AFRL-RY-WP-TR-2023-0135

AN AUTOMATIC INTELLECTUAL PROPERTY (IP) GENERATOR FOR CUSTOMIZABLE FIELD PROGRAMMABLE GATE ARRAY (FPGA) ARCHITECTURES

Pierre-Emmanuel Gaillardon
University of Utah

SEPTEMBER 2023
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2023-0135 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

CHRISTOPHER A. BOZADA
Program Manager
Aerospace Components & Subsystems Division

//Signature//

GENE M. WILKINS, Lt Col, USAF
Deputy Chief
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
September 2023	Final	6 June 2018	8 September 2022

4. TITLE AND SUBTITLE

AN AUTOMATIC INTELLECTUAL PROPERTY (IP) GENERATOR FOR CUSTOMIZABLE FIELD PROGRAMMABLE GATE ARRAY (FPGA) ARCHITECTURES

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8650-18-2-7855	N/A	62716E

5d. PROJECT NUMBER	5e. TASK NUMBER	5f. WORK UNIT NUMBER
N/A	N/A	Y1TN

6. AUTHOR(S)

Pierre-Emmanuel Gaillardon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Utah
257 South 1400 East
Salt Lake City, Utah

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'SREPORT NUMBER(S)
Air Force Research Laboratory, Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command, United States Air Forces	Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203	AFRL/RYD	AFRL-RY-WP-TR-2023-0135

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7855. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government. Report contains color.

14. ABSTRACT

This project describes the development of an automatic IP generator for customizable FPGA architectures, as key element to kick-start a viable open-source System-On-Chips (SoC) ecosystem as defined in DARPA ERI POSH program, Technical Area 2 (TA-2). Such tool can significantly reduce the development and research time required to embed custom FPGAs into SoCs, especially for DoD applications, such as next-generation unmanned aerial vehicles and advanced wireless/ Software Defined Radio (SDR) designs.

15. SUBJECT TERMS

Embedded field programmable gate arrays, system-on-a-chip, automatic intellectual property generation, customizable architectures.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	161

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Christopher Bozada	N/A

Table of Contents

Section **Page**

Section **Page**

Section **Page**

List of Figures

Figure **Page**

Figure **Page**

Figure **Page**

Figure **Page**

Figure **Page**

List of Tables

Table **Page**

SUMMARY

Physical design for Field Programmable Gate Array (FPGA) is challenging and time-consuming, primarily due to the use of a full-custom approach to aggressively optimize Performance, Power and Area (PPA) of the FPGA design. The growing number of FPGA applications demands novel architectures and shorter development cycles. The use of an automated toolchain is essential to reduce end-to-end development time. To enable high-quality custom FPGAs embedded into DoD SoCs with the least manual efforts, an automatic IP generator for customizable FPGA fabrics is required. This project developed an open-source framework enabling high-quality auto-generated customizable RTL descriptions of FPGA fabrics. The IP generator is based on the popular open-source architecture exploration tool for FPGAs, Versatile Placement and Routing (VPR). First a fully functional homogeneous FPGA core IP fabric and bitstream generator was developed, followed by the release of the automatically-generated heterogeneous FPGA core IP fabrics and secured bitstream loading protocols integrated to the FPGA IP framework. Along with the test chip, open-source testing suites and demonstration boards were developed to provide high-coverage testing and offer practical applications for full open-source FPGA fabrics developed in the POSH program.

1 INTRODUCTION

Field Programmable Gate Arrays have demonstrated themselves as a critical component in DoD applications, such as sonar, radar, unmanned vehicles, etc. Compared to Application Specific Integrated Circuits (ASICs), FPGAs allow full customization in data-processing and field updates for military SoCs. For example, FPGAs in military devices can be automatically deleted if the device was lost or captured by an enemy. In addition, FPGA implementations can often achieve better performance than general purpose processors. These advantages contribute to a rise in demand for both FPGA chips and IP fabrics in military and defense market in the last decade. Embedding proprietary FPGA IPs can bring high customization, in-system programmability and extended battery life to SoC implementations. Modern FPGA architectures are typically highly parameterized, leading to their routability and PPA to fit diverse demands required by SoC implementations. However, modern FPGA fabric IPs typically require intensive manual efforts in development, such as hand optimized layouts. Limited research and open-source efforts had been devoted to auto-generation of customizable FPGA fabric IP blocks, which is why this project addresses that need by providing an FPGA IP framework following the technical steps detailed below.

According to the scheduled plan, we have completed all tasks and milestones.

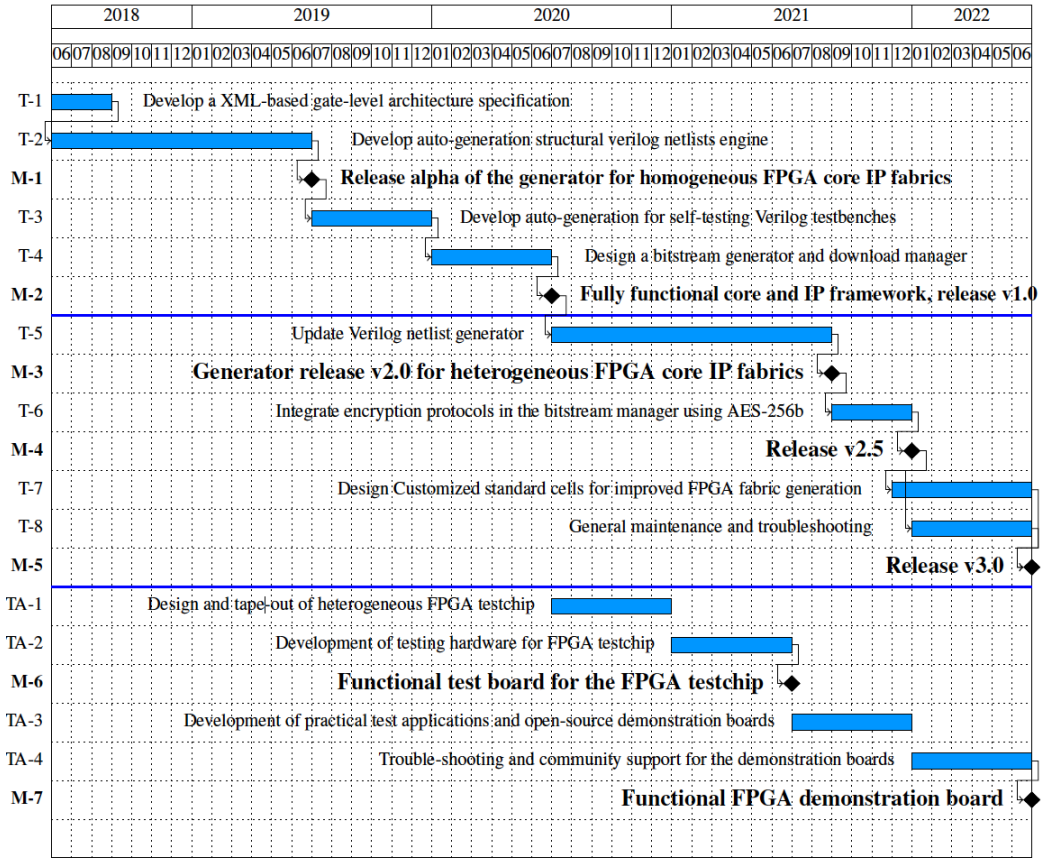


Figure 1: Schedule of Tasks and Milestones, all Tasks were Completed

2 ACTIVITY REPORT

Listed below is a diary of the major milestones and accomplishments we have accomplished:

- 09/12/2017: Public release of a simple documentation on the XML syntax on [1]
- 09/12/2017: Completion of the alpha version of the Verilog auto-generation engine for full homogeneous FPGA fabric
- 07/03/2018: Completion of the alpha version of the XML syntax that describes the port map and internal structure of a gate-level Verilog module
- 07/06/2018: Creation of a GitHub repository for the project [2]
- 07/18/2018: Completion of the alpha version of XML syntax to link the defined Verilog module to original FPGA architecture description language
- 07/18/2018: Completion of a simple documentation on the syntax
- 07/26/2018: Completion of the alpha version of the Verilog auto-generation engine for *Look-Up Tables*
  (LUTs)
- 07/26/2018: Release of a sample XML file describing a homogenous FPGA to the GitHub repository
- 08/03/2018: Completion of the alpha version of the Verilog auto-generation engine for multiplexers
- 08/10/2018: Release of the samples and templates of user-defined Verilog netlists, SRAMs, FFs, I/O pads, SCFFs (all formed from basic standard cells) which are required by homogenous FPGA architectures
- 08/18/2018: Refinement of the XML syntax for Verilog primitives in CLB architectures
- 08/23/2018: Completion of the alpha version of the Verilog auto-generation engine for flexible CLB architectures
- 09/04/2018: Update of the GitHub repository with refined syntax and main body of the alpha version
- 09/06/2018: Completion of the alpha version of the Verilog auto-generation engine for routing architecture
- 10/21/2018: Completion of the XML syntax to support self-testing testbench generations
- 10/23/2018: Back-annotation of VPR mapping results from routing resource graphs
- 10/29/2018: Successful CLB bitstream generation (homogeneous FPGAs only)
- 11/08/2018: Successful bitstream generation for the routing structures (homogeneous FPGAs only)
- 11/12/2018: Completion of the alpha version of the bitstream generator for homogeneous FPGAs

