O cell has an `inout` port as the bi-directional port
- The directionality of I/O can be controlled by a configuration-chain flip-flop defined in circuit model `ccff`
- If unused, the I/O cell will be configured to 1

7.9 Bind circuit modules to VPR architecture

Each defined circuit model should be linked to an FPGA module defined in the original part of architecture descriptions. It helps FPGA-circuit creating the circuit netlists for logic/routing blocks. Since the original part lacks such support, we create a few XML properties to link to Circuit models.

7.9.1 Switch Blocks

Original VPR architecture description contains an XML node called switchlist under which all the multiplexers of switch blocks are described. To link a defined circuit model to a multiplexer in the switch blocks, a new XML property circuit_model_name should be added to the descriptions.

Here is an example:

```
<switch_block>
  <switch type="mux" name="<string>" circuit_model_name="<string>"/>
</switch_block>
```

- `circuit_model_name="<string>"` should match a circuit model whose type is `mux` defined in *Circuit Library*.

7.9.2 Connection Blocks

To link the defined circuit model of the multiplexer to the Connection Blocks, a `circuit_model_name` should be annotated to the definition of Connection Blocks switches.

Here is the example:

```
<connection_block>
  <switch type="ipin_cblock" name="<string>" circuit_model_name="<string>"/>
</connection_block>
```

- `circuit_model_name="<string>"` should match a circuit model whose type is `mux` defined in *Circuit Library*.

7.9.3 Channel Wire Segments

Similar to the Switch Boxes and Connection Blocks, the channel wire segments in the original architecture descriptions can be adapted to provide a link to the defined circuit model.

```
<segmentlist>
  <segment name="<string>" circuit_model_name="<string>"/>
</segmentlist>
```

- `circuit_model_name="<string>"` should match a circuit model whose type is `chan_wire` defined in *Circuit Library*.

7.9.4 Physical Tile Annotation

Original VPR architecture description contains `<tile>` XML nodes to define physical tile pins. OpenFPGA allows users to define pin/port of physical tiles as global ports.

Here is an example:

```
<tile_annotations>
  <global_port name="<string>" is_clock="<bool>" is_reset="<bool>" is_set="<bool>"␣
↪default_val="<int>">
    <tile name="<string>" port="<string>" x="<int>" y="<int>"/>
    ...
  </global_port>
</tile_annotations>
```

- `name="<string>"` is the port name to appear in the top-level FPGA fabric.
- `is_clock="<bool>"` define if the global port is a clock port at the top-level FPGA fabric. An operating clock port will be driven by proper signals in auto-generated testbenches.
- `is_reset="<bool>"` define if the global port is a reset port at the top-level FPGA fabric. An operating reset port will be driven by proper signals in testbenches.
- `is_set="<bool>"` define if the global port is a set port at the top-level FPGA fabric. An operating set port will be driven by proper signals in testbenches.

Note: A port can only be defined as `clock` or `set` or `reset`.

Note: All the global port from a physical tile port is only used in operating phase. Any ports for programmable use are not allowed!

- `default_val="<int>"` define if the default value for the global port when initialized in testbenches. Valid values are either `0` or `1`. For example, the default value of an active-high reset pin is `0`, while an active-low reset pin is `1`.

Note: A global port could be connected from different tiles by defining multiple <tile> lines under a global port!!!

`<tile name="<string>" port="<string>" x="<int>" y="<int>"/>`

- `name="<string>"` is the name of a physical tile, e.g., `name="clb"`.

- `port="<string>"` is the port name of a physical tile, e.g., `port="clk[0:3]"`.
- `x="<int>"` is the x coordinate of a physical tile, e.g., `x="1"`. If the x coordinate is set to `-1`, it means all the valid x coordinates of the selected physical tile in the FPGA device will be considered.
- `y="<int>"` is the y coordinate of a physical tile, e.g., `y="1"`. If the y coordinate is set to `-1`, it means all the valid y coordinates of the selected physical tile in the FPGA device will be considered.

Note: The port of physical tile must be a valid port of the physical definition in VPR architecture! If you define a multi-bit port, it must be explicitly defined in the port, e.g., clk[0:3], which must be in the range of the port definition in physical tiles of VPR architecture files!!!

Note: The linked port of physical tile must meet the following requirements:

- If the `global_port` is set as clock through `is_clock="true"`, the port of the physical tile must also be a clock port.
- If not a clock, the port of the physical tile must be defined as non-clock global
- The port of the physical tile should have zero connectivity (`Fc=0`) in VPR architecture

---

A more illustrative example:

Fig. 7.47 illustrates the difference between the global ports defined through `circuit_model` and `tile_annotation`.

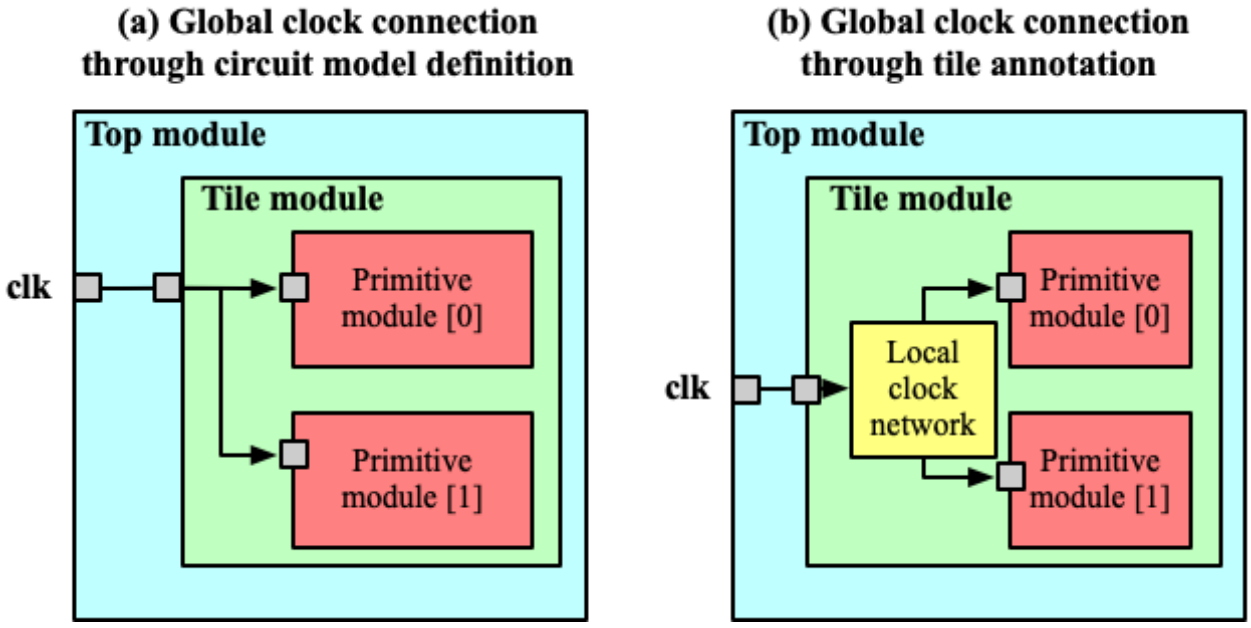


Fig. 7.47: Difference between global port definition through circuit model and tile annotation

When a global port, e.g., `clk`, is defined in `circuit_model` using the following code:

```
<circuit_model>
  <port name="clk" is_global="true" is_clock="true"/>
</circuit_model>
```

Dedicated feedthrough wires will be created across all the modules from top-level to primitive.

When a global port, e.g., `clk`, is defined in `tile_annotation` using the following code:

```
<tile_annotations>
  <global_port name="clk" is_clock="true">
    <tile name="clb" port="clk"/>
  </global_port>
</tile_annotations>
```

Note that a global port can also be defined to drive only a partial bit of a port of a physical tile.

```
<tile_annotations>
  <global_port name="clk" is_clock="true">
    <tile name="clb" port="clk[3:3]"/>
  </global_port>
</tile_annotations>
```

Clock port `clk` of each `clb` tile will be connected to a common clock port of the top module, while local clock network is customizable through VPR's architecture description language. For instance, the local clock network can be a programmable clock network.

7.9.5 Primitive Blocks inside Multi-mode Configurable Logic Blocks

The architecture description employs a hierarchy of `pb_types` to depict the sub-modules and complex interconnections inside logic blocks. Each leaf node and interconnection in the pb_type hierarchy should be linked to a circuit model. Each primitive block, i.e., the leaf `pb_types`, should be linked to a valid circuit model, using the XML syntax `circuit_model_name`. The `circuit_model_name` should match the given name of a `circuit_model` defined by users.

```
<pb_type_annotations>
  <!-- physical pb_type binding in complex block IO -->
  <pb_type name="io" physical_mode_name="physical"/>
  <pb_type name="io[physical].iopad" circuit_model_name="iopad" mode_bits="1"/>
  <pb_type name="io[inpad].inpad" physical_pb_type_name="io[physical].iopad" mode_bits="1
↪"/>
  <pb_type name="io[outpad].outpad" physical_pb_type_name="io[physical].iopad" mode_bits=
↪"0"/>
  <!-- End physical pb_type binding in complex block IO -->

  <!-- physical pb_type binding in complex block CLB -->
  <!-- physical mode will be the default mode if not specified -->
  <pb_type name="clb">
    <!-- Binding interconnect to circuit models as their physical implementation, if not␣
↪defined, we use the default model -->
    <interconnect name="crossbar" circuit_model_name="mux_2level"/>
  </pb_type>
  <pb_type name="clb.fle" physical_mode_name="physical"/>
  <pb_type name="clb.fle[physical].fabric.frac_logic.frac_lut6" circuit_model_name="frac_
↪lut6" mode_bits="0"/>
  <pb_type name="clb.fle[physical].fabric.ff" circuit_model_name="static_dff"/>
  <!-- Binding operating pb_type to physical pb_type -->
  <pb_type name="clb.fle[n2_lut5].lut5inter.ble5.lut5" physical_pb_type_name="clb.
```

(continues on next page)

(continued from previous page)

```
↪fle[physical].fabric.frac_logic.frac_lut6" mode_bits="1" physical_pb_type_index_factor=
↪"0.5">
    <!-- Binding the lut5 to the first 5 inputs of fracturable lut6 -->
    <port name="in" physical_mode_port="in[0:4]"/>
    <port name="out" physical_mode_port="lut5_out" physical_mode_pin_rotate_offset="1"/>
  </pb_type>
  <pb_type name="clb.fle[n2_lut5].lut5inter.ble5.ff" physical_pb_type_name="clb.
↪fle[physical].fabric.ff"/>
  <pb_type name="clb.fle[n1_lut6].ble6.lut6" physical_pb_type_name="clb.fle[physical].
↪fabric.frac_logic.frac_lut6" mode_bits="0">
    <!-- Binding the lut6 to the first 6 inputs of fracturable lut6 -->
    <port name="in" physical_mode_port="in[0:5]"/>
    <port name="out" physical_mode_port="lut6_out"/>
  </pb_type>
  <pb_type name="clb.fle[n1_lut6].ble6.ff" physical_pb_type_name="clb.fle[physical].
↪fabric.ff" physical_pb_type_index_factor="2" physical_pb_type_index_offset="0"/>
  <!-- End physical pb_type binding in complex block IO -->
</pb_type_annotations>
```

<pb_type name="<string>" physical_mode_name="<string>">

Specify a physical mode for multi-mode pb_type defined in VPR architecture.

Note: This should be applied to non-primitive pb_type, i.e., pb_type have child pb_type.

- name="<string>" specifiy the full name of a pb_type in the hierarchy of VPR architecture.
- physical_mode_name="<string>" Specify the name of the mode that describes the physical implementation of the configurable block. This is critical in modeling actual circuit designs and architecture of an FPGA. Typically, only one physical_mode should be specified for each multi-mode pb_type.

Note: OpenFPGA will infer the physical mode for a single-mode pb_type defined in VPR architecture

---

<pb_type name="<string>" physical_pb_type_name="<string>"
circuit_model_name="<string>" mode_bits="<int>"
physical_pb_type_index_factor="<float>" physical_pb_type_index_offset="<int>">

Specify the physical implementation for a primitive pb_type in VPR architecture

---

Note: This should be applied to primitive pb_type, i.e., pb_type have no children.

Note: This definition should be placed directly under the XML node <pb_type_annotation> without any intermediate XML nodes!

- name="<string>" specifiy the full name of a pb_type in the hierarchy of VPR architecture.
- physical_pb_type_name=<string> creates the link on pb_type between operating and physical modes. This syntax is mandatory for every primitive pb_type in an operating mode pb_type. It should be a valid name of primitive pb_type in physical mode.

- circuit_model_name="<string>" Specify a circuit model to implement a pb_type in VPR architecture. The circuit_model_name is mandatory for every primitive``pb_type`` in a physical_mode pb_type.
- mode_bits="<int>" Specify the configuration bits for the circuit_model when operating at an operating mode. The length of mode_bits should match the port size defined in circuit_model. The mode_bits should be derived from circuit designs while users are responsible for its correctness. FPGA-Bitstreamm will add the mode_bits during bitstream generation.
- physical_pb_type_index_factor="<float>" aims to align the indices for pb_type between operating and physical modes, especially when an operating mode contains multiple pb_type (num_pb>1) that are linked to the same physical pb_type. When physical_pb_type_name is larger than 1, the index of pb_type will be multipled by the given factor.
- physical_pb_type_index_offset=<int> aims to align the indices for pb_type between operating and physical modes, especially when an operating mode contains multiple pb_type (num_pb>1) that are linked to the same physical pb_type. When physical_pb_type_name is larger than 1, the index of pb_type will be shifted by the given factor.

<interconnect name="<string>" circuit_model_name="<string>">

- name="<string>" specify the name of a interconnect in VPR architecture. Different from pb_type, hierarchical name is not required here.
- circuit_model_name="<string>" For the interconnection type direct, the type of the linked circuit model should be wire. For multiplexers, the type of linked circuit model should be mux. For complete, the type of the linked circuit model can be either mux or wire, depending on the case.

Note: A <pb_type name="<string>"> parent XML node is required for the interconnect-to-circuit bindings whose interconnects are defined under the pb_type in VPR architecture description.

<port name="<string>" physical_mode_port="<string>"
physical_mode_pin_initial_offset="<int>"
physical_mode_pin_rotate_offset="<int>"/>
physical_mode_port_rotate_offset="<int>"/>

Link a port of an operating pb_type to a port of a physical pb_type

- name="<string>" specifiy the name of a port in VPR architecture. Different from pb_type, hierarchical name is not required here.
- physical_mode_pin="<string>" creates the link of ``port of pb_type between operating and physical modes. This syntax is mandatory for every primitive pb_type in an operating mode pb_type. It should be a valid port name of leaf pb_type in physical mode and the port size should also match.

Note: Users can define multiple ports. For example: physical_mode_pin="a[0:1] b[2:2]". When multiple ports are used, the physical_mode_pin_initial_offset and physical_mode_pin_rotate_offset should also be adapt. For example: physical_mode_pin_rotate_offset="1 0")

- physical_mode_pin_initial_offset="<int>" aims to align the pin indices for port of pb_type between operating and physical modes, especially when part of port of operating mode is mapped to a port in physical pb_type. When physical_mode_pin_initial_offset is larger than zero, the pin index of pb_type (whose index is large than 1) will be shifted by the given offset.

Note: A quick example to understand the initial offset For example, an initial offset of -32 is used to map

- operating pb_type `bram[0].dout[32]` with a full path `memory[dual_port].bram[0]`
- operating pb_type `bram[0].dout[33]` with a full path `memory[dual_port].bram[0]`

to

- physical pb_type `bram[0].dout_a[0]` with a full path `memory[physical].bram[0]`
- physical pb_type `bram[0].dout_a[1]` with a full path `memory[physical].bram[0]`

Note: If not defined, the default value of `physical_mode_pin_initial_offset` is set to `0`.

- `physical_mode_pin_rotate_offset="<int>"` aims to align the pin indices for `port` of `pb_type` between operating and physical modes, especially when an operating mode contains multiple `pb_type` (`num_pb>1`) that are linked to the same physical `pb_type`. When `physical_mode_pin_rotate_offset` is larger than zero, the pin index of `pb_type` (whose index is large than 1) will be shifted by the given offset, **each time a pin in the operating mode is binded to a pin in the physical mode**.

Note: A quick example to understand the rotate offset For example, a rotating offset of 9 is used to map

- operating pb_type `mult_9x9[0].a[0]` with a full path `mult[frac].mult_9x9[0]`
- operating pb_type `mult_9x9[1].a[1]` with a full path `mult[frac].mult_9x9[1]`

  to

- physical pb_type `mult_36x36.a[0]` with a full path `mult[physical].mult_36x36[0]`
- physical pb_type `mult_36x36.a[9]` with a full path `mult[physical].mult_36x36[0]`

Note: If not defined, the default value of `physical_mode_pin_rotate_offset` is set to `0`.

Warning: The result of using `physical_mode_pin_rotate_offset` is fundementally different than `physical_mode_port_rotate_offset`!!! Please read the examples carefully and pick the one fitting your needs.

- `physical_mode_port_rotate_offset="<int>"` aims to align the port indices for `port` of `pb_type` between operating and physical modes, especially when an operating mode contains multiple `pb_type` (`num_pb>1`) that are linked to the same physical `pb_type`. When `physical_mode_port_rotate_offset` is larger than zero, the pin index of `pb_type` (whose index is large than 1) will be shifted by the given offset, **only when all the pins of a port in the operating mode is binded to all the pins of a port in the physical mode**.

Note: A quick example to understand the rotate offset For example, a rotating offset of 9 is used to map

- operating pb_type `mult_9x9[0].a[0:8]` with a full path `mult[frac].mult_9x9[0]`
- operating pb_type `mult_9x9[1].a[0:8]` with a full path `mult[frac].mult_9x9[1]`

to

- physical pb_type `mult_36x36.a[0:8]` with a full path `mult[physical].mult_36x36[0]`
- physical pb_type `mult_36x36.a[9:17]` with a full path `mult[physical].mult_36x36[0]`

Note: If not defined, the default value of `physical_mode_port_rotate_offset` is set to `0`.

Note: It is highly recommended that only one physical mode is defined for a multi-mode configurable block. Try not to use nested physical mode definition. This will ease the debugging and lead to clean XML description.

Note: Be careful in using `physical_pb_type_index_factor`, `physical_pb_type_index_offset` and `physical_mode_pin_rotate_offset`! Try to avoid using them unless for highly complex configuration blocks with very deep hierarchy.

7.10 Fabric Key

Fabric key is a secure key for users to generate bitstream for a specific FPGA fabric. With this key, OpenFPGA can generate correct bitstreams for the FPGA. Using a wrong key, OpenFPGA may error out or generate wrong bitstreams. The fabric key support allows users to build secured/classified FPGA chips even with an open-source tool.

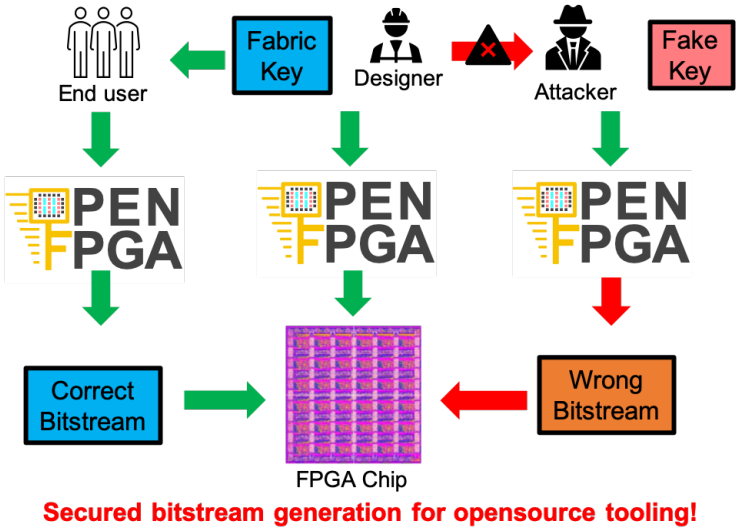


Fig. 7.48: The use of fabric key to secure the FPGA chip design

Note: Users are the only owner of the key. OpenFPGA will not store or replicate the key.

7.10.1 Key Generation

A fabric key can be achieved in the following ways:

- OpenFPGA can auto-generate a fabric key using random algorithms (see detail in *build_fabric*)
- Users can craft a fabric key based on auto-generated file by following the file format description.

7.10.2 File Format

See details in *Fabric Key (.xml)*

CHAPTER

EIGHT

OPENFPGA SHELL

8.1 Launch OpenFPGA Shell

OpenFPGA employs a shell-like user interface, in order to integrate all the tools in a well-modularized way. Currently, OpenFPGA shell is an unified platform to call `vpr`, `FPGA-Verilog`, `FPGA-Bitstream`, `FPGA-SDC` and `FPGA-SPICE`. To launch OpenFPGA shell, users can choose two modes.

`--interactive` `or -i`

Launch OpenFPGA in interactive mode where users type-in command by command and get runtime results

`--file` `or -f`

Launch OpenFPGA in script mode where users write commands in scripts and FPGA will execute them

`--batch_execution` `or -batch`

Execute OpenFPGA script in batch mode. This option is only valid for script mode.

- If in batch mode, OpenFPGA will abort immediately when fatal errors occurred.
- If not in batch mode, OpenFPGA will enter interactive mode when fatal errors occurred.

`--version` `or -v`

Print version information of OpenFPGA

`--help` `or -h`

Show the help desk

8.2 OpenFPGA Script Format

OpenFPGA accepts a simplified tcl-like script format.

`Comments`

Any content after a # will be treated as comments. Comments will not be executed.

> **Note:** comments can be added inline or as a new line. See the example below

`Continued` `line`

Lines to be continued should be finished with \. Continued lines will be conjuncted and executed as one line

Note: please ensure necessary spaces. Otherwise it may cause command parser fail.

The following is an example.

```
# Run VPR for the s298 design
vpr ./test_vpr_arch/k6_frac_N10_40nm.xml ./test_blif/and.blif --clock_modeling route #--
↪write_rr_graph example_rr_graph.xml

# Read OpenFPGA architecture definition
read_openfpga_arch -f ./test_openfpga_arch/k6_frac_N10_40nm_openfpga.xml

# Write out the architecture XML as a proof
#write_openfpga_arch -f ./arch_echo.xml

# Annotate the OpenFPGA architecture to VPR data base
link_openfpga_arch --activity_file ./test_blif/and.act \
                   --sort_gsb_chan_node_in_edges #--verbose

# Check and correct any naming conflicts in the BLIF netlist
check_netlist_naming_conflict --fix --report ./netlist_renaming.xml

# Apply fix-up to clustering nets based on routing results
pb_pin_fixup --verbose

# Apply fix-up to Look-Up Table truth tables based on packing results
lut_truth_table_fixup #--verbose

# Build the module graph
#  - Enabled compression on routing architecture modules
#  - Enable pin duplication on grid modules
build_fabric --compress_routing \
             --duplicate_grid_pin #--verbose

# Repack the netlist to physical pbs
# This must be done before bitstream generator and testbench generation
# Strongly recommend it is done after all the fix-up have been applied
repack #--verbose

# Build the bitstream
#  - Output the fabric-independent bitstream to a file
build_architecture_bitstream --verbose \
                             --file /var/tmp/xtang/openfpga_test_src/fabric_indepenent_
↪bitstream.xml

# Build fabric-dependent bitstream
build_fabric_bitstream --verbose

# Write the Verilog netlist for FPGA fabric
#  - Enable the use of explicit port mapping in Verilog netlist
write_fabric_verilog --file /var/tmp/xtang/openfpga_test_src/SRC \
                     --explicit_port_mapping \
                     --include_timing \
```

(continues on next page)

(continued from previous page)

```
                    --include_signal_init \
                    --support_icarus_simulator \
                    --print_user_defined_template \
                    --verbose

# Write the Verilog testbench for FPGA fabric
#  - We suggest the use of same output directory as fabric Verilog netlists
#  - Must specify the reference benchmark file if you want to output any testbenches
#  - Enable top-level testbench which is a full verification including programming␣
↪circuit and core logic of FPGA
#  - Enable pre-configured top-level testbench which is a fast verification skipping␣
↪programming phase
#  - Simulation ini file is optional and is needed only when you need to interface␣
↪different HDL simulators using openfpga flow-run scripts
write_verilog_testbench --file /var/tmp/xtang/openfpga_test_src/SRC \
                        --reference_benchmark_file_path /var/tmp/xtang/and.v \
                        --print_top_testbench \
                        --print_preconfig_top_testbench \
                        --print_simulation_ini /var/tmp/xtang/openfpga_test_src/
↪simulation_deck.ini

# Write the SDC files for PnR backend
#  - Turn on every options here
write_pnr_sdc --file /var/tmp/xtang/openfpga_test_src/SDC

# Write the SDC to run timing analysis for a mapped FPGA fabric
write_analysis_sdc --file /var/tmp/xtang/openfpga_test_src/SDC_analysis

# Finish and exit OpenFPGA
exit
```

8.3 Commands

As OpenFPGA integrates various tools, the commands are categorized into different classes:

8.3.1 Basic Commands

version

Show OpenFPGA version information

help

Show help desk to list all the available commands

exit

Exit OpenFPGA shell

8.3.2 VPR Commands

vpr

OpenFPGA allows users to call `vpr` in the standard way as documented in the vtr_project.

8.3.3 Setup OpenFPGA

read_openfpga_arch

Read the XML file about architecture description (see details in *General Hierarchy*)

`--file` `<string>` or `-f <string>`
: Specify the file name. For example, `--file openfpga_arch.xml`

`--verbose`
: Show verbose log

write_openfpga_arch

Write the OpenFPGA XML architecture file to a file

`--file` `<string>` or `-f <string>`
: Specify the file name. For example, `--file arch_echo.xml`

`--verbose`
: Show verbose log

read_openfpga_simulation_setting

Read the XML file about simulation settings (see details in *Simulation settings*)

--file <string> or -f <string>

Specify the file name. For example, `--file auto_simulation_setting.xml`

--verbose

Show verbose log

write_openfpga_simulation_setting

Write the OpenFPGA XML simulation settings to a file

--file <string> or -f <string>

Specify the file name. For example, `--file auto_simulation_setting_echo.xml`. See details about file format at *Simulation settings*.

--verbose

Show verbose log

read_openfpga_bitstream_setting

Read the XML file about bitstream settings (see details in *Bitstream Setting (.xml)*)

--file <string> or -f <string>

Specify the file name. For example, `--file bitstream_setting.xml`

--verbose

Show verbose log

write_openfpga_bitstream_setting

Write the OpenFPGA XML bitstream settings to a file

--file <string> or -f <string>

Specify the file name. For example, `--file auto_bitstream_setting_echo.xml`. See details about file format at *Bitstream Setting (.xml)*.

--verbose

Show verbose log

link_openfpga_arch

Annotate the OpenFPGA architecture to VPR data base

--activity_file <string>

Specify the signal activity file. For example, `--activity_file counter.act`. This is required when users wants OpenFPGA to automatically find the number of clocks in simulations. See details at *Simulation settings*.

--sort_gsb_chan_node_in_edges

Sort the edges for the routing tracks in General Switch Blocks (GSBs). Strongly recommand to turn this on for uniquifying the routing modules

--verbose

Show verbose log

write_gsb_to_xml

Write the internal structure of General Switch Blocks (GSBs) across a FPGA fabric, including the interconnection between the nodes and node-level details, to XML files

--file <string> or -f <string>

Specify the output directory of the XML files. Each GSB will be written to an indepedent XML file For example, `--file /temp/gsb_output`

--unique

Only output unique GSBs to XML files

--exclude_rr_info

Exclude routing resource graph information from output files, e.g., node id as well as other attributes. This is useful to check the connection inside GSBs purely.

--exclude <string>

Exclude part of the GSB data to be outputted. Can be [sb``|``cbx``|``cby]. Users can exclude multiple parts by using a splitter `,`. For example,

- `--exclude sb`
- `--exclude sb,cbx`

--gsb_names <string>

Specify the name of GSB to be outputted. Users can specify multiple GSBs by using a splitter `,`. When specified, only the GSBs whose names match the list will be outputted to files. If not specified, all the GSBs will be outputted.

Note: When option `--unique` is enable, the given name of GSBs should match the unique modules!

For example,

- `--gsb_names gsb_2__4_,gsb_3__2_`
- `--gsb_names gsb_2__4_`

--verbose

Show verbose log

Note: This command is used to help users to study the difference between GSBs

---

check_netlist_naming_conflict

Check and correct any naming conflicts in the BLIF netlist This is strongly recommended. Otherwise, the outputted Verilog netlists may not be compiled successfully.

> **Warning:** This command may be deprecated in future when it is merged to VPR upstream

--fix

Apply fix-up to the names that violate the syntax

--report <string>

Report the naming fix-up to an XML-based log file. For example, `--report rename.xml`

pb_pin_fixup

Apply fix-up to clustering nets based on routing results This is strongly recommended. Otherwise, the bitstream generation may be wrong

> **Warning:** This command may be deprecated in future when it is merged to VPR upstream

--verbose

Show verbose log

lut_truth_table_fixup

Apply fix-up to Look-Up Table truth tables based on packing results

> **Warning:** This command may be deprecated in future when it is merged to VPR upstream

--verbose

Show verbose log

build_fabric

Build the module graph.

--compress_routing

Enable compression on routing architecture modules. Strongly recommend this as it will minimize the number of routing modules to be outputted. It can reduce the netlist size significantly.

--duplicate_grid_pin

Enable pin duplication on grid modules. This is optional unless ultra-dense layout generation is needed

--load_fabric_key <string>

Load an external fabric key from an XML file. For example, `--load_fabric_key fpga_2x2.xml` See details in *Fabric Key (.xml)*.

--generate_random_fabric_key

Generate a fabric key in a random way

--write_fabric_key <string>.

Output current fabric key to an XML file. For example, `--write_fabric_key fpga_2x2.xml` See details in *Fabric Key (.xml)*.

--frame_view

Create only frame views of the module graph. When enabled, top-level module will not include any nets. This option is made for save runtime and memory.

> **Warning:** Recommend to turn the option on when bitstream generation is the only purpose of the flow. Do not use it when you need generate netlists!

--verbose

Show verbose log

Note: This is a must-run command before launching FPGA-Verilog, FPGA-Bitstream, FPGA-SDC and FPGA-SPICE

---

write_fabric_hierarchy

Write the hierarchy of FPGA fabric graph to a plain-text file

--file <string> or -f <string>

Specify the file name to write the hierarchy.

--depth <int>

Specify at which depth of the fabric module graph should the writer stop outputting. The root module start from depth 0. For example, if you want a two-level hierarchy, you should specify depth as 1.

--verbose

Show verbose log

Note: This file is designed for hierarchical PnR flow, which requires the tree of Multiple-Instanced-Blocks (MIBs).

---

write_fabric_io_info

Write the I/O information of FPGA fabric to an XML file

--file <string> or -f <string>

Specify the file name to write the I/O information

--no_time_stamp

Do not print time stamp in bitstream files

--verbose

Show verbose log

Note: This file is designed for pin constraint file conversion.

---

pcf2place

Convert a Pin Constraint File (.pcf, see details in *Pin Constraints File (.pcf)*) to a placement file)

--pcf <string>

Specify the path to the users' pin constraint file

--blif <string>

Specify the path to the users' post-synthesis netlist

--fpga_io_map <string>

Specify the path to the FPGA I/O location. Achieved by the command *write_fabric_io_info*

--pin_table <string>

Specify the path to the pin table file, which describes the pin mapping between chip I/Os and FPGA I/Os. See details in *Pin Table File (.csv)*

--fpga_fix_pins <string>

Specify the path to the placement file which will be outputted by running this command

--no_time_stamp

Do not print time stamp in bitstream files

--verbose

Show verbose log

8.3.4 FPGA-Bitstream

repack

Repack the netlist to physical pbs Repack is an essential procedure before building a bitstream, which aims to packing each programmable blocks by considering **only** the physical modes. Repack's functionality are in the following aspects:

- It annotates the net mapping results from operating modes (considered by VPR) to the physical modes (considered by OpenFPGA)
- It re-routes all the nets by considering the programmable interconnects in physical modes **only**.

Note: This must be done before bitstream generator and testbench generation. Strongly recommend it is done after all the fix-up have been applied

---

--design_constraints

Apply design constraints from an external file. Normally, repack takes the net mapping from VPR packing and routing results. Alternatively, repack can accept the design constraints, in particular, net remapping, from an XML-based design constraint description. See details in *Repack Design Constraints (.xml)*.

Warning: Design constraints are designed to help repacker to identify which clock net to be mapped to which pin, so that multi-clock benchmarks can be correctly implemented, in the case that VPR may not have sufficient vision on clock net mapping. **Try not to use design constraints to remap any other types of nets!!!**

--ignore_global_nets_on_pins

Specify the mapping results of global nets should be ignored on which pins of a `pb_type`. For example, `--ignore_global_nets_on_pins clb.I[0:11]`. Once specified, the mapping results on the pins for all the global nets, such as clock, reset *etc.*, are ignored. Routing traces will be appeneded to other pins where the same global nets are mapped to.