- 11/19/2018: Verilog testbench (no self-testing feature) generation for full fabric validation
- 11/27/2018: Successful verification of the Verilog testbench generator using toy BLIF benchmarks (Inverter/XOR/AND2 gates)
- 12/03/2018: EDA flow refinement, implementing YOSYS + VPR design flow and supporting full Verilog-to-Bitstream conversion
- 12/07/2018: Successful verification of the Verilog testbench using toy Verilog benchmarks (pipe-lined 3-bit and 8-bit adders)
- 12/12/2018: Completion of the self-testing Verilog testbench generation to validate full FPGA fabric
- 12/14/2018: Completion of the configuration-skip self-testing Verilog testbench generation to validate the core logic of the FPGA fabrics
- 01/17/2019: Completion of the XML syntax to support explicit port mapping to standard cells
- 01/23/2019: Update on the Verilog generator to support explicit port mapping
- 01/29/2019: Completion of the XML syntax to support fracturable LUTs
- 02/05/2019: Update on the Verilog generator to support explicit port mapping
- 02/07/2019: Update on the Verilog generator to support Icarus simulator
- 02/11/2019: Completion of the XML syntax to support multi-mode CLB architectures
- 02/25/2019: Completion of a XML example describing a Stratix-4 like FPGA using multi-mode syn- tax
- 02/25/2019: Completion of the Verilog generation supporting multi-mode CLB architectures
- 03/05/2019: Completion of the alpha version of the bitstream generator for multi-mode CLB architectures
- 03/11/2019: Completion of the alpha version of the SDC generator
- 03/12/2019: First VPR modularization conference call
- 03/12/2019: Upgrade of the Verilog generation to support formal verification method
- 03/15/2018: Successful formal verification on 8 MCNC benchmarks
- 03/29/2019: SDC generation for fabric timing constraint
- 04/03/2019: SDC generation for benchmark-dependent sign-off
- 04/10/2019: Preliminary performance analysis of FPGA fabric and state-of-the-art comparison
- 04/29/2019: Routing resource graph modularization (VPR8 framework)
- 05/03/2019: XML syntax and Verilog generation for customizable buffer insertion in LUTs
- 05/08/2019: Updated compilation system using CMake
- 05/13/2019: Upgraded SDC generator to support timing constraints for CLBs with loop-breakers

- 05/08/2019: Completion of architecture exploration for test chip
- 05/23/2019: Updated XML syntax, Verilog and Bitstream generation support memory-decoders-based configuration circuits
- 06/03/2019: Integration of regression test to TravisCI
- 06/11/2019: Identification of the redundancy in the connection and switch blocks
- 06/24/2019: Completed tileable routing architecture generation
- 07/16/2019: Tested tileable FPGA architectures over different switch block patterns
- 07/24/2019: Completed support on mapping to standard cell MUX2
- 07/30/2019: Tuned Yosys script to support versatile BRAM architectures
- 08/05/2019: Completed support on adding local encoders to routing multiplexers
- 08/09/2019: Completed LUT RAM support
- 08/15/2019: Start code reconstruction of OpenFPGA (to support VPR8)
- 08/20/2019: Deploy reconstructed regression test to TravisCI
- 09/10/2019: Completed RTL definition for the test chip fabric core
- 09/26/2019: Finished the cross-column/row clb-to-clb connection support in VPR; Built scan-chain in test chip RTL
- 09/26/2019: Refactored flow-run scripts supporting a complete XML-to-iVerilog design flow; Basic regression tests are online in Travis CI
- 10/10/2019: Design document about test chip is online with detailed plans on RTL finalization, PnR and testing
- 10/17/2019: Refactored module data structure and Verilog writer for graph-based module. The new Verilog writer engine for clbs is ready to be plugged-in
- 10/24/2019: RTL finalized for 32 ×32 fabric with spypads addition.
- 10/24/2019: Refactored FPGA-Verilog engine, being ready for testing
- 10/31/2019: Refactored FPGA-Bitstream and FPGA-Verilog are plugged-in and passed small artificial benchmarks and test-chip fabric
- 10/31/2019: Completed functional verification on the final RTL using Synopsys VCS
- 11/07/2019: Reforged engine passed MCNC big20 with single-mode and multi-mode test cases; Enriched regression test, covering compact routing, tileable routing, using external standard cells, heterogeneous blocks, *etc.*
- 11/07/2019: Built hard macros for repeatable tiles to be used in the 32 ×32 fabric
- 11/14/2019: Reforged engine passed heterogeneous architectures and artificial benchmarks; SDC generator has been reforged
- 11/14/2019: Obtain DRC-clean hard macros
- 11/21/2019: Completed presentation and discussed about collaboration with QuickLogic

during Andrea's site visit

- 12/05/2019: Reforged engine passed regression tests covering both MCNC big20 and EPFL bench- mark suites
- 12/05/2019: Completed version 1 of test chip GDS II
- 12/05/2019: Created a public release of OpenFPGA including the new features developed in past 6 months
- 12/12/2019: RTL finalized for version 2 of test chip
- 01/16/2020: Developed libopenfpgaarch library as a parser for the XML description of OpenFPGA architectures.
- 01/16/2020: Regenerated Verilog netlists for the test chip using the latest PnR-friendly features (du- plicated pins and reorganized configuration chains)
- 01/23/2020: Developed OpenFPGA shell library as the shell interface for OpenFPGA framework
- 01/31/2020: Completed scripts to run STA on test chip
- 02/06/2020: Finished packing results fix-up at post-routing
- 02/13/2020: Extracted timing results for CLBs, CBs and SBs by running STA on PnRed netlists
- 02/13/2020: Completed LUT truth table fix-up at post-packing
- 02/20/2020: Completed Verilog netlists binding to Skywater standard cell library
- 02/20/2020: Finished integration of Verilog generator to VPR8+OpenFPGA code base
- 02/27/2020: Completed IR-drop scripts for test chip
- 02/27/2020: Finished integration of bitstream generator to VPR8+OpenFPGA code base
- 03/05/2020: Finished integration of SDC generator to VPR8+OpenFPGA code base
- 03/05/2020: Created GitHub repository to enforce version control on the scripts for PnR test chip
- 03/05/2020: Completed python scripts to auto-generate I/O assignment from a spreadsheet
- 03/12/2020: Finished integration of tileable fabric generator to VPR8+OpenFPGA code base
- 03/12/2020: Reworked online documentation for OpenFPGA + VPR8 integration
- 03/26/2020: Support for routing channels passing through multi-height and multi-width grids (span- ning multiple tiles in a row/column)
- 04/02/2020: Release example architectures on Github about the I/O customization
- 04/09/2020: Developed debugging I/O support and automated spypad addition for test chip netlists
- 04/16/2020: DRC-clean GDSII for the 2 $\times$2 FPGA
- 04/23/2020: Upgraded SDC generator to constrain timing of primitive blocks and routing tracks de- fined in VPR XML