Note: This option is designed for global nets which are applied to both data path and global networks. For example, a reset signal is mapped to both a LUT input and the reset pin of a FF. Suggest not to use the option in other purposes!

Warning: Users must specify the size/width of the pin. Currently, OpenFPGA cannot infer the pin size from the architecture!!!

--verbose

Show verbose log

build_architecture_bitstream

Decode VPR implementing results to an fabric-independent bitstream database

--read_file <string>

Read the fabric-independent bitstream from an XML file. When this is enabled, bitstream generation will NOT consider VPR results. See details at *Architecture Bitstream (.xml)*.

--write_file <string>

Output the fabric-independent bitstream to an XML file. See details at *Architecture Bitstream (.xml)*.

--no_time_stamp

Do not print time stamp in bitstream files

--verbose

Show verbose log

build_fabric_bitstream

Build a sequence for every configuration bits in the bitstream database for a specific FPGA fabric

--verbose

Show verbose log

write_fabric_bitstream

Output the fabric bitstream database to a specific file format

--file <string> or -f <string>

Output the fabric bitstream to an plain text file (only 0 or 1)

--format <string>

Specify the file format [plain_text | xml]. By default is plain_text. See file formats in *XML (.xml)* and *Plain text (.bit)*.

--fast_configuration

Reduce the bitstream size when outputing by skipping dummy configuration bits. It is applicable to configuration chain, memory bank and frame-based configuration protocols. For configuration chain, when enabled, the zeros at the head of the bitstream will be skipped. For memory bank and frame-based, when enabled, all the zero configuration bits will be skipped. So ensure that your memory cells can be correctly reset to zero with a reset signal.

> **Warning:** Fast configuration is only applicable to plain text file format!

> **Note:** If both reset and set ports are defined in the circuit modeling for programming, OpenFPGA will pick the one that will bring largest benefit in speeding up configuration.

--keep_dont_care_bits

Keep don't care bits (x) in the outputted bitstream file. This is only applicable to plain text file format. If not enabled, the don't care bits are converted to either logic 0 or 1.

--no_time_stamp

Do not print time stamp in bitstream files

--verbose

Show verbose log

write_io_mapping

Output the I/O mapping information to a file

--file <string> or -f <string>

Specify the file name where the I/O mapping will be outputted to. See file formats in *I/O Mapping File (.xml)*.

--no_time_stamp

Do not print time stamp in bitstream files

--verbose

Show verbose log

report_bitstream_distribution

Output the bitstream distribution to a file

--file <string> or -f <string>

Specify the file name where the bitstream distribution will be outputted to. See file formats in *Bitstream Distribution File (.xml)*.

--depth <int> or -d <int>

Specify the maximum depth of the block which should appear in the block

--no_time_stamp

Do not print time stamp in bitstream files

--verbose

Show verbose log

8.3.5 FPGA-Verilog

write_fabric_verilog

Write the Verilog netlist for FPGA fabric based on module graph. See details in *Fabric Netlists*.

--file <string> or -f <string>

Specify the output directory for the Verilog netlists. For example, `--file /temp/fabric_netlist/`

--default_net_type <string>

Specify the default net type for the Verilog netlists. Currently, supported types are `none` and `wire`. Default value: `none`.

--explicit_port_mapping

Use explicit port mapping when writing the Verilog netlists

--include_timing

Output timing information to Verilog netlists for primitive modules

--use_relative_path

Force to use relative path in netlists when including other netlists. By default, this is off, which means that netlists use absolute paths when including other netlists

--print_user_defined_template

Output a template Verilog netlist for all the user-defined `circuit models` in *Circuit Library*. This aims to help engineers to check what is the port sequence required by top-level Verilog netlists

--no_time_stamp

Do not print time stamp in Verilog netlists

--verbose

Show verbose log

write_full_testbench

Write the full testbench for FPGA fabric in Verilog format. See details in *Testbench*.

--file <string> or -f <string>

The output directory for all the testbench netlists. We suggest the use of same output directory as fabric Verilog netlists. For example, `--file /temp/testbench`

--bitstream <string>

The bitstream file to be loaded to the full testbench, which should be in the same file format that OpenFPGA can outputs (See detailes in *Plain text (.bit)*). For example, `--bitstream and2.bit`

--fabric_netlist_file_path <string>

Specify the fabric Verilog file if they are not in the same directory as the testbenches to be generated. If not specified, OpenFPGA will assume that the fabric netlists are the in the same directory as testbenches and assign default names. For example, `--file /temp/fabric/fabric_netlists.v`

--reference_benchmark_file_path <string>

Specify the reference benchmark Verilog file if you want to output any self-checking testbench. For example, `--reference_benchmark_file_path /temp/benchmark/counter_post_synthesis.v`

Note: If not specified, the testbench will not include any self-checking feature!

--pin_constraints_file <string> or -pcf <string>

Specify the *Pin Constraints File* (PCF) if you want to custom stimulus in testbenches. For example, `-pin_constraints_file pin_constraints.xml` Strongly recommend for multi-clock simulations. See detailed file format about *Pin Constraints File (.xml)*.

--bus_group_file <string> or -bgf <string>

Specify the *Bus Group File* (BGF) if you want to group pins to buses. For example, `-bgf bus_group.xml` Strongly recommend when input HDL contains bus ports. See detailed file format about *Bus Group File (.xml)*.

--fast_configuration

Enable fast configuration phase for the top-level testbench in order to reduce runtime of simulations. It is applicable to configuration chain, memory bank and frame-based configuration protocols. For configuration chain, when enabled, the zeros at the head of the bitstream will be skipped. For memory bank and frame-based, when enabled, all the zero configuration bits will be skipped. So ensure that your memory cells can be correctly reset to zero with a reset signal.

Note: If both reset and set ports are defined in the circuit modeling for programming, OpenFPGA will pick the one that will bring largest benefit in speeding up configuration.

--explicit_port_mapping

Use explicit port mapping when writing the Verilog netlists

--default_net_type <string>

Specify the default net type for the Verilog netlists. Currently, supported types are `none` and `wire`. Default value: `none`.

--include_signal_init

Output signal initialization to Verilog testbench to smooth convergence in HDL simulation

Note: We strongly recommend users to turn on this flag as it can help simulators to converge quickly.

Warning: Signal initialization is only applied to the datapath inputs of routing multiplexers (considering the fact that they are indispensible cells of FPGAs)! If your FPGA does not contain any multiplexer cells, signal initialization is not applicable.

--no_time_stamp

Do not print time stamp in Verilog netlists

--use_relative_path

Force to use relative path in netlists when including other netlists. By default, this is off, which means that netlists use absolute paths when including other netlists

--verbose

Show verbose log

write_preconfigured_fabric_wrapper

Write the Verilog wrapper for a preconfigured FPGA fabric. See details in *Testbench*.

--file <string> or -f <string>

The output directory for the netlists. We suggest the use of same output directory as fabric Verilog netlists. For example, `--file /temp/testbench`

--fabric_netlist_file_path <string>

Specify the fabric Verilog file if they are not in the same directory as the testbenches to be generated. If not specified, OpenFPGA will assume that the fabric netlists are the in the same directory as testbenches and assign default names. For example, `--file /temp/fabric/fabric_netlists.v`

--pin_constraints_file <string> or -pcf <string>

Specify the *Pin Constraints File* (PCF) if you want to custom stimulus in testbenches. For example, `-pin_constraints_file pin_constraints.xml` Strongly recommend for multi-clock simulations. See detailed file format about *Pin Constraints File (.xml)*.

--bus_group_file <string> or -bgf <string>

Specify the *Bus Group File* (BGF) if you want to group pins to buses. For example, `-bgf bus_group.xml` Strongly recommend when input HDL contains bus ports. See detailed file format about *Bus Group File (.xml)*.

--explicit_port_mapping

Use explicit port mapping when writing the Verilog netlists

--default_net_type <string>

Specify the default net type for the Verilog netlists. Currently, supported types are `none` and `wire`. Default value: `none`.

--embed_bitstream <string>

Specify if the bitstream should be embedded to the Verilog netlists in HDL codes. Available options are `none`, `iverilog` and `modelsim`. Default value: `modelsim`.

> **Warning:** If the option `none` is selected, bitstream will not be embedded. Users should force the bitstream through HDL simulator commands. Otherwise, functionality of the wrapper netlist is wrong!

> **Warning:** Please specify `iverilog` if you are using icarus iVerilog simulator.

--include_signal_init

Output signal initialization to Verilog testbench to smooth convergence in HDL simulation

> **Note:** We strongly recommend users to turn on this flag as it can help simulators to converge quickly.

> **Warning:** Signal initialization is only applied to the datapath inputs of routing multiplexers (considering the fact that they are indispensible cells of FPGAs)! If your FPGA does not contain any multiplexer cells, signal initialization is not applicable.

--no_time_stamp

Do not print time stamp in Verilog netlists

--verbose

Show verbose log

write_preconfigured_testbench

Write the Verilog testbench for a preconfigured FPGA fabric. See details in *Testbench*.

--file <string> or **-f** <string>

The output directory for all the testbench netlists. We suggest the use of same output directory as fabric Verilog netlists. For example, `--file /temp/testbench`

--fabric_netlist_file_path <string>

Specify the fabric Verilog file if they are not in the same directory as the testbenches to be generated. If not specified, OpenFPGA will assume that the fabric netlists are the in the same directory as testbenches and assign default names. For example, `--file /temp/fabric/fabric_netlists.v`

--reference_benchmark_file_path <string>

Specify the reference benchmark Verilog file if you want to output any self-checking testbench. For example, `--reference_benchmark_file_path /temp/benchmark/counter_post_synthesis.v`

> **Note:** If not specified, the testbench will not include any self-checking feature!

--pin_constraints_file <string> or -pcf <string>

Specify the *Pin Constraints File* (PCF) if you want to custom stimulus in testbenches. For example, -pin_constraints_file pin_constraints.xml Strongly recommend for multi-clock simulations. See detailed file format about *Pin Constraints File (.xml)*.

--bus_group_file <string> or -bgf <string>

Specify the *Bus Group File* (BGF) if you want to group pins to buses. For example, -bgf bus_group.xml Strongly recommend when input HDL contains bus ports. See detailed file format about *Bus Group File (.xml)*.

--explicit_port_mapping

Use explicit port mapping when writing the Verilog netlists

--default_net_type <string>

Specify the default net type for the Verilog netlists. Currently, supported types are none and wire. Default value: none.

--no_time_stamp

Do not print time stamp in Verilog netlists

--use_relative_path

Force to use relative path in netlists when including other netlists. By default, this is off, which means that netlists use absolute paths when including other netlists

--verbose

Show verbose log

write_simulation_task_info

Write an interchangeable file in .ini format to interface HDL simulators, such as iVerilog and Modelsim.

--file <string> or -f <string>

Specify the file path to output simulation-related information. For example, --file simulation.ini

--hdl_dir <string>

Specify the directory path where HDL netlists are created. For example, --hdl_dir ./SRC

--reference_benchmark_file_path <string>

Must specify the reference benchmark Verilog file if you want to output any testbenches. For example, --reference_benchmark_file_path /temp/benchmark/counter_post_synthesis.v

--testbench_type <string>

Specify the type of testbenches [preconfigured_testbench``|``full_testbench]. By default, it is the preconfigured_testbench.

--time_unit <string>

Specify a time unit to be used in SDC files. Acceptable values are string: as | fs | ps | ns | us | ms | ks | Ms. By default, we will consider second (ms).

--verbose

Show verbose log

8.3.6 FPGA-SDC

write_pnr_sdc

Write the SDC files for PnR backend

--file <string> or -f <string>

Specify the output directory for SDC files For example, `--file /temp/pnr_sdc`

--hierarchical

Output SDC files without full path in hierarchy

--flatten_names

Use flatten names (no wildcards) in SDC files

--time_unit <string>

Specify a time unit to be used in SDC files. Acceptable values are string: `as` | `fs` | `ps` | `ns` | `us` | `ms` | `ks` | `Ms`. By default, we will consider second (`s`).

--output_hierarchy

Output hierarchy of Multiple-Instance-Blocks(MIBs) to plain text file. This is applied to constrain timing for grids, Switch Blocks and Connection Blocks.

Note: Valid only when `compress_routing` is enabled in `build_fabric`

--constrain_global_port

Constrain all the global ports of FPGA fabric.

--constrain_non_clock_global_port

Constrain all the non-clock global ports as clocks ports of FPGA fabric

---

Note: `constrain_global_port` will treat these global ports in Clock Tree Synthesis (CTS), in purpose of balancing the delay to each sink. Be carefull to enable `constrain_non_clock_global_port`, this may significanly increase the runtime of CTS as it is supposed to be routed before any other nets. This may cause routing congestion as well.

--constrain_grid

Constrain all the grids of FPGA fabric

--constrain_sb

Constrain all the switch blocks of FPGA fabric

--constrain_cb

Constrain all the connection blocks of FPGA fabric

--constrain_configurable_memory_outputs

Constrain all the outputs of configurable memories of FPGA fabric

--constrain_routing_multiplexer_outputs

Constrain all the outputs of routing multiplexer of FPGA fabric

--constrain_switch_block_outputs

Constrain all the outputs of switch blocks of FPGA fabric

--constrain_zero_delay_paths

Constrain all the zero-delay paths in FPGA fabric

> **Note:** Zero-delay path may cause errors in some PnR tools as it is considered illegal

--verbose

Enable verbose output

write_configuration_chain_sdc

Write the SDC file to constrain the timing for configuration chain. The timing constraints will always start from the first output (Q) of a Configuration Chain Flip-flop (CCFF) and ends at the inputs of the next CCFF in the chain. Note that Qb of CCFF will not be constrained!

--file <string> or -f <string>

Specify the output SDC file. For example, `--file cc_chain.sdc`

--time_unit <string>

Specify a time unit to be used in SDC files. Acceptable values are string: `as` | `fs` | `ps` | `ns` | `us` | `ms` | `ks` | `Ms`. By default, we will consider second (`s`).

--max_delay <float>

Specify the maximum delay to be used. The timing value should follow the time unit defined in this command.

--min_delay <float>

Specify the minimum delay to be used. The timing value should follow the time unit defined in this command.

> **Note:** Only applicable when configuration chain is used as configuration protocol

write_sdc_disable_timing_configure_ports

Write the SDC file to disable timing for configure ports of programmable modules. The SDC aims to break the combinational loops across FPGAs and avoid false path timing to be visible to timing analyzers

--file <string> or -f <string>

Specify the output SDC file. For example, `--file disable_config_timing.sdc`.

--flatten_names

Use flatten names (no wildcards) in SDC files

--verbose

Show verbose log

write_analysis_sdc

Write the SDC to run timing analysis for a mapped FPGA fabric

--file <string> or -f <string>

Specify the output directory for SDC files. For example, `--file counter_sta_analysis.sdc`

--flatten_names

Use flatten names (no wildcards) in SDC files

--time_unit <string>

Specify a time unit to be used in SDC files. Acceptable values are string: `as` | `fs` | `ps` | `ns` | `us` | `ms` | `ks` | `Ms`. By default, we will consider second (`s`).

CHAPTER

NINE

FPGA-SPICE

Warning: FPGA-SPICE has not been integrated to VPR8 version yet. Please the following tool guide is for VPR7 version now

9.1 Command-line Options

All the command line options of FPGA-SPICE can be shown by calling the help menu of VPR. Here are all the FPGA-SPICE-related options that you can find:

FPGA-SPICE Supported Options:

```
--fpga_spice
--fpga_spice_dir <directory_path_output_spice_netlists>
--fpga_spice_print_top_testbench
--fpga_spice_print_lut_testbench
--fpga_spice_print_hardlogic_testbench
--fpga_spice_print_pb_mux_testbench
--fpga_spice_print_cb_mux_testbench
--fpga_spice_print_sb_mux_testbench
--fpga_spice_print_cb_testbench
--fpga_spice_print_sb_testbench
--fpga_spice_print_grid_testbench
--fpga_spice_rename_illegal_port
--fpga_spice_signal_density_weight <float>
--fpga_spice_sim_window_size <float>
--fpga_spice_leakage_only
--fpga_spice_parasitic_net_estimation_off
--fpga_spice_testbench_load_extraction_off
--fpga_spice_sim_mt_num <int>
```

Note: FPGA-SPICE requires the input of activity estimation results (*.act file) from ACE2. Remember to use the option –activity_file <act_file> to read the activity file.

Note: To dump full-chip-level testbenches, the option –-fpga_spice_print_top_testbench should be enabled.

Note: To dump grid-level testbenches, the options – fpga_spice_print_grid_testbench, –

fpga_spice_print_cb_testbench and – fpga_spice_print_sb_testbench should be enabled.

Note: To dump component-level testbenches, the options –fpga_spice_print_lut_testbench, –fpga_spice_print_hardlogic_testbench, –fpga_spice_print_pb_mux_testbench, –fpga_spice_print_cb_mux_testbench and –fpga_spice_print_sb_mux_testbench should be enabled.

Table 9.1: Commmand-line Options of FPGA-SPICE

Command Options	Description
–fpga_spice	Turn on the FPGA-SPICE.
–fpga_spice_dir <dir_path>	Specify the directory that all the SPICE netlists will be outputted to. <dir_path> is the destination directory.
–fpga_spice_print_top_testbench	Print the full-chip-level testbench for the FPGA.
–fpga_spice_print_lut_testbench	Print the testbenches for all the LUTs.
–fpga_spice_print_hardlogic_testbench	Print the test benches for all the hard logic.
–fpga_spice_print_pb_mux_testbench	Print the testbenches for all the multiplexers in the logic blocks.
–fpga_spice_print_cb_mux_testbench	Print the testbenches for all the multiplexers in Connection Boxes.
– fpga_spice_print_sb_mux_testbench	Print the testbenches for all the multiplexers in Switch Blocks.
–fpga_spice_print_cb_testbench	Print the testbenches for all the CBs.
–fpga_spice_print_sb_testbench	Print the testbenches for all the SBs.
–fpga_spice_print_grid_testbench	Print the testbenches for the logic blocks.
–fpga_spice_rename_illegal_port	Rename illegal port names
–fpga_spice_signal_density_weight <float>	Set the weight of signal density.
–fpga_spice_sim_window_size <float>	Set the window size in determining the number of clock cycles in simulation.
–fpga_spice_leakage_only	FPGA-SPICE conduct power analysis on the leakage power only.
–fpga_spice_parasitic_net_estimation_off	Turn off the parasitic net estimation technique.
–fpga_spice_testbench_load_extraction_off	Turn off the load effect on net estimation technique.
–fpga_spice_sim_mt_num <int>	Set the number of multi-thread used in simulation

Note: The parasitic net estimation technique is used to analyze the parasitic net activities which improve the accuracy of power analysis. When turned off, the errors between the full-chip-level and grid/component-level testbenches will increase."

9.2 Hierarchy of SPICE Output Files

All the generated SPICE netlists are located in the <spice_dir> as you specify in the command-line options. Under the <spice_dir>, FPGA-SPICE creates a number of folders: include, subckt, lut_tb, dff_tb, grid_tb, pb_mux_tb, cb_mux_tb, sb_mux_tb, top_tb, results. Under the <spice_dir>, FPGA-SPICE also creates a shell script called run_hspice_sim.sh, which run all the simulations for all the testbenches. The folders contain the sub-circuits and testbenches, and their contents are shown as follows.

Table 9.2: Folder hierarchy of FPGA-SPICE

Folder	Content
includes	The header files which contain the parameters for stimuli and measurement, as defined in <tech_lib>.
subckt	Contain all the auto-generated sub-circuits, such as inverters, buffers, transmission gates, multiplexers, LUTs, and even logic blocks, connection boxes, and switch blocks.
lut_tb	Contain all the testbenches for LUTs. This folder is created only when option print_spice_lut_testbench is enabled.
dff_tb	Contain all the testbenches for FFs. This folder is created only when option print_spice_dff_testbench is enabled.
grid_tb	Contain all the testbenches for logic blocks (grid-level testbenches). This folder is created only when option print_spice_grid_testbench is enabled.
pb_mux_tb	Contain the testbenches for the multiplexers inside logic blocks. This folder is created only when option print_spice_pbmux_test-bench is enabled.
cb_mux_tb	Contain all the testbenches for the multiplexers inside connection boxes. This folder is created only when option print_spice_cbmux_testbench is enabled.
sb_mux_tb	Contain all the testbenches for the multiplexers inside switch blocks. This folder is created only when option print_spice_sbmux_test-bench is enabled.
top_tb	Contain the full-chip-level testbench. This folder is created only when option print_spice_top_testbench is enabled.
results	An empty folder when created. It stores all the simulation results by running the shell script run_hspice_sim.sh.

9.3 Run SPICE simulation

- Simulation results

The HSPICE simulator creates an LIS file (*.lis) to store the results. In each LIS file, you can find the leakage power and dynamic power of each module, as well the total leakage power and the total dynamic power of all the modules in a SPICE netlist.

The following is an example of simulation results of a pb_mux testbench.:

```
total_leakage_srams= -16.4425u

total_dynamic_srams=  83.0480u

total_energy_per_cycle_srams= 269.7773f

total_leakage_power_mux[0to76]=-140.1750u

total_energy_per_cycle_mux[0to76]= -37.5871p

total_leakage_power_pb_mux=-140.1750u

total_energy_per_cycle_pb_mux= -37.5871p
```

Note: total_energy_per_cycle_srams represents the total energy per cycle of all the SRAMs of the multiplexers in this testbench, while total_energy_per_cycle_pb_mux is the total energy per cycle of all the multiplexer structures in this

testbench.

Therefore, the total energy per cycle of all the multiplexers in this testbench should be the sum of total_energy_per_cycle_srams and total_energy_per_cycle_pb_mux.

Similarly, the total leakage power of all the multiplexers in this testbench should be the sum of total_leakage_srams and total_leakage_power_pb_mux.

The leakage power is measured for the first clock cycle, where FPGA-SPICE set all the voltage stimuli in constant voltage levels.

The total energy per cycle is measured for the rest of clock cycles (the 1st clock cycle is not included).

The total power can be calculated by,

$total\_energy\_per\_cycle \cdot clock\_freq$

where clock_freq is the clock frequency used in SPICE simulations.

9.4 Create Customized SPICE Modules

To make sure the customized SPICE netlists can be correctly included in FPGA-SPICE, the following rules should be fully respected:

1. The customized SPICE netlists could contain multiple sub-circuits but the names of these sub-circuits should not be conflicted with any reserved words.. Here is an example of defining a sub-circuit in SPICE netlists. The <subckt_name> should be a unique one, which should not be conflicted with any reserved words. .subckt <subckt_name> <ports>

2. The ports of sub-circuit to be included should strictly follow the sequence: <input_ports> <output_ports> <sram_ports> <clock_ports> <vdd> <gnd> It is not necessary to keep the names of ports be the same with what is defined in the SPICE models. But the bandwidth of the ports should be consistent with what is defined in the Circuit models.

Note: If the customized SPICE netlists include inverters, buffers or transmission gates, it is recommended to use those auto-generated by FPGA-SPICE. It is also recommended to use the transistor sub-circuit (vpr_nmos and vpr_pmos) auto-generated by FPGA-SPICE. In the appendix, we introduce how to use these useful sub-circuits.

CHAPTER

TEN

FPGA-VERILOG

10.1 Fabric Netlists

In this part, we will introduce the hierarchy, dependency and functionality of each Verilog netlist, which are generated to model the FPGA fabric.

Note: These netlists are automatically generated by the OpenFPGA command `write_fabric_verilog`. See *FPGA-Verilog* for its detailed usage.

All the generated Verilog netlists are located in the directory as you specify in the OpenFPGA command `write_fabric_verilog`. Inside the directory, the Verilog netlists are organized as illustrated in Fig. 10.1.

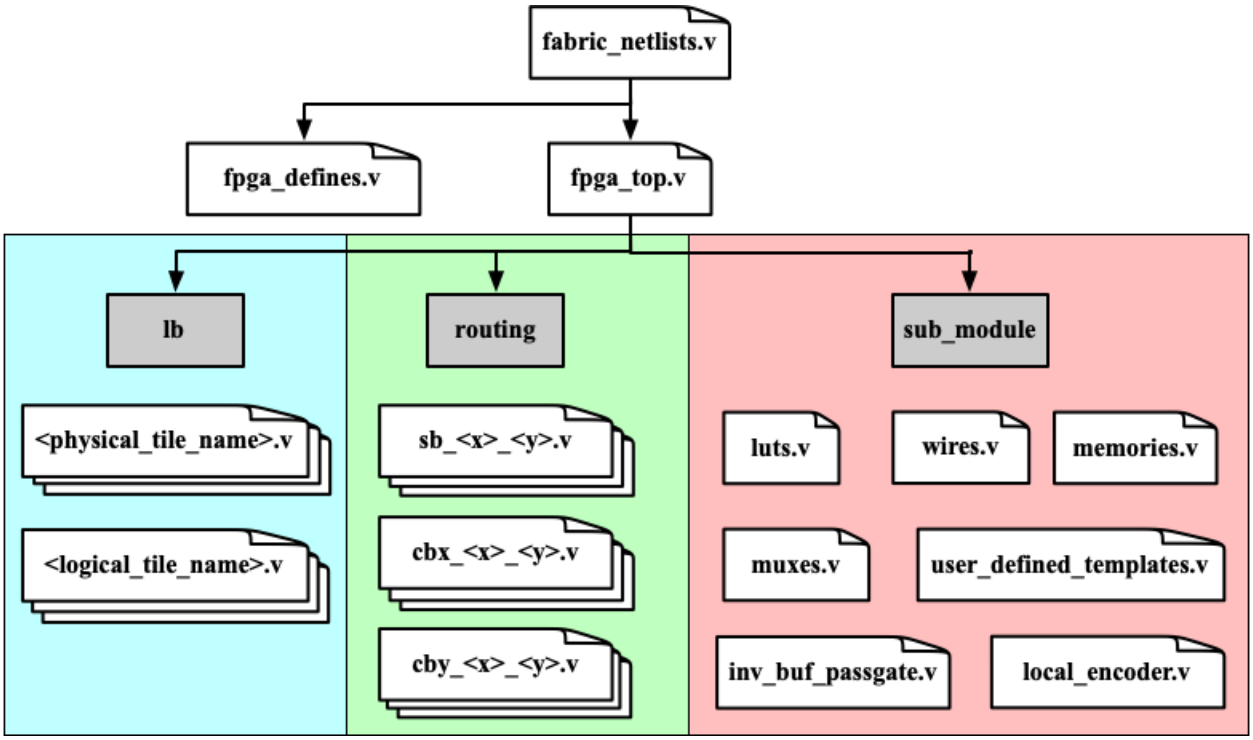


Fig. 10.1: Hierarchy of Verilog netlists modeling a FPGA fabric

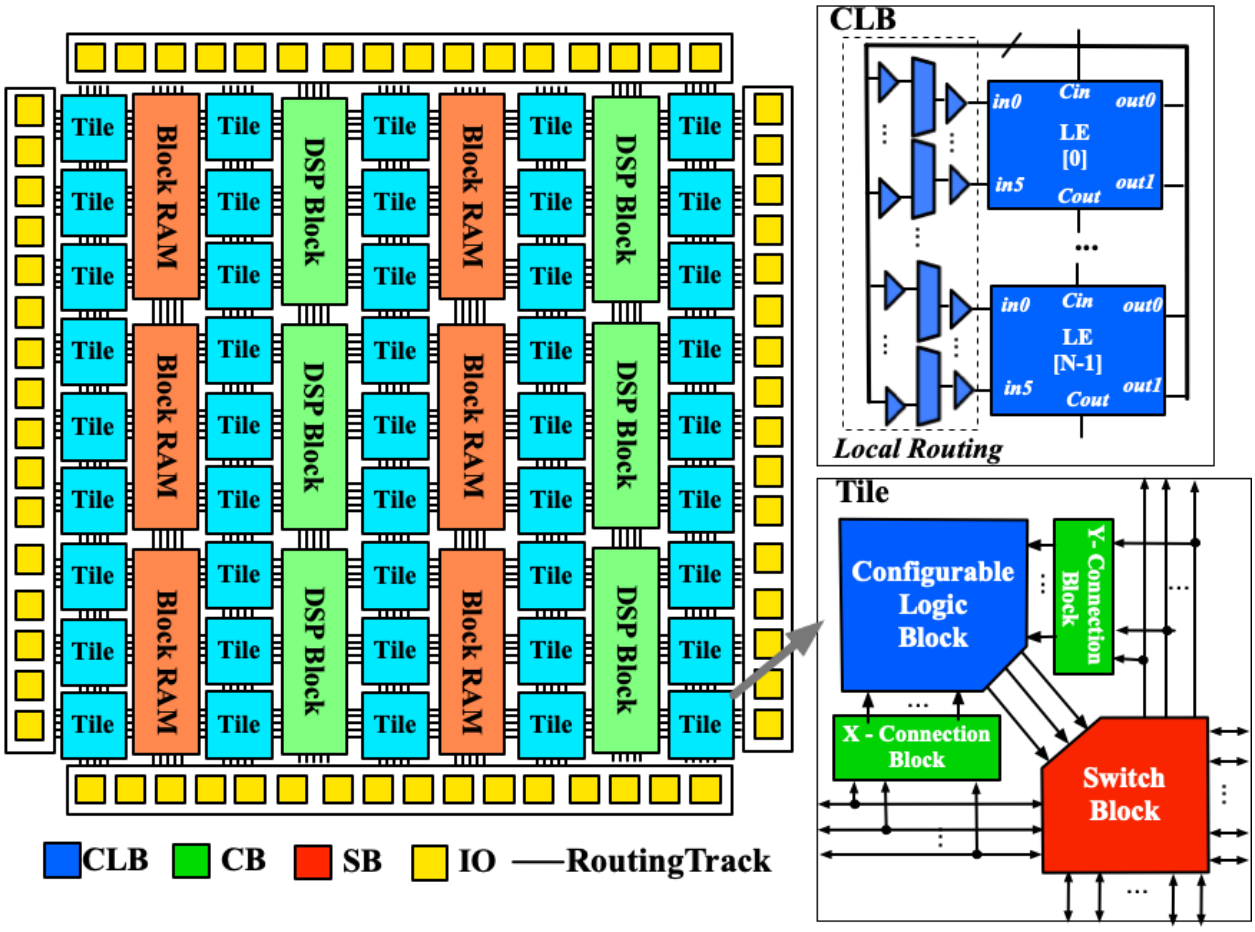


Fig. 10.2: An illustrative FPGA fabric modelled by the Verilog netlists

10.1.1 Top-level Netlists

fabric_netlists.v

This file includes all the related Verilog netlists that are used by the `fpga_top.v`. This file is created to simplify the netlist addition for HDL simulator and backend tools. This is the only file you need to add to a simulator or backend project.

Note: User-defined (external) Verilog netlists are included in this file.

---

fpga_top.v

This netlist contains the top-level module of the fpga fabric, corresponding to the fabric shown in Fig. 10.2.

fpga_defines.v

This file includes pre-processing flags required by the `fpga_top.v`, to smooth HDL simulation. It will include the folliwng pre-procesing flags:

- `` `define ENABLE_TIMING `` When enabled, all the delay values defined in primitive Verilog modules will be considered in compilation. This flag is added when `--include_timing` option is enabled when calling the `write_fabric_verilog` command.

Note: We strongly recommend users to turn on this flag as it can help simulators to converge quickly.

10.1.2 Logic Blocks

This sub-directory contains all the Verilog modules modeling configurable logic blocks, heterogeneous blocks as well as I/O blocks. Take the example in Fig. 10.2, the modules are CLBs, DSP blocks, I/Os and Block RAMs.

<physical_tile_name>.v

For each <physical_tile> defined in the VPR architecture description, a Verilog netlist will be generated to model its internal structure.

Note: For I/O blocks, separated <physical_tile_name>.v will be generated for each side of a FPGA fabric.

<logical_tile_name>.v

For each root `pb_type` defined in the <complexblock> of VPR architecture description, a Verilog netlist will be generated to model its internal structure.

10.1.3 Routing Blocks

This sub-directory contains all the Verilog modules modeling Switch Blocks (SBs) and Connection Blocks (CBs). Take the example in Fig. 10.2, the modules are the Switch Blocks, X- and Y- Connection Blocks of a tile.

sb_<x>_<y>.v

For each unique Switch Block (SB) created by VPR routing resource graph generator, a Verilog netlist will be generated. The <x> and <y> denote the coordinate of the Switch Block in the FPGA fabric.

cbx_<x>_<y>.v

For each unique X-direction Connection Block (CBX) created by VPR routing resource graph generator, a Verilog netlist will be generated. The <x> and <y> denote the coordinate of the Connection Block in the FPGA fabric.

cby_<x>_<y>.v

For each unique Y-direction Connection Block (CBY) created by VPR routing resource graph generator, a Verilog netlist will be generated. The <x> and <y> denote the coordinate of the Connection Block in the FPGA fabric.