- 04/23/2020: LVS-clean GDSII for the 2 ×2 FPGA
- 05/07/2020: Release v0.9 version of GDSII which is DRC- and LVS-clean, as well as passed IR drop analysis
- 05/07/2020: Upgraded SDC generator to support time units, wildcards and flexible hierarchies
- 05/14/2020: Completed v1.0 version of GDSII (passed timing sign-offs) and submit to TAPO for the preliminary evaluation
- 05/21/2020: Pull requests about delayless switch names accepted to VPR upstream
- 05/28/2020: Finished frame-based configuration protocol support
- 05/28/2020: Release v2.1 version of GDSII (a 6 ×6 FPGA) passing all the sign-offs and submitted to TAPO
- 06/04/2020: Pull requests about packable modes accepted to VPR upstream
- 06/11/2020: Developed memory bank and vanilla configuration protocol support
- 06/11/2020: Completed PGA and BGA designs for GDSII v2.1 version and sent to QuickPak for review
- 06/18/2020: Landed fabric key feature in OpenFPGA framework
- 06/18/2020: Applied frame views in the top-level PnR of a 12 ×12 FPGA
- 06/25/2020: Support for interchangeable bitstream file format
- 06/25/2020: Developed and tested a customized clock network on a 2 ×2 FPGA
- 07/02/2020: Optimized netlist file sizes by using bus ports
- 07/02/2020: Successful integration of fabric key in the PnR practice of a 2 ×2 FPGA
- 07/09/2020: Optimized runtime and memory usage of bitstream generation for 100k-LUT FPGAs
- 07/09/2020: Deprecated VPR7 in OpenFPGA framework and related obsoleted codes
- 07/16/2020: Developed power-gating support
- 07/22/2020: Completed Yosys+OpenFPGA integration and deployed test cases to CI
- 08/27/2020: Finished backend practice for a 40k-LUT FPGA within 24 hours
- 09/06/2020: Submitted WOSET paper about OpenFPGA tooling
- 09/14/2020: Extended compiler compatibility for OpenFPGA
- 09/14/2020: Developed buffer customization support for routing multiplexers
- 09/24/2020: Supported versatile flip-flop/latch/SRAM designs with/without reset/set/enable signals
- 10/01/2020: Supported multi-region configuration chains
- 10/08/2020: Enriched documentation by adding technical features summary, tool capability and 3rd party tool support description

- 10/08/2020: Finished an alpha version of FPGA architecture for Skywater 130nm tape-outs
- 10/15/2020: Created SOFA GitHub repository and developed several variants of FPGA architecture to be taped out
- 10/22/2020: Finished QoR evaluation of pre-routed clock trees against Synopsys ICC2. Managed to scale up physical design strategy to 100k-LUT FPGAs
- 10/29/2020: Released alpha versions of 2x2 and 12x12 FPGA GDS for Skywater 130nm tape-out run
- 11/05/2020: Supported multi-region memory banks and configuration frames
- 11/05/2020: Upgraded SOFA architecture to satisfy Caravel SoC's I/O interface requirements
- 11/12/2020: Merged the pull request about post-routing synchronization to VPR master branch
- 11/12/2020: Supported defining global ports from VPR's physical tile description
- 11/19/2020: SOFA standard-cell version passed post-PnR functional verification
- 12/10/2020: Supported I/O grids in the center part of FPGA fabrics
- 12/17/2020: Merged QuickLogic's pull request about Yosys upgrade to OpenFPGA master branch
- 12/17/2020: Released the SOFA custom-cell version which passed post-PnR functional verification
- 01/21/2021: Released a preliminary support on LUTRAM in the OpenFPGA framework
- 01/28/2021: Added version display support to the OpenFPGA framework
- 02/04/2021: Patches on the routing resource graph merged to upstream VPR
- 02/11/2021: Generated a proof-of-concept FPGA layout using the naive OpenROAD flow
- 02/11/2021: Fixed DRC errors in the final checking of Skywater tape-out run and resubmitted GDSII files
- 02/11/2021: Landed soft adder support to enhance EDA supports for QuickLogic's architectures
- 02/11/2021: Landed custom LUT circuit model support
- 02/11/2021: Added three tutorial videos to the online documentation
- 02/18/2021: Finished multi-clock support in the OpenFPGA framework
- 02/18/2021: Supported bitstream overloading through .eblif netlists
- 02/25/2021: QuickLogic contributed 20 benchmarks for OpenFPGA's long-term support on Quick- logic's FPGA architectures
- 03/04/2021: Finished a dice-cell (12T-SRAM) layout using the Skywater 130nm PDK
- 03/04/2021: Sent package requests to MOSIS for the Intel 22FFL MPW project
- 03/04/2021: Patches on graphic user's interface merged to upstream VPR

- 03/04/2021: Supported ‘default nettype in the Verilog generator of OpenFPGA
- 03/04/2021: Finished documentation regarding custom cells used in Skywater tape-outs
- 03/11/2021: Sent GF 12nm test chips to Integra for their package assembly
- 04/15/2021: Submitted a paper to RADECS 2021
- 04/22/2021: Integrated IWLS’2005 benchmarks to OpenFPGA
- 04/29/2021: Managed to power-up Globalfoundries 12nm test chip
- 04/29/2021: Finished a 16×16 FPGA fabric using OpenROAD
- 05/06/2021: Supported multi-mode flip-flop design in OpenFPGA
- 05/13/2021: Finished the tutorial about how to use the user template.v
- 05/20/2021: Started AES algorithm integration to OpenFPGA
- 05/27/2021: Merged the first pull request on routing resource graph refactoring to VTR master branch
- 05/27/2021: Implemented the blinker application on the GlobalFoundries 12nm test chip
- 06/03/2021: Merged the second pull request on routing resource graph refactoring to VTR master branch
- 06/10/2021: Reworked testbench generator to accept .bit file
- 06/10/2021: FPL’2021 demo paper accepted
- 06/10/2021: Support default net type in testbench generator
- 06/10/2021: Finished preliminary support on the pre-routed clock network for the physical design of heterogeneous FPGAs
- 06/10/2021: Released a demo video about the GlobalFoundries 12nm test chip
- 06/16/2021: Released an YouTube tutorial video about how to implement a custom user defined template.v file
- 07/07/2021: Submitted the GDS of SOFA Plus to efabless MPW2 shuttle
- 07/12/2021: Started VexRISCV implementation benchmarks on SOFA architecture
- 07/18/2021: Merged pull requests on routing resource graph refactoring to VTR master branch
- 08/16/2021: Created FPGA GDS-ready macro database for fast SoC developments
- 09/08/2021: Submitted a paper to ISSCC 2022 conference
- 09/13/2021: Finished the online user FAQ documentation page
- 09/23/2021: Explored the open-source SpyDrNet project for restructuring physical placement
- 10/07/2021: Filmed and edited second iteration of the Standard Cell Library Tutorial
- 10/14/2021: Carry chain working in VPR with simple benchmark (16-bit adder)
- 10/28/2021: Launched SpyDrNet-Physical repository to maintain a SpyDrNet extension
- 11/04/2021: Early *Programing Management Unit* (PMU) architecture proposal

- 11/11/2021: Submitted two papers to the CICC'22 conference on RRAM and FROG circuits
- 12/16/2021: Prepared RRAM test chip for MPW-4 eFabless shuttle submission
- 01/06/2022: TCASII journal submission of our programmable local clock filtering for rad-hard FPGAs
- 01/27/2022: First CTS automation algorithm implemented and verified through the ICC2 flow to emulate the Pareto Pruning phase
- 03/09/2022: Development of SpyDrNet-Physical with documentation [3]
- 04/20/2022: CTS version 2.0 integrated in the place and route flow
- 04/28/2022: Testing and development of functional verification approach for post restructured netlist
- 05/18/2022: Soft-core exploration platform restructured, to be a user-friendly Python API
- 05/31/2022: SiliconCompiler updated to implement a simple design in the eFabless Caravel wrapper
- 06/02/2022: Functional RTL and verification tests for the PMU version 3.0
- 06/09/2022: Deadline for Skywater 130nm MPW-6 eFabless tape-out shuttle

3 TECHNICAL ACHIEVEMENTS: METHODS, PROCEDURES AND RESULTS

3.1 T-1: Develop XML-based Gate-level Architecture Specification

The XML syntax is an extension of the original FPGA architecture description language supported by VTR, with (1) enriched gate-level design parameters for primitive blocks, such as LUTs, FFs and multiplexers; (2) architecture parameters to link the Verilog modules to defined gate-level design parameters and model different configuration circuitry, such as memory access decoders and scan-chains. The parser for all the XML syntax has been implemented in the VTR engine. The gate-level circuit design parameters of an FPGA module are defined under an XML property called circuit model, reusing the syntax from FPGA-SPICE to avoid duplication in XML files. In general, a circuit_model includes the following properties:

```
<circuit model type="<string>" name="<string>" prefix="<string>"
is default="<int>" verilog netlist="<string>" dump structural verilog="<string>"/>
<design technology type="<string>"/>
<input buffer exist="<string>" circuit model name="<string>"/>
<output buffer exist="<string>" circuit model name="<string>"/>
<pass gate logic circuit model name="<string>"
<port type=""<string>" prefix="<string>" size="<int>" is global="<string>"
circuit model name="<string>"/>
      </circuit model>
```

The rest of this section is devoted to explaining the functionality and motivation of the keywords. Note that the keywords in input_buffer, output_buffer, pass_gate_logic and port are general for all the Verilog modules, while the keywords in design_technology are typically exclusive to a specific type of Verilog modules. Keywords that define the Verilog module are explained in Table 1.