10.1.4 Primitive Modules

This sub-directory contains all the primitive Verilog modules, which are used to build the logic blocks and routing blocks.

luts.v

Verilog modules for all the Look-Up Tables (LUTs), which are defined as `<circuit_model name="lut">` of OpenFPGA architecture description. See details in *Circuit Library*.

wires.v

Verilog modules for all the routing wires, which are defined as `<circuit_model name="wire|chan_wire">` of OpenFPGA architecture description. See details in *Circuit Library*.

memories.v

Verilog modules for all the configurable memories, which are defined as `<circuit_model name="ccff|sram">` of OpenFPGA architecture description. See details in *Circuit Library*.

muxes.v

Verilog modules for all the routing multiplexers, which are defined as `<circuit_model name="mux">` of OpenFPGA architecture description. See details in *Circuit Library*.

Note: multiplexers used in Look-Up Tables are also defined in this netlist.

---

inv_buf_passgate.v

Verilog modules for all the inverters, buffers and pass-gate logics, which are defined as `<circuit_model name="inv_buf|pass_gate">` of OpenFPGA architecture description. See details in *Circuit Library*.

local_encoder.v

Verilog modules for all the encoders and decoders, which are created when routing multiplexers are defined to include local encoders. See details in *Circuit model examples*.

user_defined_templates.v

This is a template netlist, which users can refer to when writing up their user-defined Verilog modules. The user-defined Verilog modules are those `<circuit_model>` in the OpenFPGA architecture description with a specific `verilog_netlist` path. It contains Verilog modules with ports declaration (compatible to other netlists that are auto-generated by OpenFPGA) but without any functionality. This file is created only when the option `--print_user_defined_template` is enabled when calling the `write_fabric_verilog` command.

Warning: Do not include this netlist in simulation without any modification to its content!

10.2 Testbench

In this part, we will introduce the hierarchy, dependency and functionality of each Verilog testbench, which are generated to verify a FPGA fabric implemented with an application.

Testbench Type	Runtime	Test Vector	Test Coverage
Full	Long	Random Stimuli	Full fabric
Formal-oriented	Short	Random Stimuli Formal Method	Programmable fabric only

OpenFPGA can auto-generate two types of Verilog testbenches to validate the correctness of the fabric: full and formal-oriented. Both testbenches share the same organization, as depicted in Fig. 10.3. To enable self-testing, the FPGA and user's RTL design (simulate using an HDL simulator) are driven by the same input stimuli, and any mismatch on their outputs will raise an error flag.

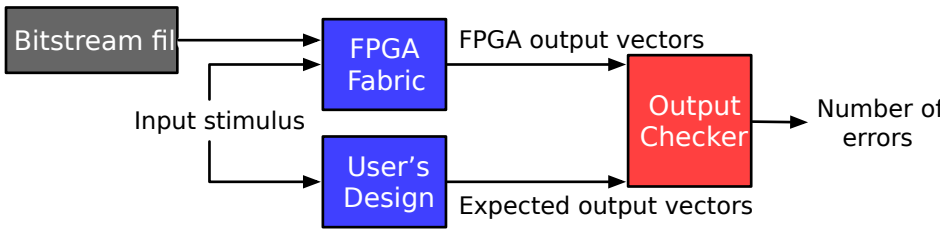


Fig. 10.3: Principles of Verilog testbenches: (1) using common input stimuli; (2) applying bitstream; (3) checking output vectors.

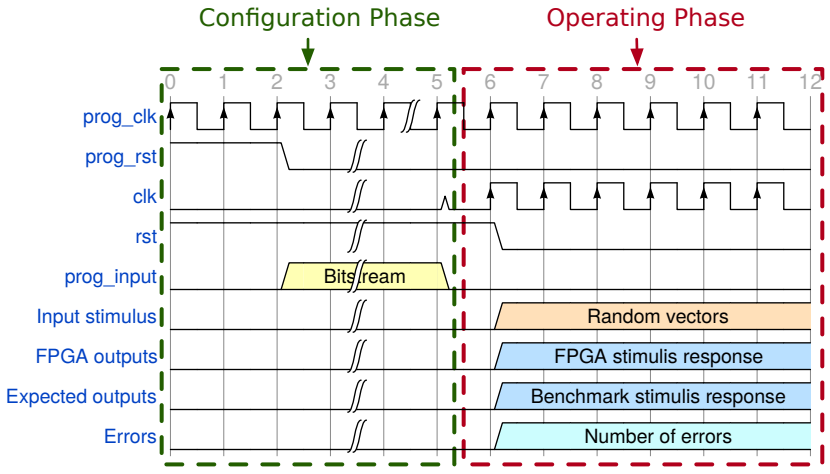


Fig. 10.4: Illustration on the waveforms in full testbench

10.2.1 Full Testbench

Full testbench aims at simulating an entire FPGA operating period, consisting of two phases:

- the **Configuration Phase**, where the synthesized design bitstream is loaded to the programmable fabric, as highlighted by the green rectangle of Fig. 10.4;
- the **Operating Phase**, where random input vectors are auto-generated to drive both Devices Under Test (DUTs), as highlighted by the red rectangle of Fig. 10.4. Using the full testbench, users can validate both the configuration circuits and programming fabric of an FPGA.

10.2.2 Formal-oriented Testbench

The formal-oriented testbench aims to test a programmed FPGA is instantiated with the user's bitstream. The module of the programmed FPGA is encapsulated with the same port mapping as the user's RTL design and thus can be fed to a formal tool for a 100% coverage formal verification. Compared to the full testbench, this skips the time-consuming configuration phase, reducing the simulation time, potentially also significantly accelerating the functional verification, especially for large FPGAs.

Warning: Formal-oriented testbenches do not validate the configuration protocol of FPGAs. It is used to validate FPGA with a wide range of benchmarks.

10.2.3 General Usage

All the generated Verilog testbenches are located in the directory as you specify in the OpenFPGA command `write_fabric_verilog`. Inside the directory, the Verilog testbenches are organized as illustrated in Fig. 10.5.

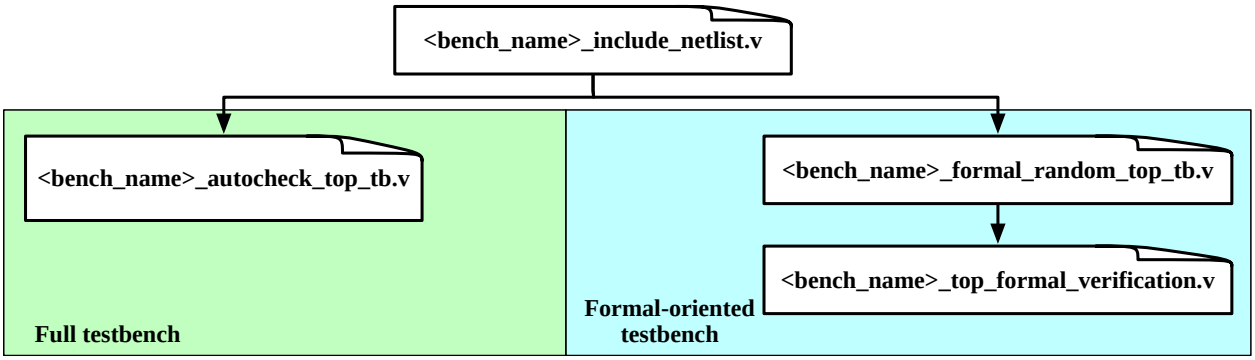


Fig. 10.5: Hierarchy of Verilog testbenches for a FPGA fabric implemented with an application

Note: <bench_name> is the module name of users' RTL design.

<bench_name>_include_netlist.v

This file includes all the related Verilog netlists that are used by the testbenches, including both full and formal oriented testbenches. This file is created to simplify the netlist addition for HDL simulator. This is the only file you need to add to a simulator.

Note: Fabric Verilog netlists are included in this file.

<bench_name>_autocheck_top_tb.v

This is the netlist for full testbench.

<bench_name>_formal_random_top_tb.v

This is the netlist for formal-oriented testbench.

<bench_name>_top_formal_verification.v

This netlist includes a Verilog module of a pre-configured FPGA fabric, which is a wrapper on top of the `fpga_top.v` netlist. The wrapper module has the same port map as the top-level module of user's RTL design, which be directly def to formal verification tools to validate FPGA's functional equivalence. Fig. 10.6 illustrates the organization of a pre-configured module, which consists of a FPGA fabric (see *Fabric Netlists*) and a hard-coded bitstream. Only used I/Os of FPGA fabric will appear in the port list of the pre-configured module.

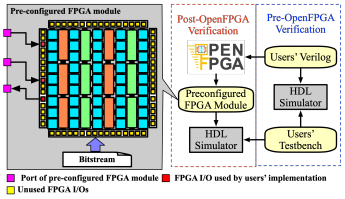


Fig. 10.6: Internal structure of a pre-configured FPGA module

CHAPTER

ELEVEN

FPGA-BITSTREAM

FPGA-Bitstream can generate two types of bitstreams:

11.1 Generic Bitstream

11.1.1 Usage

Generic bitstream is a fabric-independent bitstream where configuration bits are organized out-of-order in a database. This can be regarded as a raw bitstream used for

- `debugging`: Hardware engineers can validate if their configuration memories across the FPGA fabric are assigned to expected values
- `an exchangeable file format for bitstream assembler`: Software engineers can use the raw bitstream to build a bitstream assembler which organize the bitstream in the loadable formate to FPGA chips.
- `creation of artificial bitstream`: Test engineers can craft artificial bitstreams to test each element of the FPGA fabric, which is typically not synthesizable by VPR. Use the `--read_file` option to load the artifical bitsteam to OpenFPGA (see details in *FPGA-Bitstream*).

Warning: The fabric-independent bitstream cannot be directly loaded to FPGA fabrics

11.1.2 File Format

See details in *Architecture Bitstream (.xml)*

11.2 Fabric-dependent Bitstream

11.2.1 Usage

Fabric-dependent bitstream is design to be loadable to the configuration protocols of FPGAs. The bitstream just sets an order to the configuration bits in the database, without duplicating the database. OpenFPGA framework provides a fabric-dependent bitstream generator which is aligned to our Verilog netlists. The fabric-dependent bitstream can be found in the pre-configured Verilog testbenches. The fabric bitsteam can be outputted in different file format in terms of usage.

11.2.2 Plain Text File Format

See details in *Plain text (.bit)*

11.2.3 XML File Format

See details in *XML (.xml)*

CHAPTER

TWELVE

FILE FORMATS

OpenFPGA widely uses XML format for interchangable files

12.1 Pin Constraints File (.xml)

The *Pin Constraints File* (PCF) aims to create pin binding between an implementation and an FPGA fabric. It is a common file format used by FPGA vendors, for example, **`QuickLogic<https://docs.verilogtorouting.org/en/latest/vpr/file_formats/#placement-file-format-place>`_**.

An example of design constraints is shown as follows.

```
<pin_constraints>
  <set_io pin="clk[0]" net="clk0" default_value="1"/>
  <set_io pin="clk[1]" net="clk1"/>
  <set_io pin="clk[2]" net="OPEN"/>
  <set_io pin="clk[3]" net="OPEN"/>
</pin_constraints>
```

pin="<string>"

The pin name of the FPGA fabric to be constrained, which should be a valid pin defined in OpenFPGA architecture description. Explicit index is required, e.g., `clk[1:1]`. Otherwise, default index `0` will be considered, e.g., `clk` will be translated as `clk[0:0]`.

net="<string>"

The net name of the pin to be mapped, which should be consistent with net definition in your `.blif` file. The reserved word `OPEN` means that no net should be mapped to a given pin. Please ensure that it is not conflicted with any net names in your `.blif` file.

default_value="<string>"

The default value of a net to be constrained. This is mainly used when generating testbenches. Valid value is `0` or `1`. If defined as `1`, the net is be driven by the inversion of its stimuli.

Note: This feature is mainly used to generate the correct stimuli for some pin whose polarity can be configurable. For example, the `Reset` pin of an FPGA fabric may be active-low or active-high depending on its configuration.

Note: The default value in pin constraint file has a higher priority than the `default_value` syntax in the *Circuit Library*.

12.2 Repack Design Constraints (.xml)

Warning: For the best practice, current repack design constraints only support the net remapping between pins in the same port. Pin constraints are **NOT** allowed for two separated ports.

- A legal pin constraint example: when there are two clock nets, `clk0` and `clk1`, pin constraints are forced on two pins in a clock port `clk[0:2]` (e.g., `clk[0] = clk0` and `clk[1] == clk1`).
- An **illegal** pin constraint example: when there are two clock nets, `clk0` and `clk1`, pin constraints are forced on two clock ports `clkA[0]` and `clkB[0]` (e.g., `clkA[0] = clk0` and `clkB[0] == clk1`).

An example of design constraints is shown as follows.

```
<repack_design_constraints>
  <pin_constraint pb_type="clb" pin="clk[0]" net="clk0"/>
  <pin_constraint pb_type="clb" pin="clk[1]" net="clk1"/>
  <pin_constraint pb_type="clb" pin="clk[2]" net="OPEN"/>
  <pin_constraint pb_type="clb" pin="clk[3]" net="OPEN"/>
</repack_design_constraints>
```

pb_type="<string>"

The pb_type name to be constrained, which should be consistent with VPR's architecture description.

pin="<string>"

The pin name of the `pb_type` to be constrained, which should be consistent with VPR's architecture description.

net="<string>"

The net name of the pin to be mapped, which should be consistent with net definition in your `.blif` file. The reserved word `OPEN` means that no net should be mapped to a given pin. Please ensure that it is not conflicted with any net names in your `.blif` file.

Warning: Design constraints is a feature for power-users. It may cause repack to fail. It is users's responsibility to ensure proper design constraints

12.3 Architecture Bitstream (.xml)

OpenFPGA can output the generic bitstream to an XML format, which is easy to debug. As shown in the following XML code, configuration bits are organized block by block, where each block could be a LUT, a routing multiplexer *etc*. Each `bitstream_block` includes the following information:

- `name` represents the instance name which you can find in the fabric netlists
- `hierarchy_level` represents the depth of this block in the hierarchy of the FPGA fabric. It always starts from 0 as the root.
- `hierarchy` represents the location of this block in FPGA fabric. The hierachy includes the full hierarchy of this block
  - `instance` denotes the instance name which you can find in the fabric netlists
  - `level` denotes the depth of the block in the hierarchy

- `input_nets` represents the path ids and net names that are mapped to the inputs of block. Unused inputs will be tagged as `unmapped` which is a reserved word of OpenFPGA. Path id corresponds the selected `path_id` in the <bitstream> node.
- `output_nets` represents the path ids and net names that are mapped to the outputs of block. Unused outputs will be tagged as `unmapped` which is a reserved word OpenFPGA.
- `bitstream` represents the configuration bits affiliated to this block.
  - `path_id` denotes the index of inputs which is propagated to the output. Note that smallest valid index starts from zero. Only routing multiplexers have the path index. Unused routing multiplexer will not have a `path_id` of `-1`, which allows bitstream assembler to freely find the best path in terms of Quality of Results (QoR). A used routing multiplexer should have a zero or positive `path_id`.
  - `bit` denotes a single configuration bit under this block. It contains
    - `memory_port` the memory port name which you can find in the fabric netlists by following the hierarchy.
    - `value` a binary value which is the configuration bit assigned to the memory port.

```
<bitstream_block name="fpga_top" hierarchy_level="0">
  <!-- Bitstream block of a 4-input Look-Up Table in a Configurable Logic Block (CLB) -->
  <bitstream_block name="grid_clb_1_1" hierarchy_level="1">
    <bitstream_block name="logical_tile_clb_mode_clb__0" hierarchy_level="2">
      <bitstream_block name="logical_tile_clb_mode_default__fle_0" hierarchy_level="3">
        <bitstream_block name="logical_tile_clb_mode_default__fle_mode_n1_lut4__ble4_0"␣
↪hierarchy_level="4">
          <bitstream_block name="logical_tile_clb_mode_default__fle_mode_n1_lut4__ble4_
↪mode_default__lut4_0" hierarchy_level="5">
            <bitstream_block name="lut4_config_latch_mem" hierarchy_level="6">
              <hierarchy>
                <instance level="0" name="fpga_top"/>
                <instance level="1" name="grid_clb_1_1"/>
                <instance level="2" name="logical_tile_clb_mode_clb__0"/>
                <instance level="3" name="logical_tile_clb_mode_default__fle_0"/>
                <instance level="4" name="logical_tile_clb_mode_default__fle_mode_n1_
↪lut4__ble4_0"/>
                <instance level="5" name="logical_tile_clb_mode_default__fle_mode_n1_
↪lut4__ble4_mode_default__lut4_0"/>
                <instance level="6" name="lut4_config_latch_mem"/>
              </hierarchy>
              <bitstream>
                <bit memory_port="mem_out[0]" value="0"/>
                <bit memory_port="mem_out[1]" value="0"/>
                <bit memory_port="mem_out[2]" value="0"/>
                <bit memory_port="mem_out[3]" value="0"/>
                <bit memory_port="mem_out[4]" value="0"/>
                <bit memory_port="mem_out[5]" value="0"/>
                <bit memory_port="mem_out[6]" value="0"/>
                <bit memory_port="mem_out[7]" value="0"/>
                <bit memory_port="mem_out[8]" value="0"/>
                <bit memory_port="mem_out[9]" value="0"/>
                <bit memory_port="mem_out[10]" value="0"/>
                <bit memory_port="mem_out[11]" value="0"/>
                <bit memory_port="mem_out[12]" value="0"/>
```