Table 1: Description of XML Syntax Defining a primitive Block

XML Syntax	Description
type	This keyword aims at defining the functionality of a Verilog module and is used to identify if special ports/structures are required when auto-generating the Verilog netlist of the module. The available options are inv buf, pass gate, sram, wire, chan wire mux, lut, ff, sff, hard logic and iopad, corresponding to inverter/buffer, pass-transistor/transmission-gate, SRAM, routing wire segments, LUT, multiplexer, FFs, SCFF, I/O pads, and other hard logics in FPGAs such as adders and heterogeneous blocks. We believe these types have covered all the possible primitive blocks existing in modern FPGA architectures.
name	To create a unique identifier in linking the circuit module to primitive block in the original architecture description, this keyword represents the unique name of each Verilog module that appear in its definition.
prefix	To help instancing/indexing the circuit modules in Verilog netlists, this keyword defines the prefix of each Verilog instance in all the netlists. For example, when instancing a Verilog module, it is names as <prefix> <int> .
is default	FPGAs consist of a considerable amount of duplicated circuits. It is very likely that more than one primitive blocks use the same Verilog module. This keyword aims at reducing the complexity (avoid duplicated lines) of XML file. When the keyword is enabled, the Verilog module will be considered as the default choice for the primitive blocks in the FPGA architectures.
verilog netlist	The keyword is used to specify a user-defined Verilog netlist for a primitive block. When the property is enabled, the Verilog netlist for this module will not be auto-generated. This is an interesting feature to model optimized FPGA components, such as LUTs and DSP blocks, whose performances are highly dependent on the technology and the circuit structure. This brings the capability to study the system-level impact of the circuit elementary blocks, thereby enabling interesting circuit/architecture co-optimization opportunities.
dump struct-ural verilog	When the value of this keyword is set to be true, the Verilog generator will output gate-level netlists of this module, instead of its behavior-level. Gate-level netlists enable custom layout-level optimization while behavior-level is more suitable for high-speed formal verification and easier debugging with HDL simulators.

3.1.1 Input and Output Inverters/Buffers, Pass-transistors/Transmission Gates

We have created two XML properties input buffer and output buffer for specifying the existence of inverters and buffers at input and output ports of a Verilog module. We have also created an XML keyword pass gate logic for specifying the pass-transistors/transmission gates used in a Verilog module, such as multiplexer and LUT. The XML keyword circuit model name specifies the inverter or buffer to be used at input and output ports. The circuit model name must be a valid name when defining a Verilog module (See Table 1).

3.1.2 Ports Definition

Keywords that define the port list of a Verilog module are explained in Table 2. Note that the ports defined under circuit model are critical in two cases: (1) linking a Verilog module to a primitive block in the FPGA architecture (the port map should match with that of a primitive block) and (2) instancing a user-defined Verilog netlists (the port map should match with the module declaration given by the user Verilog netlist).

Table 2: Description of XML Syntax Defining Ports of a Primitive Block

XML Syntax	Description
type	This property aims at defining the functionality of a port of a Verilog module. The available options are input, output, inout, clock and sram. In addition to the regular ports, such as input, output, inout and clock, we support the configuration ports that are connected to configuration SRAM modules, which are ignored in the original VPR architecture description language.
prefix	This property defines the prefix of the port when the Verilog module is instanced. For example, when instancing the Verilog module, this port will be named as <prefix> <int>
size	This property defines the width of a port.
is global	The property is used to specify if the port is connected to a global signal, such as clock, set and reset.
circuit model name	The property is used to specify the SRAM or SCFF Verilog modules that a configuration port will be connected to.

3.1.3 Primitive Blocks Description

First, we provide XML examples for the supported primitive blocks in Table 3, Table 5, Table 6 and Table 8, in order to demonstrate the capability of XML syntax in describing all the modules of a homogeneous FPGA architectures. As special syntax (highlighted in bold) have been added under the XML node design technology in some primitive blocks, in order to support their diverse circuit variations. Table 3 lists several examples of basic elements that are frequently used in primitive blocks. To support versatile circuit topologies, we introduce a few special syntax (highlighted with bold text) associated with those examples.

3.1.3.1 Inverters and Buffers

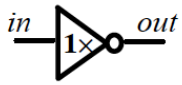
To support the various drive strengths that may exist in the FPGA architectures, we have created syntax that are exclusively for inverters and buffers. For example, to drive long metal wires across CLBs, buffers with large drive strength have to be used. Table 4 describes the special syntax that corresponds to the examples in Table 3.

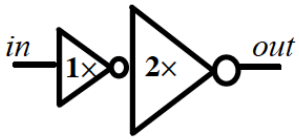
3.1.3.2 Pass-gate Logic

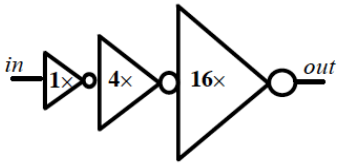
To support different pass-gate/transmission gates logic structures as used in FPGAs (e.g., Xilinx FPGAs typically employ transmission-gates while Altera FPGAs use pass-transistors), we provide a syntax to specify the type of pass-gate logic. Therefore, in a keyword topology, we allow the user to specify the type of pass logic, which can be either pass-transistor or transmission-gate, as exemplified in Table 3.

Table 3: Examples of Basic Elements and associated XML Description

Primitive Blocks	XML Description

	`<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>`

	`<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>`

	`<circuit_model type="inv_buf" name="tap_buf4"`
	`prefix="tap_buf4">`
	`  <design_technology type="cmos" topology="buffer"`
	`size="1" tapered="on" tap_buf_level="3" f_per_stage="4"/>`
	`  <port type="input" prefix="in" size="1"/>`
	`  <port type="output" prefix="out" size="1"/>`
	`</circuit_model>`

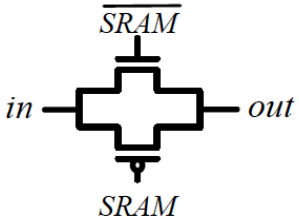
	`<circuit_model type="pass_gate" name="tgate"`
	`prefix="tgate">`
	`  <design_technology type="cmos"`
	`  topology="transmission_gate"/>`
	`  <port type="input" prefix="in" size="1"/>`
	`  <port type="input" prefix="sram" size="1"/>`
	`  <port type="input" prefix="sramb" size="1"/>`
	`  <port type="output"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

Table 4: Description of XML Syntax for Inverters/Buffers/Pass-gate Logic

Special Syntax for Inverters	
XML Syntax	**Description**
topology	The value of this keyword can be `inverter` or `buffer`, specifying the type of this component to be either an inverter or a buffer.
size	Specify the drive strength of inverter/buffer. **For an inverter**, the size denotes the drive. **For a buffer**, the size is the drive of the inverter at the second level. **For a tapered (multi-level) buffer**, the size is the drive of the inverter at the first level.
tapered	Specify if this is a tapered buffer, which is only valid when the topology is set to `buffer`.
Tap_buf_level	Specify the number of levels in a tapered buffer. For example, in Table 3, we show a tapered buffer with three levels.
F_per_stage	Specify the factor of driving efforts in each level of a tapered buffer. For example, in Table 3, we show a tapered buffer where the drive of each stage is a factor of four to its previous stage.

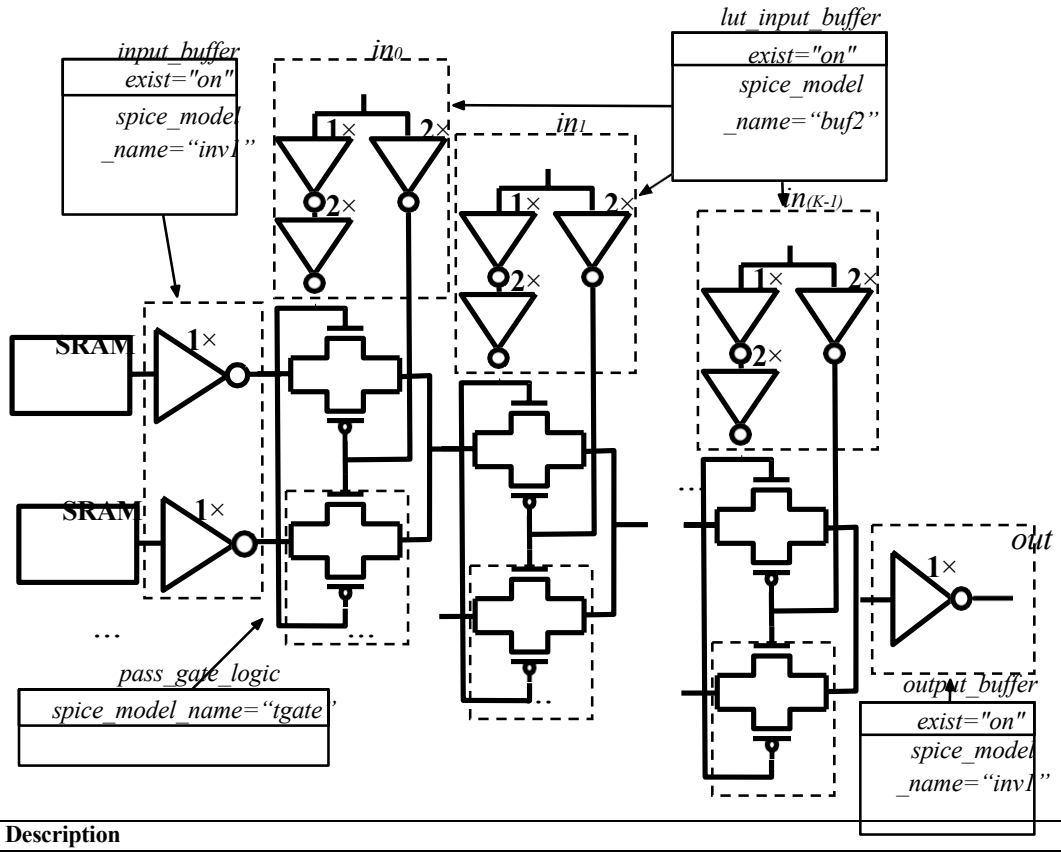
3.1.3.3 LUT

In Table 5, we show an example of a LUT circuit and its associated XML description as supported by our Verilog generator. To support various LUT structures, we introduce the special syntax `lut input buffer`. LUT has two types of inputs, one is the regular inputs, the other is the configuration memory port (consider it as special inputs). The input buffer definition is devoted to specify the existence of buffers between the LUT memory port and the actual configuration SRAMs/SCFFs. To support the customization of the regular input buffering, a special keyword `lut input buffer` is created (as used in Table 5), where users can specify the

buffers to be used for the regular inputs.

Table 5: Examples of Primitive Blocks and associated XML Description

6-input Look-Up Table (LUT) based on transmission gate



XML Description

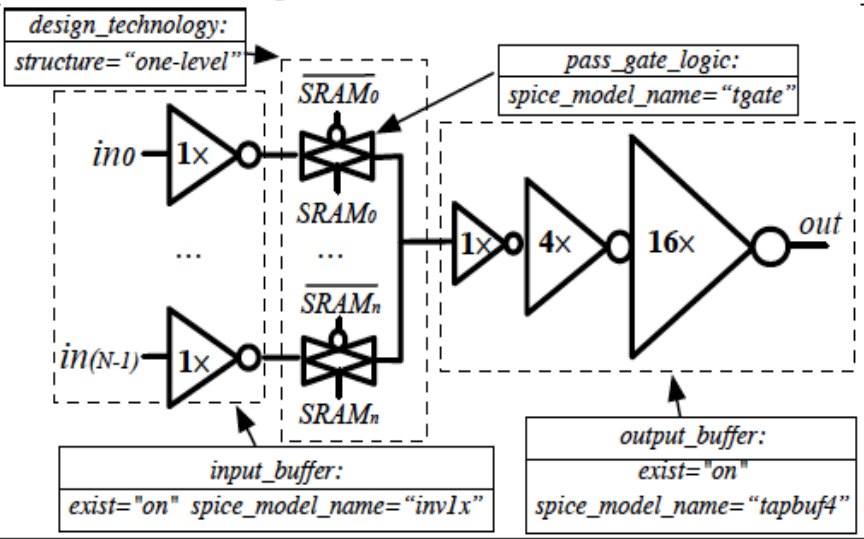
```
<circuit model type="lut" name="lut6" prefix="lut6">
  <input buffer exist="on" circuit model="inv1x"/>
   <output buffer exist="on" circuit model name="inv1x"/>
   <lut input buffer exist="on" circuit model name="buf2"/>
   <pass gate logic circuit model name="tgate"/>
   <port type="input" prefix="in" size="6"/>
   <port type="output" prefix="out" size="1"/>
   <port type="sram" prefix="sram" size="64"/>
</circuit model>
```

3.1.3.4 Multiplexer

To achieve best area, delay and power, different structures of multiplexers are traditionally used in a single FPGA, depending on their input size, fan-out and loads to drive. For example, a one-level structure is preferred in small fan-in conditions, while a multi-level structure guarantees the best overall perfor- mance in large fan-in conditions. In Table 6, we show an example of a multiplexer and its associated XML description. Using the keywords structure and num level, presented in Table 7, users can describe any multiplexing structure.

Table 6: Examples of a Multiplexer and Associated XML Description

4-input Multiplexer based on transmission gate



XML Description

```
<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>
```

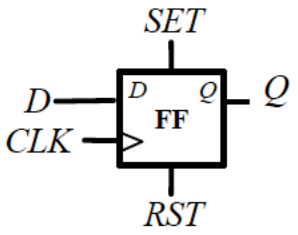
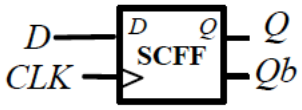
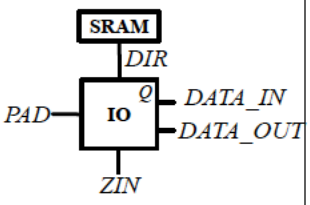
Table 7: Description of XML Syntax for a Multiplexer Block

XML Syntax	Description
structure	The value of this keyword can be one-level, multi-level and tree-like, covering all the possible multiplexer structures in FPGAs.
num level	This keyword aims at specifying the number of levels in a multi-level multiplexer.

3.1.3.5 User-defined Primitive Blocks

In Table 8, we show examples of user-defined primitive blocks, flip-flops, scan-chain flip-flops and I/O pads.

Table 8: Examples of User-defined Primitive Blocks and associated XML Description

Primitive Blocks	XML Description
	<circuit_model type="ff" name="dff" prefix="dff" verilog_netlist="ff.v"> <port type="input" prefix="D" size="1"/> <port type="input" prefix="Set" size="1" is_global="true"/> <port type="input" prefix="Reset" size="1" is_global="true"/> <port type="output" prefix="Q" size="1"/> <port type="clock" prefix="clk" size="1" is_global="true"/> </circuit_model>
	<circuit_model type="sff" name="scff" prefix="scff" verilog_netlist="scff.v"> <port type="input" prefix="D" size="1"/> <port type="output" prefix="Q" size="2"/> <port type="clock" prefix="clk" size="1" is_global="true"/> </circuit_model>
	<circuit_model type="iopad" name="iopad" prefix="iopad" verilog_netlist="io.v"> <port type="inout" prefix="pad" size="1"/> <port type="sram" prefix="dir" size="1" circuit_model_name="scff"/> <port type="input" prefix="data_in" size="1"/> <port type="input" prefix="zin" size="1" is_global="true"/> <port type="output" prefix="data_out" size="1"/> </circuit_model>

3.1.1 Linking Primitive Blocks to the FPGA Architecture

To link a primitive block in the architecture description with its Verilog modules, we create an additional XML property `circuit model name`. The XML property `circuit model name` can be added to not only primitive blocks, such as LUT and FF, but also interconnections elements, such as *Switch Block* (SB), Connection *Block* (CB) and local routing architecture of CLBs. Additionally, to specify the type of configuration circuitry, we create XML syntax `verilog` and `organization` under the XML node sram. Fig. 2 shows an illustrative example, where a compact homogeneous FPGA architecture is linked to the Verilog modules described in Table 5, Table 6 and Table 8.

```
<sram>
   <verilog organization="scan-chain" circuit model name="scff"/>
</sram>
<switch type="mux" name="cb mux" circuit model name="mux 2level"/>
<switch type="mux" name="sb mux" circuit model name="mux 1level"/>
<pb type name="clb">
   <pb type name="ble">
      <pb type name="lut" circuit model name="lut6">
      <pb type name="ff" circuit model name="dff">
      <interconnect>
         <mux input="lut.out ff.Q" output="ble.out" circuit model name="mux 1level">
      </interconnect>
   </pb type>
   <interconnect>
      <mux input="ble.out clb.in" output="ble.in" circuit model name="mux 2level">
   </interconnect>
</pb type>
```