(continues on next page)

(continued from previous page)

```
                <bit memory_port="mem_out[13]" value="0"/>
                <bit memory_port="mem_out[14]" value="0"/>
                <bit memory_port="mem_out[15]" value="0"/>
              </bitstream>
            </bitstream_block>
          </bitstream_block>
        </bitstream_block>
      </bitstream_block>
    </bitstream_block>
  </bitstream_block>

  <!-- More bitstream blocks -->

  <!-- Bitstream block of a 2-input routing multiplexer in a Switch Block (SB) -->
  <bitstream_block name="sb_0__2_" hierarchy_level="1">
    <bitstream_block name="mem_right_track_0" hierarchy_level="2">
      <hierarchy>
        <instance level="0" name="fpga_top"/>
        <instance level="1" name="sb_0__2_"/>
        <instance level="2" name="mem_right_track_0"/>
      </hierarchy>
      <input_nets>
        <path id="0" net_name="unmapped"/>
        <path id="1" net_name="unmapped"/>
      </input_nets>
      <output_nets>
        <path id="0" net_name="unmapped"/>
      </output_nets>
      <bitstream path_id="-1">
        <bit memory_port="mem_out[0]" value="0"/>
        <bit memory_port="mem_out[1]" value="0"/>
      </bitstream>
    </bitstream_block>
  </bitstream_block>
</bitstream_block>
```

12.4 Fabric-dependent Bitstream

12.4.1 Plain text (.bit)

This file format is designed to be directly loaded to an FPGA fabric. It does not include any comments but only bitstream.

The information depends on the type of configuration procotol.

vanilla

A line consisting of `0` | `1`

scan_chain

Multiple lines consisting of `0` | `1`

For example, a bitstream for 1 configuration regions:

```
0
1
0
0
```

For example, a bitstream for 4 configuration regions:

```
0000
1010
0110
0120
```

Note: When there are multiple configuration regions, each line may consist of multiple bits. For example, `0110` represents the bits for 4 configuration regions, where the 4 digits correspond to the bits from region `0, 1, 2, 3` respectively.

memory_bank

Multiple lines will be included, each of which is organized as <bl_address><wl_address><bits>. The size of address line and data input bits are shown as a comment in the bitstream file, which eases the development of bitstream downloader. For example

```
// Bitstream width (LSB -> MSB): <bl_address 5 bits><wl_address 5 bits><data input␣
↪1 bits>
```

The first part represents the Bit-Line address. The second part represents the Word-Line address. The third part represents the configuration bit. For example

```
<bitline_address><wordline_address><bit_value>
<bitline_address><wordline_address><bit_value>
...
<bitline_address><wordline_address><bit_value>
```

Note: When there are multiple configuration regions, each <bit_value> may consist of multiple bits. For example, `0110` represents the bits for 4 configuration regions, where the 4 digits correspond to the bits from region `0, 1, 2, 3` respectively.

ql_memory_bank using decoders

Multiple lines will be included, each of which is organized as <bl_address><wl_address><bits>. The size of address line and data input bits are shown as a comment in the bitstream file, which eases the development of bitstream downloader. For example

```
// Bitstream width (LSB -> MSB): <bl_address 5 bits><wl_address 5 bits><data input␣
↪1 bits>
```

The first part represents the Bit-Line address. The second part represents the Word-Line address. The third part represents the configuration bit. For example

```
<bitline_address><wordline_address><bit_value>
<bitline_address><wordline_address><bit_value>
...
<bitline_address><wordline_address><bit_value>
```

Note: When there are multiple configuration regions, each <bit_value> may consist of multiple bits. For example, `0110` represents the bits for 4 configuration regions, where the 4 digits correspond to the bits from region `0, 1, 2, 3` respectively.

ql_memory_bank using flatten BL and WLs

Multiple lines will be included, each of which is organized as <bl_data><wl_data>. The size of data are shown as a comment in the bitstream file, which eases the development of bitstream downloader. For example

```
// Bitstream width (LSB -> MSB): <Region 1: bl_data 5 bits><Region 2: bl_data 4␣
↪bits><Region 1: wl_data 5 bits><Region 2: wl_data 6 bits>
```

The first part represents the Bit-Line data from multiple configuration regions. The second part represents the Word-Line data from multiple configuration regions. For example

```
<bitline_data_region1><bitline_data_region2><wordline_data_region1><wordline_data_
↪region2>
<bitline_data_region1><bitline_data_region2><wordline_data_region1><wordline_data_
↪region2>
...
<bitline_data_region1><bitline_data_region2><wordline_data_region1><wordline_data_
↪region2>
```

Note: The WL data of region is one-hot.

ql_memory_bank using shift registers

Multiple lines will be included, each of which is organized as <bl_data> or <wl_data>. The size of data are shown as a comment in the bitstream file, which eases the development of bitstream downloader. For example

```
// Bitstream word count: 36
// Bitstream bl word size: 39
// Bitstream wl word size: 37
// Bitstream width (LSB -> MSB): <bl shift register heads 1 bits><wl shift register␣
↪heads 1 bits>
```

The bitstream data are organized by words. Each word consists of two parts, BL data to be loaded to BL shift register chains and WL data to be loaded to WL shift register chains For example

```
// Word 0
// BL Part
<bitline_shift_register_data@clock_0>  ----
<bitline_shift_register_data@clock_1>   ^
<bitline_shift_register_data@clock_1>   |
...                                    BL word size
<bitline_shift_register_data@clock_n-2> |
<bitline_shift_register_data@clock_n-1> v
<bitline_shift_register_data@clock_n>  ----
// Word 0
// WL Part
<wordline_shift_register_data@clock_0>  ----
<wordline_shift_register_data@clock_1>   ^
<wordline_shift_register_data@clock_1>   |
```

(continues on next page)

(continued from previous page)

```
...                                    WL word size
<wordline_shift_register_data@clock_n-2> |
<wordline_shift_register_data@clock_n-1> v
<wordline_shift_register_data@clock_n>  ----
// Word 1
// BL Part
<bitline_shift_register_data@clock_0>  ----
<bitline_shift_register_data@clock_1>   ^
<bitline_shift_register_data@clock_1>   |
...                                    BL word size
<bitline_shift_register_data@clock_n-2> |
<bitline_shift_register_data@clock_n-1> v
<bitline_shift_register_data@clock_n>  ----
// Word 1
// WL Part
<wordline_shift_register_data@clock_0>  ----
<wordline_shift_register_data@clock_1>   ^
<wordline_shift_register_data@clock_1>   |
...                                    WL word size
<wordline_shift_register_data@clock_n-2> |
<wordline_shift_register_data@clock_n-1> v
<wordline_shift_register_data@clock_n>  ----
... // More words
```

Note: The BL/WL data may be multi-bit, while each bit corresponds to a configuration region

Note: The WL data of region is one-hot.

frame_based

Multiple lines will be included, each of which is organized as <address><data_input_bits>. The size of address line and data input bits are shown as a comment in the bitstream file, which eases the development of bitstream downloader. For example

```
// Bitstream width (LSB -> MSB): <address 14 bits><data input 1 bits>
```

Note that the address may include don't care bit which is denoted as **x**.

Note: OpenFPGA automatically convert don't care bit to logic **0** when generating testbenches.

For example

```
<frame_address><bit_value>
<frame_address><bit_value>
...
<frame_address><bit_value>
```

Note: When there are multiple configuration regions, each <bit_value> may consist of multiple bits. For example, **0110** represents the bits for 4 configuration regions, where the 4 digits correspond to the bits from

region `0, 1, 2, 3` respectively.

12.4.2 XML (.xml)

This file format is designed to generate testbenches using external tools, e.g., CocoTB.

In principle, the file consist a number of XML node <region>, each region has a unique id, and contains a number of XML nodes <bit>.

- `id`: The unique id of a configuration region in the fabric bitstream.

A quick example:

```
<region id="0">
  <bit id="0" value="1" path="fpga_top.grid_clb_1__2_.logical_tile_clb_mode_clb__0.mem_
↪fle_9_in_5.mem_out[0]"/>
  </bit>
</region>
```

Each XML node <bit> contains the following attributes:

- `id`: The unique id of the configuration bit in the fabric bitstream.
- `value`: The configuration bit value.
- `path` represents the location of this block in FPGA fabric, i.e., the full path in the hierarchy of FPGA fabric.

A quick example:

```
<bit id="0" value="1" path="fpga_top.grid_clb_1__2_.logical_tile_clb_mode_clb__0.mem_fle_
↪9_in_5.mem_out[0]"/>
</bit>
```

Other information may depend on the type of configuration procotol.

memory_bank

- `bl`: Bit line address information
- `wl`: Word line address information

A quick example:

```
<bit id="0" value="1" path="fpga_top.grid_clb_1__2_.logical_tile_clb_mode_clb__0.
↪mem_fle_9_in_5.mem_out[0]"/>
  <bl address="000000"/>
  <wl address="000000"/>
</bit>
```

frame_based

- `frame`: frame address information

Note: Frame address may include don't care bit which is denoted as `x`.

A quick example:

```
<bit id="0" value="1" path="fpga_top.grid_clb_1__2_.logical_tile_clb_mode_clb__0.
↪mem_fle_9_in_5.mem_out[0]"/>
  <frame address="0001000x00000x01"/>
</bit>
```

12.5 Bitstream Setting (.xml)

An example of bitstream settings is shown as follows. This can define a hard-coded bitstream for a reconfigurable resource in FPGA fabrics.

Warning: Bitstream setting is a feature for power-users. It may cause wrong bitstream to be generated. For example, the hard-coded bitstream is not compatible with LUTs whose nets may be swapped during routing stage (cause a change on the truth table as well as bitstream). It is users's responsibility to ensure correct bitstream.

```
<openfpga_bitstream_setting>
  <pb_type name="<string>" source="eblif" content=".param LUT" is_mode_select_bistream=
↪"true" bitstream_offset="1"/>
  <interconnect name="<string>" default_path="<string>"/>
</openfpga_bitstream_setting>
```

12.5.1 pb_type-related Settings

The following syntax are applicable to the XML definition tagged by `pb_type` in bitstream setting files.

name="<string>"

The `pb_type` name to be constrained, which should be the full path of a `pb_type` consistent with VPR's architecture description. For example,

```
pb_type="clb.fle[arithmetic].soft_adder.adder_lut4"
```

source="<string>"

The source of the `pb_type` bitstream, which could be from a `.eblif` file. For example,

```
source="eblif"
```

content="<string>"

The content of the `pb_type` bitstream, which could be a keyword in a `.eblif` file. For example, `content=".attr LUT"` means that the bitstream will be extracted from the `.attr LUT` line which is defined under the `.blif model` (that is defined under the `pb_type` in VPR architecture file).

is_mode_select_bitstream="<bool>"

Can be either `true` or `false`. When set `true`, the bitstream is considered as mode-selection bitstream, which may overwrite `mode_bits` definition in `pb_type_annotation` of OpenFPGA architecture description. (See details in *Primitive Blocks inside Multi-mode Configurable Logic Blocks*)

bitstream_offset="<int>"

Specify the offset to be applied when overloading the bitstream to a target. For example, a LUT may have a 16-bit bitstream. When `offset=1`, bitstream overloading will skip the first bit and start from the second bit of the 16-bit bitstream.

12.5.2 Interconnection-related Settings

The following syntax are applicable to the XML definition tagged by `interconnect` in bitstream setting files.

name="<string>"

The `interconnect` name to be constrained, which should be the full path of a `pb_type` consistent with VPR's architecture description. For example,

```
pb_type="clb.fle[arithmetic].mux1"
```

default_path="<string>"

The default path denotes an input name that is consistent with VPR's architecture description. For example, in VPR architecture, there is a mux defined as

```
<mux name="mux1" input="iopad.inpad ff.Q" output="io.inpad"/>
```

The default path can be either `iopad.inpad` or `ff.Q` which corresponds to the first input and the second input respectively.

12.6 Fabric Key (.xml)

A fabric key follows an XML format. As shown in the following XML code, the key file includes the organization of configurable blocks in the top-level FPGA fabric.

12.6.1 Configurable Region

The top-level FPGA fabric can consist of several configurable regions, where a region may contain one or multiple configurable blocks. Each configurable region can be configured independently and in parrallel.

<region id="<int>"/>

- `id` indicates the unique id of a configurable region in the fabric.

> **Warning:** The id must start from zero!

> **Note:** The number of regions defined in the fabric key must be consistent with the number of regions defined in the configuration protocol of architecture description. (See details in *Configuration Protocol*).

The following example shows how to define multiple configuration regions in the fabric key.

```
<fabric_key>
  <region id="0">
    <bl_shift_register_banks>
        <bank id="0" range="bl[0:24]"/>
        <bank id="1" range="bl[25:40]"/>
    </bl_shift_register_banks>
    <wl_shift_register_banks>
        <bank id="0" range="wl[0:19],wl[40:59]"/>
        <bank id="1" range="wl[21:39],wl[60:69]"/>
```

(continues on next page)

(continued from previous page)

```
    </wl_shift_register_banks>
    <key id="0" name="grid_io_bottom" value="0" alias="grid_io_bottom_1__0_"/>
    <key id="1" name="grid_io_right" value="0" alias="grid_io_right_2__1_"/>
    <key id="2" name="sb_1__1_" value="0" alias="sb_1__1_"/>
  </region>
  <region id="1">
    <bl_shift_register_banks>
        <bank id="0" range="bl[0:24]"/>
        <bank id="1" range="bl[25:40]"/>
    </bl_shift_register_banks>
    <wl_shift_register_banks>
        <bank id="0" range="wl[0:19]"/>
    </wl_shift_register_banks>
    <key id="3" name="cbx_1__1_" value="0" alias="cbx_1__1_"/>
    <key id="4" name="grid_io_top" value="0" alias="grid_io_top_1__2_"/>
    <key id="5" name="sb_0__1_" value="0" alias="sb_0__1_"/>
  </region>
  <region id="2">
    <bl_shift_register_banks>
        <bank id="0" range="bl[0:24]"/>
        <bank id="1" range="bl[25:40]"/>
        <bank id="2" range="bl[41:59]"/>
    </bl_shift_register_banks>
    <wl_shift_register_banks>
        <bank id="0" range="wl[0:19]"/>
        <bank id="1" range="wl[21:39]"/>
    </wl_shift_register_banks>
    <key id="6" name="sb_0__0_" value="0" alias="sb_0__0_"/>
    <key id="7" name="cby_0__1_" value="0" alias="cby_0__1_"/>
    <key id="8" name="grid_io_left" value="0" alias="grid_io_left_0__1_"/>
  </region>
  <region id="3">
    <bl_shift_register_banks>
        <bank id="0" range="bl[0:24]"/>
        <bank id="1" range="bl[25:40]"/>
    </bl_shift_register_banks>
    <wl_shift_register_banks>
        <bank id="0" range="wl[0:19]"/>
        <bank id="1" range="wl[21:39]"/>
        <bank id="2" range="wl[40:49]"/>
    </wl_shift_register_banks>
    <key id="9" name="sb_1__0_" value="0" alias="sb_1__0_"/>
    <key id="10" name="cbx_1__0_" value="0" alias="cbx_1__0_"/>
    <key id="11" name="cby_1__1_" value="0" alias="cby_1__1_"/>
    <key id="12" name="grid_clb" value="0" alias="grid_clb_1__1_"/>
  </region>
</fabric_key>
```

12.6.2 Configurable Block

Each configurable block is defined as a key. There are two ways to define a key, either with alias or with name and value.