Figure 2: Compact Homogeneous FPGA Architecture Links to the Verilog Modules

3.1.4 Multi-Mode CLB Architectures Support

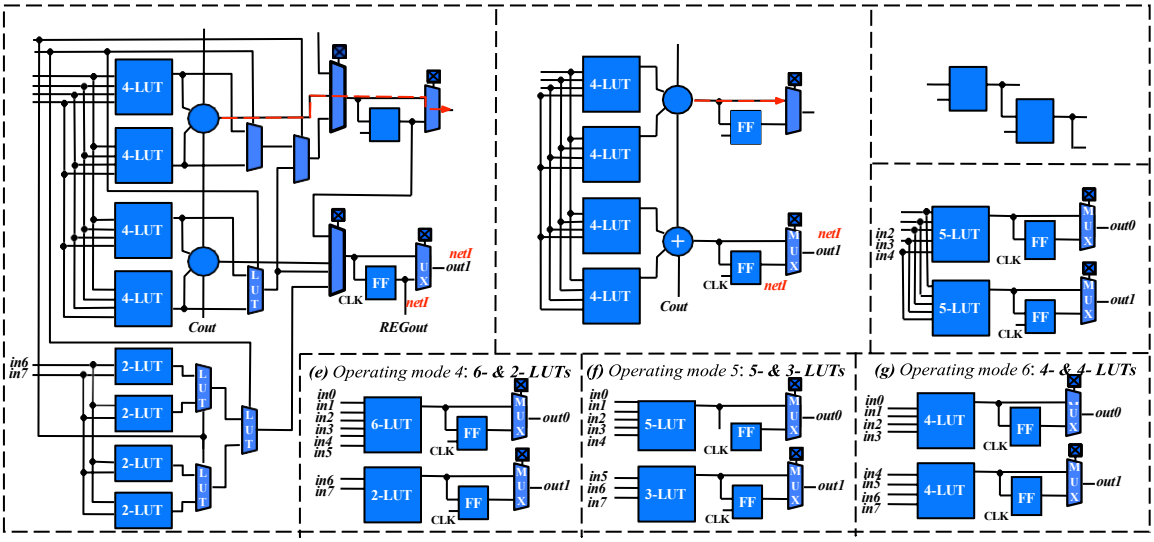


Figure 3: (a) Physical Implementation of a BLE and (b)(c)(d)(e)(f)(g) its Operating Modes with Back Annotation of Routing Nets

To support modern FPGA features, we have further enriched the VPR XML syntax in order to support multi-mode CLB architectures. An illustration of the "main" modes used in commercial FPGAs and supported in our tool are shown in Fig. 3. Our XML parser and Verilog generation engine have been upgraded to support this novel syntax. Several critical bugs in VPR related to the multi-mode support (badly routed CLBs, missing back-annotations after routing, etc.) have also been fixed. Those bugs exist in all versions of VPR and possibly also impact the team from Princeton. We informed them about these bugs to upstream the fixes to VPR as part of the modularization effort.

3.1.4.1 Circuit-level Modeling to Support Fracturable LUTs

Fracturable LUTs as shown in Fig. 3(a) have become an ubiquitous element in modern FPGA architectures. To capture state-of-art FPGA structures, we have introduced a novel XML syntax to support the fracturable LUT circuits. Fig. 4 shows an example how the circuit designs of a fracturable LUT can be described and customized in the extended XML language. We offer a high description flexibility with the properties `structure`, highlighted in Fig. 4. For example, users can specify which inputs are disabled during a fracturable mode using the XML property `tri state` map. The levels and po- sitions of fracturable outputs can be freely defined through the XML properties `lut frac level` and `lut output` mask. To support mode switching of fracturable LUTs, the port map includes a special port mode rather than the regular configuration port. Moreover, FPGA-Verilog can still include user-defined Verilog netlists through the XML property `verilog netlist`.

3.1.4.2 Buffer Insertion in LUT

Commercial FPGAs typically add buffers to intermediate stages of 6-input or larger LUTs, as illus- trated in Fig. 5, in order to achieve best area-delay product. To support LUT customization, we have defined a specific XML tag and upgraded the Verilog generator. Fig. 6 shows an example where buffers (described in a `circuit` model called buf1) are added to the second and the fourth stage of a fracturable 6-input LUT. Using the new syntax location map, the buffer addition is fully customizable to any stage of any LUT.

```
<!-- Fracturable LUT modeled in XML -->
<circuit model type="lut" name="frac lut6" verilog netlist="lut.v">
  <design technology fracturable lut="true"/>
  <pass gate logic circuit model name="tgate"/>
  <port type="input" prefix="lut in" size="6" tri state map="----11" circuit model
  name="OR2"/>
  <port type="output" prefix="lut4 out" size="4" lut frac level="4" lut output
  mask="0,1,2,3"/>
  <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 name="scff" default val="1"/>
  <port type="sram" prefix="mode" size="2" mode select="true" circuit model
name="scff" default val="1"/>
</circuit model>
```

Figure 4: Examples of Extended XML Syntax for LUTs

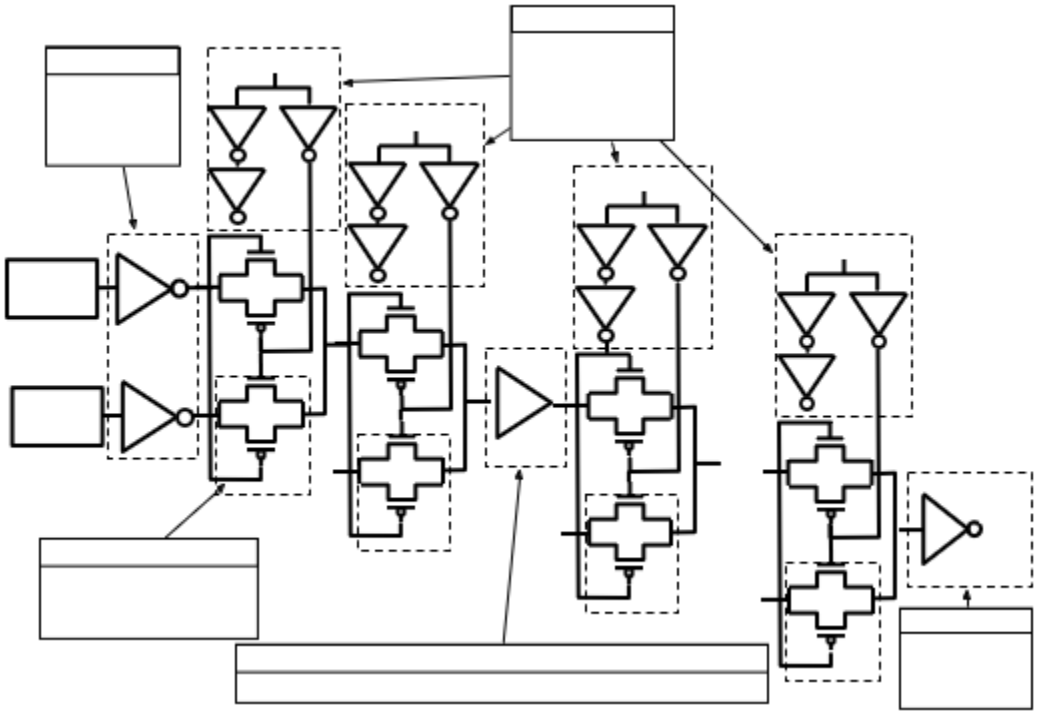


Figure 5: An Illustrative Example of Buffer Addition to Intermediate Stages of a LUT

```
<!-- Fracturable LUT with intermediate buffers -->
<circuit model type="lut" name="frac lut6" verilog netlist="lut.v">
  <design technology fracturable lut="true"/>
  <pass gate logic circuit model name="tgate"/>
  <port type="input" prefix="lut in" size="6" tri state map="----11" circuit model
  name="OR2"/>
  <port type="output" prefix="lut4 out" size="4" lut frac level="4" lut output
  mask="0,1,2,3"/>
  <lut intermediate buffer exist="on" circuit model name="buf1" location map="-1-1-"/>
  <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 name="scff" default val="1"/>
  <port type="sram" prefix="mode" size="2" mode select="true" circuit model
name="scff" default val="1"/>
</circuit model>
```

Figure 6: Examples of Buffer Addition to Intermediate Stages of a fracturable 6-input LUT

3.1.4.3 Physical BLE Modeling

The VTR XML language focus on compact description of BLE architectures instead of the detailed schematic-level representation. For instance, a complex multi-mode BLE (see Fig. 3-a) is modeled by multiple abstract operating modes (see Fig. 3b-g). This abstraction indeed simplifies the CAD algorithms to map to FPGA resources but hides important details required for the bitstream generation to the physical BLEs. To precisely describe the physical BLEs while keeping the benefits of the existing representation, we extended the XML syntax to (1) model the physical implementation of BLEs and (2) link the components in the various operating modes to the physical modules. To provide an intuitive illustration, we take the example of the multi-mode CLB shown in Fig. 3(a)(b) and present its XML description in Fig. 7. The physical implementation of a BLE is defined under the root of its XML node, the pb `type` called `fle`. The description of

the physical BLE follows the same style as the original XML language. Since the physical BLE may be unpackable due to limited synthesis support for fracturable LUTs, its visibility during packing can be turned off using a new XML property `disabled in packing` to keep the golden packing runtime and results. Under the physical mode, users can link primitive blocks to circuit implementations using a XML property `circuit` model name. Fig. 7 shows how a fracturable LUT is linked to a defined circuit model in Fig. 4. Under the operating modes, each virtual pb `type` has to be linked to its physical implementation through several XML properties `physical` pb type name and `physical` mode `pin`. Our XML syntax mode `bits` allows users to customize the configuration bits applied to fracturable LUTs in any operating mode. As such, without modifying neither the packing nor the synthesis engines, our XML syntax can map the configuration bits from any operating mode to its physical implementation.

```
<!-- Multi-mode BLE -->
<pb type name="ble" num pb="10" physical mode name="ble phy"/>
<!-- Physical implementation of BLE shown in Fig. 3(a) -->
  <mode name="ble phy" disabled in packing="true"/>
    <pb type name="flut6 phy" circuit model name="frac lut6">
      <input name="in" num pins="6"/>
      <output name="lut4 out" num pins="4"/>
      <output name="lut5 out" num pins="2"/>
      <output name="lut6 out" num pins="1"/>
    </pb type>
    <pb type name="lut4 phy" circuit model name="lut4">
      <input name="in" num pins="4"/>
      <output name="out" num pins="1"/>
    </pb type>
    <pb type name="adder phy" num pb="2" circuit model name="adder"/>
    <pb type name="ff phy" num pb="2" circuit model name="dff"/>
    <interconnect>
<!-- Routing multiplexers are omitted-->
    </interconnect>
  </mode>
<!-- Arithmetic mode of BLE shown in Fig. 3(b)-->
  <mode name="flut4 arithmetic"/>
    <pb type name="flut4 arith" num pb="2"/>
      <pb type name="lut4" mode bits="01" physical pb type name="lut4">
        <input name="in" num pins="4" physical mode pin="in[3:0]"/>
        <output name="out" num pins="1" physical mode pin="lut4 out"/>
      </pb type>
     <pb type name="adder" num pb="1" physical pb type name="adder phy">
      <input name="a" num pins="1" physical mode pin="a"/>
      <input name="b" num pins="1" physical mode pin="b"/>
      <input name="cin" num pins="1" physical mode pin="cin"/>
      <output name="cout" num pins="1" physical mode pin="cout"/>
      <output name="sumout" num pins="1" physical mode pin="sumout"/>
     </pb type>
     <pb type name="ff" num pb="1" physical pb type name="ff phy">
      <input name="D" num pins="1" physical mode pin="D"/>
      <output name="Q" num pins="1" physical mode pin="Q"/>
      <clock name="clk" num pins="1" physical mode pin="clk"/>
     </pb type>
      <interconnect>
<!-- Routing multiplexers are omitted -->
      </interconnect>
   </pb type>
  </mode>
<!-- More operating modes can be defined -->
</pb type>
```

Figure 7: Examples of Extended XML Syntax for a BLE

3.2 T-2: Develop Auto-generation Structural Verilog Netlists Engine

The core engine of our Verilog generator, as illustrated in Fig. 8, consists of two parts:

1. Verilog generation for primitive blocks, i.e., LUTs and multiplexers and their basic elements, i.e., inverters, buffers and transmission gates. (See details in Section 3.2.1.)
2. Verilog generation for high-level blocks, i.e., CLBs, SBs and CBs, and top-level block, i.e., full FPGA fabric. (See details in Section 3.2.2, Section 3.2.3 and Section 3.2.4 respectively.)

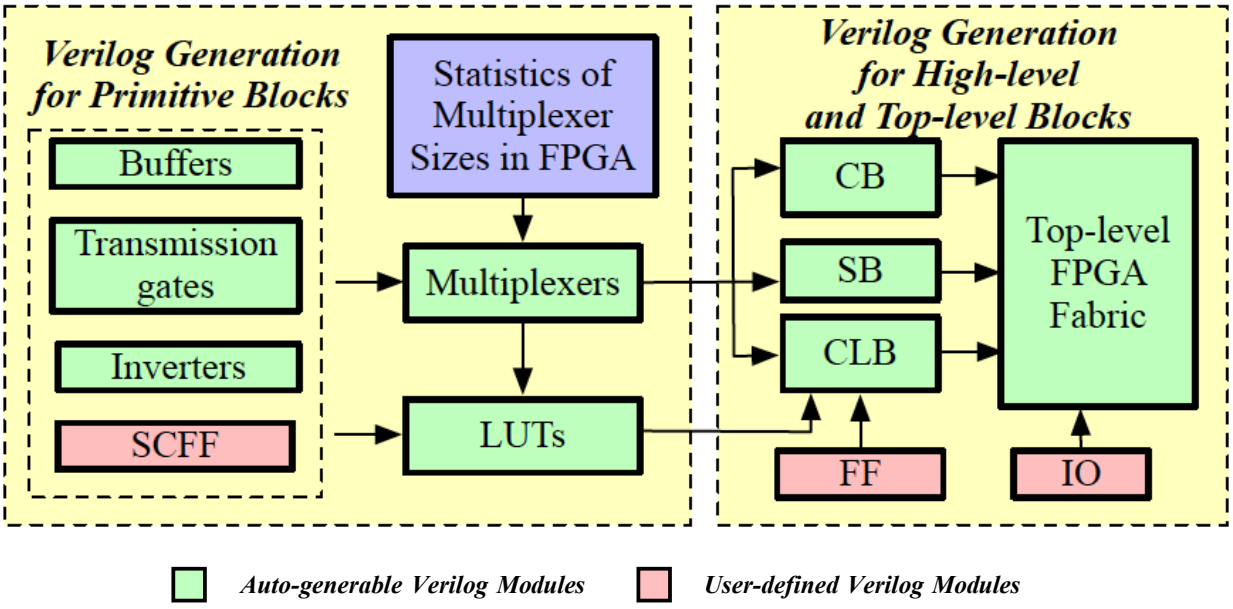


Figure 8: Verilog Auto-Generation Engine

```
// Structural Verilog for CMOS MUX basis module:  mux 1level size2 basis
module mux 1level size2 basis (
      input [0:1] in,
      output out,
      input [0:0] mem,
      input [0:0] mem inv);
   // Structure-level description
   TGATE TGATE 0 (in[0], mem[0], mem inv[0], out);
   TGATE TGATE 1 (in[1], mem inv[0], mem[0], out);
endmodule
// END Structural Verilog CMOS MUX basis module:  mux 1level size2 basis
// CMOS MUX info:  circuit model name=mux 1level, size=2, structure=one-level
module mux 1level size2 (
      input wire [0:1] in,
      output wire out,
      input wire [0:0] sram,
      input wire [0:0] sram inv);
   wire [0:1] mux2 l1 in;
   wire [0:0] mux2 l0 in;
   mux 1level size2 basis mux basis (mux2 l1 in[0:1], mux2 l0 in[0],
      sram[0:0], sram inv[0:0]);
   inv1x inv0 (in[0], mux2 l1 in[0]);
   inv1x inv1 (in[1], mux2 l1 in[1]);
   inv1x inv out (mux2 l0 in[0], out);
endmodule
```

Figure 9: Example of Auto-generated Verilog Netlists: Al1-level 2-input Multiplexer

3.1.5 Verilog Generation for Primitive Blocks

Primitive blocks are the fundamental parts of other high-level and top-level blocks. As FPGA architectures are highly parameterized, the size of LUTs and multiplexers are strongly dependent on the architecture descriptions. In particular, the input sizes of multiplexers in SBs

and CBs are implicitly determined by considering the length of wire segments, connectivity parameters and routing channel width, instead of user-specified in architecture description. Therefore, it is critical to develop Verilog auto-generation techniques especially for multiplexers. As an illustration, we show in Fig. 9 an example of auto-generated Verilog codes describing a 2-input multiplexer. Our Verilog auto-generation for primitive blocks consists of four steps:

1. Verilog generation of the basic elements, i.e., inverters, buffers and pass-gate logics, using the description syntax shown in Table 3. By naming the basic circuit elements with those available in a standard cell library, the generated Verilog netlists can be automatically mapped to the physical gates and used as is in a semi-custom design tool. For example, the inv1x in Fig. 9 can be an inverter from standard cells.