`<key id="<int>" alias="<string>" name="<string>" value="<int>"/>`

- `id` indicates the sequence of the configurable memory block in the top-level FPGA fabric.
- `name` indicates the module name of the configurable memory block. This property becomes optional when `alias` is defined.
- `value` indicates the instance id of the configurable memory block in the top-level FPGA fabric. This property becomes optional when `alias` is defined.
- `alias` indicates the instance name of the configurable memory block in the top-level FPGA fabric. If a valid alias is specified, the `name` and `value` are not required.
- `column` indicates the relative x coordinate for a configurable memory in a configurable region at the top-level FPGA fabric. This is required when the memory bank protocol is selection.

  Note: The configurable memory blocks in the same column will share the same Bit Line (BL) bus

- `row` indicates the relative y coordinate for a configurable memory in a configurable region at the top-level FPGA fabric. This is required when the memory bank protocol is selection.

  Note: The configurable memory blocks in the same row will share the same Word Line (WL) bus

Warning: For fast loading of fabric key, strongly recommend to use pairs `name` and `alias` or `name` and `value` in the fabric key file. Using only `alias` may cause long parsing time for fabric key.

The following is an example of a fabric key generate by OpenFPGA for a 2 × 2 FPGA. This key contains only `alias` which is easy to craft.

```
<fabric_key>
  <region id="0">
    <key id="0" alias="sb_2__2_"/>
    <key id="1" alias="grid_clb_2_2"/>
    <key id="2" alias="sb_0__1_"/>
    <key id="3" alias="cby_0__1_"/>
    <key id="4" alias="grid_clb_2_1"/>
    <key id="5" alias="grid_io_left_0_1"/>
    <key id="6" alias="sb_1__0_"/>
    <key id="7" alias="sb_1__1_"/>
    <key id="8" alias="cbx_2__1_"/>
    <key id="9" alias="cby_1__2_"/>
    <key id="10" alias="grid_io_right_3_2"/>
    <key id="11" alias="cbx_2__0_"/>
    <key id="12" alias="cby_1__1_"/>
    <key id="13" alias="grid_io_right_3_1"/>
    <key id="14" alias="grid_io_bottom_1_0"/>
    <key id="15" alias="cby_2__1_"/>
```

(continues on next page)

(continued from previous page)

```
    <key id="16" alias="sb_2__1_"/>
    <key id="17" alias="cbx_1__0_"/>
    <key id="18" alias="grid_clb_1_2"/>
    <key id="19" alias="cbx_1__2_"/>
    <key id="20" alias="cbx_2__2_"/>
    <key id="21" alias="sb_2__0_"/>
    <key id="22" alias="sb_1__2_"/>
    <key id="23" alias="cby_0__2_"/>
    <key id="24" alias="sb_0__0_"/>
    <key id="25" alias="grid_clb_1_1"/>
    <key id="26" alias="cby_2__2_"/>
    <key id="27" alias="grid_io_top_2_3"/>
    <key id="28" alias="sb_0__2_"/>
    <key id="29" alias="grid_io_bottom_2_0"/>
    <key id="30" alias="cbx_1__1_"/>
    <key id="31" alias="grid_io_top_1_3"/>
    <key id="32" alias="grid_io_left_0_2"/>
  </region>
</fabric_key>
```

The following shows another example of a fabric key generate by OpenFPGA for a 2 × 2 FPGA. This key contains only `name` and `value` which is fast to parse.

```
<fabric_key>
  <region id="0">
    <key id="0" name="sb_2__2_" value="0"/>
    <key id="1" name="grid_clb" value="3"/>
    <key id="2" name="sb_0__1_" value="0"/>
    <key id="3" name="cby_0__1_" value="0"/>
    <key id="4" name="grid_clb" value="2"/>
    <key id="5" name="grid_io_left" value="0"/>
    <key id="6" name="sb_1__0_" value="0"/>
    <key id="7" name="sb_1__1_" value="0"/>
    <key id="8" name="cbx_1__1_" value="1"/>
    <key id="9" name="cby_1__1_" value="1"/>
    <key id="10" name="grid_io_right" value="1"/>
    <key id="11" name="cbx_1__0_" value="1"/>
    <key id="12" name="cby_1__1_" value="0"/>
    <key id="13" name="grid_io_right" value="0"/>
    <key id="14" name="grid_io_bottom" value="0"/>
    <key id="15" name="cby_2__1_" value="0"/>
    <key id="16" name="sb_2__1_" value="0"/>
    <key id="17" name="cbx_1__0_" value="0"/>
    <key id="18" name="grid_clb" value="1"/>
    <key id="19" name="cbx_1__2_" value="0"/>
    <key id="20" name="cbx_1__2_" value="1"/>
    <key id="21" name="sb_2__0_" value="0"/>
    <key id="22" name="sb_1__2_" value="0"/>
    <key id="23" name="cby_0__1_" value="1"/>
    <key id="24" name="sb_0__0_" value="0"/>
    <key id="25" name="grid_clb" value="0"/>
    <key id="26" name="cby_2__1_" value="1"/>
```

(continues on next page)

(continued from previous page)

```
    <key id="27" name="grid_io_top" value="1"/>
    <key id="28" name="sb_0__2_" value="0"/>
    <key id="29" name="grid_io_bottom" value="1"/>
    <key id="30" name="cbx_1__1_" value="0"/>
    <key id="31" name="grid_io_top" value="0"/>
    <key id="32" name="grid_io_left" value="1"/>
  </region>
</fabric_key>
```

The following shows another example of a fabric key generate by OpenFPGA for a 2 × 2 FPGA using memory bank. This key contains only `name`, `value`, `row` and `column`.

```
<fabric_key>
  <region id="0">
    <key id="0" name="sb_2__2_" value="0" alias="sb_2__2_" column="5" row="5"/>
    <key id="1" name="grid_clb" value="3" alias="grid_clb_2__2_" column="4" row="4"/>
    <key id="2" name="sb_0__1_" value="0" alias="sb_0__1_" column="1" row="3"/>
    <key id="3" name="cby_0__1_" value="0" alias="cby_0__1_" column="1" row="2"/>
    <key id="4" name="grid_clb" value="2" alias="grid_clb_2__1_" column="4" row="2"/>
    <key id="5" name="grid_io_left" value="0" alias="grid_io_left_0__1_" column="0" row=
↪"2"/>
    <key id="6" name="sb_1__0_" value="0" alias="sb_1__0_" column="3" row="1"/>
    <key id="7" name="sb_1__1_" value="0" alias="sb_1__1_" column="3" row="3"/>
    <key id="8" name="cbx_1__1_" value="1" alias="cbx_2__1_" column="4" row="3"/>
    <key id="9" name="cby_1__1_" value="1" alias="cby_1__2_" column="3" row="4"/>
    <key id="10" name="grid_io_right" value="0" alias="grid_io_right_3__2_" column="6"␣
↪row="4"/>
    <key id="11" name="cbx_1__0_" value="1" alias="cbx_2__0_" column="4" row="1"/>
    <key id="12" name="cby_1__1_" value="0" alias="cby_1__1_" column="3" row="2"/>
    <key id="13" name="grid_io_right" value="1" alias="grid_io_right_3__1_" column="6"␣
↪row="2"/>
    <key id="14" name="grid_io_bottom" value="1" alias="grid_io_bottom_1__0_" column="2"␣
↪row="0"/>
    <key id="15" name="cby_2__1_" value="0" alias="cby_2__1_" column="5" row="2"/>
    <key id="16" name="sb_2__1_" value="0" alias="sb_2__1_" column="5" row="3"/>
    <key id="17" name="cbx_1__0_" value="0" alias="cbx_1__0_" column="2" row="1"/>
    <key id="18" name="grid_clb" value="1" alias="grid_clb_1__2_" column="2" row="4"/>
    <key id="19" name="cbx_1__2_" value="0" alias="cbx_1__2_" column="2" row="5"/>
    <key id="20" name="cbx_1__2_" value="1" alias="cbx_2__2_" column="4" row="5"/>
    <key id="21" name="sb_2__0_" value="0" alias="sb_2__0_" column="5" row="1"/>
    <key id="22" name="sb_1__2_" value="0" alias="sb_1__2_" column="3" row="5"/>
    <key id="23" name="cby_0__1_" value="1" alias="cby_0__2_" column="1" row="4"/>
    <key id="24" name="sb_0__0_" value="0" alias="sb_0__0_" column="1" row="1"/>
    <key id="25" name="grid_clb" value="0" alias="grid_clb_1__1_" column="2" row="2"/>
    <key id="26" name="cby_2__1_" value="1" alias="cby_2__2_" column="5" row="4"/>
    <key id="27" name="grid_io_top" value="1" alias="grid_io_top_2__3_" column="4" row="6
↪"/>
    <key id="28" name="sb_0__2_" value="0" alias="sb_0__2_" column="1" row="5"/>
    <key id="29" name="grid_io_bottom" value="0" alias="grid_io_bottom_2__0_" column="4"␣
↪row="0"/>
    <key id="30" name="cbx_1__1_" value="0" alias="cbx_1__1_" column="2" row="3"/>
    <key id="31" name="grid_io_top" value="0" alias="grid_io_top_1__3_" column="2" row="6
```

(continues on next page)

(continued from previous page)

```
↪"/>
    <key id="32" name="grid_io_left" value="1" alias="grid_io_left_0__2_" column="0" row=
↪"4"/>
  </region>
</fabric_key>
```

12.6.3 BL Shift Register Banks

Note: The customizable is only available when the shift-register-based memory bank is selected in *Configuration Protocol*

Each Bit-Line (BL) shift register bank is defined in the code block <bl_shift_register_banks>. A shift register bank may contain multiple shift register chains. - each shift register chain can be defined using the bank syntax - the BLs controlled by each chain can be customized through the range syntax.

<bank id="<int>" range="<ports>"/>

- id indicates the sequence of the shift register chain in the bank. The id denotes the index in the head or tail bus. For example, id="0" means the head or tail of the shift register will be in the first bit of a head bus head[0:4]
- range indicates BL port to be controlled by this shift register chain. Multiple BL ports can be defined but the sequence matters. For example, bl[0:3], bl[6:10] infers a 9-bit shift register chain whose output ports are connected from bl[0] to bl[10].

  Note: When creating the range, you must know the number of BLs in the configuration region

  Note: ports must use bl as the reserved port name

12.6.4 WL Shift Register Banks

Note: The customizable is only available when the shift-register-based memory bank is selected in *Configuration Protocol*

Each Word-Line (WL) shift register bank is defined in the code block <wl_shift_register_banks>. A shift register bank may contain multiple shift register chains. - each shift register chain can be defined using the bank syntax - the BLs controlled by each chain can be customized through the range syntax.

<bank id="<int>" range="<ports>"/>

- id indicates the sequence of the shift register chain in the bank. The id denotes the index in the head or tail bus. For example, id="0" means the head or tail of the shift register will be in the first bit of a head bus head[0:4]
- range indicates WL port to be controlled by this shift register chain. Multiple WL ports can be defined but the sequence matters. For example, wl[0:3], wl[6:10] infers a 9-bit shift register chain whose output ports are connected from wl[0] to wl[10].

Note: When creating the range, you must know the number of BLs in the configuration region

Note: ports must use `wl` as the reserved port name

12.7 I/O Mapping File (.xml)

The I/O mapping file aims to show

- What nets have been mapped to each I/O
- What is the directionality of each mapped I/O

An example of design constraints is shown as follows.

```
<io_mapping>
    <io name="gfpga_pad_GPIO_PAD[6:6]" net="a" dir="input"/>
    <io name="gfpga_pad_GPIO_PAD[1:1]" net="b" dir="input"/>
    <io name="gfpga_pad_GPIO_PAD[9:9]" net="out_c" dir="output"/>
</io_mapping>
```

name="<string>"

The pin name of the FPGA fabric which has been mapped, which should be a valid pin defined in OpenFPGA architecture description.

Note: You should be find the exact pin in the top-level module of FPGA fabric if you output the Verilog netlists.

net="<string>"

The net name which is actually mapped to a pin, which should be consistent with net definition in your `.blif` file.

dir="<string>"

The direction of an I/O, which can be either `input` or `output`.

12.8 I/O Information File (.xml)

Note: This file is in a different usage than the I/O mapping file (see details in *I/O Mapping File (.xml)*)

The I/O information file aims to show

- The number of I/O in an FPGA fabric
- The name of each I/O in an FPGA fabric
- The coordinate (in VPR domain) of each I/O in an FPGA fabric

An example of the file is shown as follows.

```
<io_coordinates>
  <io pad="gfpga_pad_GPIO_PAD[0]" x="1" y="2" z="0"/>
  <io pad="gfpga_pad_GPIO_PAD[1]" x="1" y="2" z="1"/>
  <io pad="gfpga_pad_GPIO_PAD[2]" x="1" y="2" z="2"/>
  <io pad="gfpga_pad_GPIO_PAD[3]" x="1" y="2" z="3"/>
  <io pad="gfpga_pad_GPIO_PAD[4]" x="1" y="2" z="4"/>
  <io pad="gfpga_pad_GPIO_PAD[5]" x="1" y="2" z="5"/>
  <io pad="gfpga_pad_GPIO_PAD[6]" x="1" y="2" z="6"/>
  <io pad="gfpga_pad_GPIO_PAD[7]" x="1" y="2" z="7"/>
  <io pad="gfpga_pad_GPIO_PAD[8]" x="2" y="1" z="0"/>
  <io pad="gfpga_pad_GPIO_PAD[9]" x="2" y="1" z="1"/>
  <io pad="gfpga_pad_GPIO_PAD[10]" x="2" y="1" z="2"/>
  <io pad="gfpga_pad_GPIO_PAD[11]" x="2" y="1" z="3"/>
  <io pad="gfpga_pad_GPIO_PAD[12]" x="2" y="1" z="4"/>
  <io pad="gfpga_pad_GPIO_PAD[13]" x="2" y="1" z="5"/>
  <io pad="gfpga_pad_GPIO_PAD[14]" x="2" y="1" z="6"/>
  <io pad="gfpga_pad_GPIO_PAD[15]" x="2" y="1" z="7"/>
</io_coordinates>
```

pad="<string>"

The port name of the I/O in FPGA fabric, which should be a valid port defined in output Verilog netlist.

> **Note:** You should be find the exact pin in the top-level module of FPGA fabric if you output the Verilog netlists.

x="<int>"

The x coordinate of the I/O in VPR coordinate system.

y="<int>"

The y coordinate of the I/O in VPR coordinate system.

z="<int>"

The z coordinate of the I/O in VPR coordinate system.

12.9 Bitstream Distribution File (.xml)

The bitstream distribution file aims to show

- region-level bitstream distribution - The total number of configuration bits under each region
- block-level bitstream distribution - The total number of configuration bits under each block - The number of configuration bits per block

An example of the file is shown as follows.

```
<bitstream_distribution>
  <regions>
    <region id="0" number_of_bits="2250">
    </region>
  </regions>
  <blocks>
    <block name="fpga_top" number_of_bits="2250">
      <block name="grid_clb_1__1_" number_of_bits="1700">
```

(continues on next page)

(continued from previous page)

```
      </block>
      <block name="grid_io_top_1__2_" number_of_bits="8">
      </block>
      <block name="grid_io_right_2__1_" number_of_bits="8">
      </block>
      <block name="grid_io_bottom_1__0_" number_of_bits="8">
      </block>
      <block name="grid_io_left_0__1_" number_of_bits="8">
      </block>
      <block name="sb_0__0_" number_of_bits="40">
      </block>
      <block name="sb_0__1_" number_of_bits="40">
      </block>
      <block name="sb_1__0_" number_of_bits="40">
      </block>
      <block name="sb_1__1_" number_of_bits="40">
      </block>
      <block name="cbx_1__0_" number_of_bits="88">
      </block>
      <block name="cbx_1__1_" number_of_bits="94">
      </block>
      <block name="cby_0__1_" number_of_bits="88">
      </block>
      <block name="cby_1__1_" number_of_bits="88">
      </block>
    </block>
  </blocks>
</bitstream_distribution>
```

12.9.1 Region-Level Bitstream Distribution

Region-level bitstream distribution is shown under the <regions> code block

id="<string>"

The unique index of the region, which can be found in the *Fabric Key (.xml)*

number_of_bits="<string>"

The total number of configuration bits in this region

12.9.2 Block-Level Bitstream Distribution

Block-level bitstream distribution is shown under the <blocks> code block

name="<string>"

The block name represents the instance name which you can find in the fabric netlists

number_of_bits="<string>"

The total number of configuration bits in this block

12.10 Bus Group File (.xml)

The bus group file aims to show

- How bus ports are flatten by EDA engines, e.g., synthesis.
- What are the pins in post-routing corresponding to the bus ports before synthesis

An example of file is shown as follows.

```
<bus_group>
  <bus name="i_addr[0:3]" big_endian="false">
    <pin id="0" name="i_addr_0_"/>
    <pin id="1" name="i_addr_1_"/>
    <pin id="2" name="i_addr_2_"/>
    <pin id="3" name="i_addr_3_"/>
  </bus>
</bus_group>
```

12.10.1 Bus-related Syntax

name="<string>"

The bus port defined before synthesis, e.g., addr[0:3]

big_endian="<bool>"

Specify if this port should follow big endian or little endian in Verilog netlist. By default, big endian is assumed, e.g., addr[0:3].

12.10.2 Pin-related Syntax

id="<int>"

The index of the current pin in a bus port. The index must be the range of **[LSB, MSB-1]** that are defined in the bus.

name="<string>"

The pin name after bus flatten in synthesis results

12.11 Pin Constraints File (.pcf)

Note: This file is in a different usage than the Pin Constraints File in XML format (see details in *Pin Constraints File (.xml)*)

The PCF file is the file which **users** should craft to assign their I/O constraints

An example of the file is shown as follows.

```
set_io a pad_fpga_io[0]
set_io b[0] pad_fpga_io[4]
set_io c[1] pad_fpga_io[6]
```

set_io <net> <pin>

Assign a net (defined as an input or output in users' HDL design) to a specific pin of an FPGA device (typically a packaged chip).

Note: The net should be single-bit and match the port declaration of the top-module in users' HDL design

Note: FPGA devices have different pin names, depending their naming rules. Please contact your vendor about details.

12.12 Pin Table File (.csv)

Note: This file is typically a spreadsheet provided by FPGA vendors. Please contact your vendor for the exact file.

Note: OpenFPGA will not include or guarantee the correctness of the file!!!

The pin table file is the file which describes the pin mapping between a chip and an FPGA inside the chip.

An example of the file is shown as follows.

```
orientation,row,col,pin_num_in_cell,port_name,mapped_pin,GPIO_type,Associated Clock,
↪Clock Edge
TOP,,,,gfpga_pad_IO_A2F[0],pad_fpga_io[0],,,
TOP,,,,gfpga_pad_IO_F2A[0],pad_fpga_io[0],,,
TOP,,,,gfpga_pad_IO_A2F[4],pad_fpga_io[1],,,
TOP,,,,gfpga_pad_IO_F2A[4],pad_fpga_io[1],,,
TOP,,,,gfpga_pad_IO_A2F[8],pad_fpga_io[2],,,
TOP,,,,gfpga_pad_IO_F2A[8],pad_fpga_io[2],,,
TOP,,,,gfpga_pad_IO_A2F[31],pad_fpga_io[3],,,
TOP,,,,gfpga_pad_IO_F2A[31],pad_fpga_io[3],,,
RIGHT,,,,gfpga_pad_IO_A2F[32],pad_fpga_io[4],,,
RIGHT,,,,gfpga_pad_IO_F2A[32],pad_fpga_io[4],,,
RIGHT,,,,gfpga_pad_IO_A2F[40],pad_fpga_io[5],,,
RIGHT,,,,gfpga_pad_IO_F2A[40],pad_fpga_io[5],,,
BOTTOM,,,,gfpga_pad_IO_A2F[64],pad_fpga_io[6],,,
BOTTOM,,,,gfpga_pad_IO_F2A[64],pad_fpga_io[6],,,
LEFT,,,,gfpga_pad_IO_F2A[127],pad_fpga_io[7],,,
LEFT,,,,gfpga_pad_IO_A2F[127],pad_fpga_io[7],,,
```

An pin table may serve in various purposes. However, for OpenFPGA, the following attributes are required

orientation

Specify on which side the pin locates

port_name

Specify the port name of the FPGA fabric

`mapped_pin`

Specify the pin name of the FPGA chip

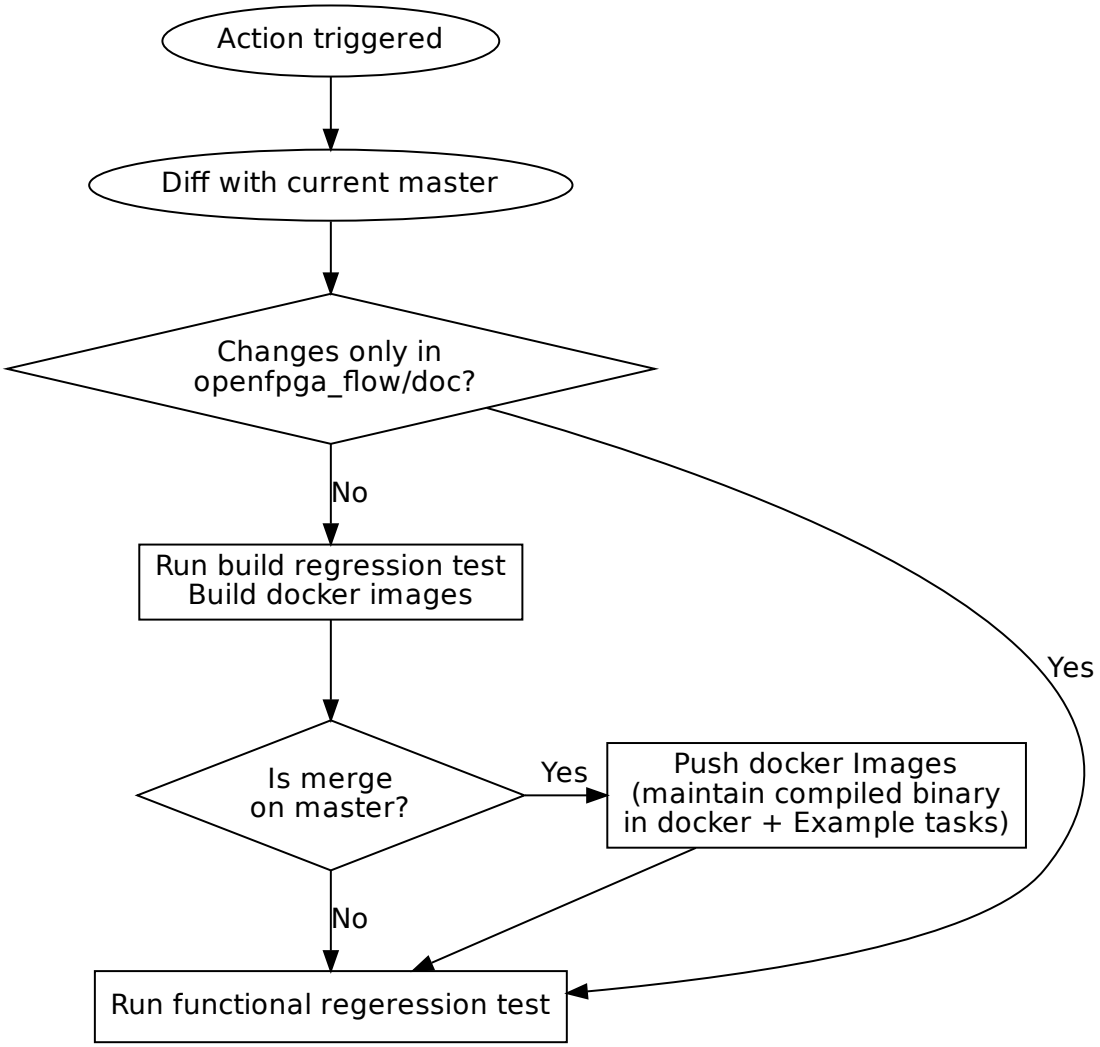
Warning: Currently, the direction of the port is inferred by the `port_name`. A postfix of `A2F` indicates an input port, while a postfix of `F2A` indicates an output port.

CHAPTER

THIRTEEN

CI/CD SETUP

OpenFPGA implements CI/CD system using Github actions. The following figure shows the Actions implements flow. The source building is skipped if there are changes only in `openfpga_flow` or `docs` directory, in which case the docker image compiled for the latest master branch is used for running a regression.



Build `regression test`

The OpenFPGA soure is compiled with the following set of compilers.

1. gcc-5
2. gcc-6
3. gcc-7
4. gcc-8
5. gcc-9
6. clang-6.0
7. clang-8

The docker images for these build enviroment are available on github packages.

Functional `regeression test`

OpenFPGA maintains a set of functional tests to validate the different functionality. The test are broadly catagories into `basic_reg_test`, `fpga_verilog_reg_test`, `fpga_bitstream_reg_test`, `fpga_sdc_reg_test`, and `fpga_spice_reg_test`. A functional regression test is run for every commit on every branch.

13.1 How to debug failed regression test

In case the `funtional regression test` fails, the actions script will collect all `.log` files from the task directory and upload as a artifacts on github storage. These artifacts can be downloaded from the github website actions tab, for more reference follow this article.

NOTE : The retention time of these artifacts is 1 day, so in case user want to reserve the failure log for longer duration back it up locally

13.2 Release Docker Images

ghcr.io/lnis-uofu/openfpga-master:latest

This is a bleeding-edge release from the current master branch of OpenFPGA. It is updated automatically whenever there is activity on the master branch. Due to high development activity, we recommend the user to use the bleeding-edge version to get access to all new features and report an issue in case there are any bugs.

13.3 CI after cloning repository

If you clone the repository the CI setup will still function, except the based images are still pullled from "lnis-uofu" repsitory and the master branch of cloned repo will not push final docker image to any repository .

In case you want to host your own copies of OpenFPGA base images and final release create a githib secret variable with name `DOCKER_REPO` and set it to `true`. This will make ci script to download base images from your own repo pakcages, and upload final realse to the same.

If you don not want to use docker images based regression test and like to compile all the bianries for each CI run. You can set `IGNORE_DOCKER_TEST` secrete variable to `true`.

Note: Once you add `DOCKER_REPO` variable, you need to genrerate base images. To do this trigger mannual workflow `Build docker CI images`

CHAPTER

FOURTEEN

VERSION NUMBER

14.1 Convention

OpenFPGA follows the semantic versioning, where the version number is in the form of

```
[Major].[Minor].[Patch]
```

For example, version `1.2.300` denotes

- One major milestone is achieved.
- Two minor milestone is achieved after the major revision `1.0.0`
- `300` patches has been applied after the minor revision `1.2.0`

14.2 Version Update Rules

Warning: Please discuss with maintainers before modifying major and minor numbers.

Warning: Please do not modify patch number manually.

To update the version number, please follow the rules:

- Major and minor version number are defined by maintainers
- Patch number is automatically updated through github actions. See detailed in the workflow file

Version updates are made in the following scenario

- When a minor milestone is achieved, the minor revision number can be increased by 1. The following condition is considered as a minor milestone: - a new feature has been developed. - a critical patch has been applied. - a sufficient number of small patches has been applied in the past quarter. In other words, the minor revision will be updated by the end of each quarter as long as there are patches.
- When several minor milestones are achieved, the major revision number can be increased by 1. The following condition is considered as a major milestone: - significant improvements on Quality-of-Results (QoR). - significant changes on user interface. - a techical feature is developed and validated by the community, which can impact the complete design flow.

CHAPTER

FIFTEEN

REGRESSION TESTS

Regression tests are designed to cover various technical features of the OpenFPGA projects, including but not limited to

- Netlist generation
- Netlist verification
- Bitstream generation

Considering the large number of technical features, regression tests are categorized into several groups, which can be found at `openfpga_flow/regression_test_scripts/`

15.1 Run a Test

Note: Make sure you have compiled OpenFPGA and set up your environment before reaching this step. See details in getting_started_tutorials.

To run a regression test, users can execute a shell script (assume you are under the root directory of the project), for example,

```
./openfpga_flow/regression_test_scripts/basic_reg_test.sh [OPTIONS]
```

Note: `basic_reg_test` can be replaced by other tests which are under `openfpga_flow/regression_test_scripts/`

15.2 Test Options

There are a few options available when running the tests.

`--debug`

This option can turn on debug mode when running regression tests. By default it is `off`.

`--show_thread_logs`

This option can enable verbose output when running regression tests. By default it is `off`.

Note: To avoid massive outputs, suggest to run the tests with default options. In CI, always recommend to turn on the debug and verbose options

`--remove_run_dir all`

This option is to remove all the previous run results for a specific regression test. Suggest to use when there are limited disk space.

Note: Be careful before using this option! It may cause permanent loss on test results.

CHAPTER

SIXTEEN

CONTACT

General questions:

Prof. Pierre-Emmanuel Gaillardon

pierre-emmanuel.gaillardon@utah.edu

Technical Details about FPGA-SPICE/Verilog/Bitstream/SDC:

Dr. Xifan Tang

xifan@osfpga.org

Technical Details about physical design

Ganesh Gore

ganesh.gore@utah.edu

CHAPTER

SEVENTEEN

PUBLICATIONS & REFERENCES

CHAPTER

EIGHTEEN

FREQUENTLY ASKED QUESTIONS

18.1 Where is the best place to get help with OpenFPGA?

Currently, we have an active github issues page found here. Users can see if their questions have already been answered by searching the open or closed issues, and users are recommended to post questions there first. Asking questions on the github issues page allows us to answer the question for everyone who may be experiencing similar problems as well.

18.2 What should I do if check-in tests failed when first installing OpenFPGA?

First, check to make sure all dependencies for OpenFPGA and Python have been installed and are up-to-date on the desired device. To see the full list of depenencies, please visit our github dependencies page. This issue has been discussed in issue 280.

18.3 How to sweep design parameters in a task run of OpenFPGA design flow?

Testing multiple script parameters for a variable is possible by modifying the task.conf file. Doing so will create a job for each combination of the variables. A solution is discussed in issue 228.

18.4 How do I setup OpenFPGA to be used by multiple users on a single device?

OpenFPGA can support multiple users on a shared device using the environment variable `OPENFPGA_ROOT`. The OpenFPGA script for running tasks needs `OPENFPGA_ROOT` to be the path to the OpenFPGA root directory. Users can then run the script on a task in the current working directory. A solution is discussed in issue 209.

18.5 How do I contribute to OpenFPGA?

Users of OpenFPGA that are interested in contributing must complete the following:

- Create a branch. For external collaborators, please fork the repository first and create a branch in the fork.
- Creatre a pull request and fill out our pull request template. It is easy for us to acknowledge and review your pull request.
- Wait or keep debugging until all the CI tests pass.
- Request for a review. You may expect several rounds of review and discussion before the pull request is approved.

For more information on the VTR see vtr_doc or vtr_github

For more information on the Yosys see yosys_doc or yosys_github

For more information on the original FPGA architecture description language see xml_vtr

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[BRM99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[GW12] J. B. Goeders and S. J. E. Wilton. VersaPower: Power Estimation for Diverse FPGA Architectures. In *2012 International Conference on Field-Programmable Technology*, 229–234. Dec 2012. doi:10.1109/FPT.2012.6412139.

[LAR11] Jason Luu, Jason Helge Anderson, and Jonathan Scott Rose. Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, 227–236. New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1950413.1950457, doi:10.1145/1950413.1950457.

[RLY+12] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, 77–86. New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2145694.2145708, doi:10.1145/2145694.2145708.

[TGM15] X. Tang, P. Gaillardon, and G. De Micheli. Fpga-spice: a simulation-based power estimation framework for fpgas. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, volume, 696–703. Oct 2015. doi:10.1109/ICCD.2015.7357183.

[TGA+19] X. Tang, E. Giacomin, A. Alacchi, B. Chauviere, and P. Gaillardon. Openfpga: an opensource framework enabling rapid prototyping of customizable fpgas. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, volume, 367–374. Sep. 2019. doi:10.1109/FPL.2019.00065.

[TGAG19] X. Tang, E. Giacomin, A. Alacchi, and P. Gaillardon. A study on switch block patterns for tileable fpga routing architectures. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, volume, 247–250. 2019. doi:10.1109/ICFPT47387.2019.00039.

[TGMG19] X. Tang, E. Giacomin, G. D. Micheli, and P. Gaillardon. FPGA-SPICE: A Simulation-Based Architecture Evaluation Framework for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(3):637–650, March 2019. doi:10.1109/TVLSI.2018.2883923.

INDEX

Symbols

A

B

C

D

F

G

I

L

M

V

W

X

Y

Z