2. Statistical count of the input sizes for the multiplexers used in the FPGA architecture. To avoid duplication of primitive blocks, our Verilog generator first creates a list of different input sizes of multiplexer that are employed in the FPGA architecture, by traversing the routing resource graph of VPR.

3. Verilog generation of basic multiplexers. Using the results from the second step, Verilog modules of multiplexer with different input sizes are created. The internal structure of multiplexers is generated by considering the description in Table 6. Note that the multiplexers of LUTs are also generated in this step.

4. Verilog generation of LUTs, using multiplexers, inverters, and buffers.

3.1.6 Verilog Generation for CLBs

To support the hierarchy of the CLB architecture definition as shown in Section 3.1.4, we developed a recursive *Depth-First Search* (DFS) algorithm whose pseudo code is shown in Algorithm 1. The algorithm can first recursively visit of all the child pb type under current node. When a primitive pb type is reached, a Verilog module for the primitive node is outputted. Otherwise, it outputs a Verilog module containing Verilog modules of all the child pb type and multiplexers interconnecting the child nodes. As such, our algorithm can support any CLB architecture that can be described by VPR. Fig. 10 shows an example of Verilog codes describing a BLE containing a LUT, a FF and a multiplexer.

```
Function rec_gen_verilog_pb_type(current pb type):
    foreach child pb type ∈ current pb type do
        rec_gen_verilog_pb_type(child pb type);
    end
    if (TRUE == is_primitive_pb_type(current pb type)) then
        gen_verilog_primitive_block(current pb type);
        return;
    gen_verilog_pb_type_declaration(current pb type);
    foreach child pb type ∈ current pb type do
        call_verilog_child_pb_type(child pb type);
    end
    gen_verilog_mux(current pb type);
end
```

Algorithm 1: Verilog generation engine for CLB (pseudo-code)

3.2.1 Verilog Generation for SBs and CBs

The Verilog modules are generated by referring to the connectivity in *Routing Resource Graph* (rr graph) in VPR. Algorithm 2 depicts the engine to generate the Verilog modules of the CBs. We enumerate all the coordinates of CBs in VPR and identify if each node in the **rr** graph belongs to the CBs. For each node in the CBs, we evaluate if it corresponds to a multiplexer (and consider its fan-in). Then, our engine can generate a wire or a multiplexer according to the evaluation results. The Verilog generation for the SBs share the sample principle as Algorithm 2. Fig. 11 shows an example of Verilog codes describing a CB, where we only show a multiplexer to keep the simplicity of illustration.

```
//---- An example of a BLE module -----
module ble6 0 mode ble6 (
     input [0:0] prog_clk, input [0:0] pReset, input [0:0] clk,
     input [0:0] Reset, input [0:0] Set,
     input wire mode ble6_in 0 , input wire mode ble6_in 1 ,
     input wire mode ble6_in 2 , input wire mode ble6_in 3 ,
     input wire mode ble6_in 4 , input wire mode ble6_in 5 ,
     input wire mode ble6_out 0 , input wire mode ble6_clk 0 ,
     input [9116:9180] scff in, input [9116:9180] scff out);
  ble6 0 mode ble6 lut6 0 lut6 0 (prog clk[0:0], pReset[0:0], clk[0:0], Reset[0:0],
lut6 0__in 0 , lut6 0__in 1 , lut6 0__in 2 , lut6 0__in 3 , lut6 0__in 4 , lut6 0__in 5 ,
lut6 0_out 0 , scff in[9116:9179], scff out[9116:9179]);
  dff ff 0 (Set[0:0],Reset[0:0],clk[0:0],ff 0__D 0 , ff 0__Q 0 );
  wire [0:1] in bus mux 1level size2 0 ;
  assign in bus mux 1level size2 0 [0] = ff 0__Q 0 ;
  assign in bus mux 1level size2 0 [1] = lut6 0_out 0 ;
  wire [9180:9180] mux 1level size2 0 scff outb ;
  mux 1level size2 mux 0 ( in bus mux 1level size2 0 , mode ble6_out 0 ,
scff out[9180:9180], mux 1level size2 0 scff outb[9180:9180]);
  direct interc direct interc 0 (mode ble6__in 0 , lut6 0__in 0 );
  direct interc direct interc 1 (mode ble6__in 1 , lut6 0__in 1 );
  direct interc direct interc 2 (mode ble6__in 2 , lut6 0__in 2 );
  direct interc direct interc 3 (mode ble6__in 3 , lut6 0__in 3 );
  direct interc direct interc 4 (mode ble6__in 4 , lut6 0__in 4 );
  direct interc direct interc 5 (mode ble6__in 5 , lut6 0__in 5 );
  direct interc direct interc 6 (lut6 0__out 0 , ff 0__D 0 );
  direct interc direct interc 7 (mode ble6_clk 0 , ff 0_clk 0 ));
endmodule
```

Figure 10: Example of Auto-generated Verilog Netlists - a BLE Containing a LUT, a FF and a Multiplexer

```
rr graph: VPR Routing Resource Graph.
fpga size: Array size of a FPGA architecture.
Function gen verilog cb(rr graph:):
    foreach x ∈fpga size do
        foreach y ∈fpga size do
            foreach node ∈rr graph: do
                if (FALSE == is rr node in cb(node)) then
                    continue;
                if (is rr node mux(node)) then
                    gen verilog mux(node);
                else
                    gen verilog wire(node);
                end
            end
        end
    end
end
```

Algorithm 2: Verilog generation engine for CB (pseudo-code).

```
//---- An example of a CB module -----
module cbx 1 10 (
    input [0:0] prog clk, input [0:0] pReset,
    input [0:0] clk, input [0:0] Reset,
    input chanx 1 10 midout 0 , input chanx 1 10 midout 1 ,
    input chanx 1 10 midout 2 , input chanx 1 10 midout 3 ,
    output grid 1 11 pin 0 2 0 , output grid 1 11 pin 0 2 2 ,
    output grid 1 11 pin 0 2 4 , output grid 1 11 pin 0 2 6 ,
    output grid 1 11 pin 0 2 8 , output grid 1 11 pin 0 2 10 ,
    output grid 1 11 pin 0 2 12 , output grid 1 11 pin 0 2 14 ,
    output grid 1 10 pin 0 0 0 , output grid 1 10 pin 0 0 4 ,
    output grid 1 10 pin 0 0 8 , output grid 1 10 pin 0 0 12 ,
    output grid 1 10 pin 0 0 16 , output grid 1 10 pin 0 0 20 ,
    output grid 1 10 pin 0 0 24 , output grid 1 10 pin 0 0 28 ,
    output grid 1 10 pin 0 0 32 , output grid 1 10 pin 0 0 36 ,
    input [6776:6793] scff in, input [6776:6793] scff out);
  wire [0:1] mux 1level size2 1 inbus;
  assign mux 1level size2 1 inbus[0] = chanx 1 10 midout 0 ;
  assign mux 1level size2 1 inbus[1] = chanx 1 10 midout 1 ;
  wire [6776:6776] mux 1level size2 1 scff outb;
  mux 1level size2 mux 1 (mux 1level size2 1 inbus, grid 1 11 pin 0 2 0 ,
scff out[6776:6776], mux 1level size2 1 scff outb[6776:6776]);
// Other multiplexers are not show for the simplicity in illustration
endmodule
```

Figure 11: Example of Auto-generated Verilog Netlists: a Connection Block

3.1.7 Verilog Generation for FPGA Fabric

With all the high-level modules, i.e., CLBs, SBs and CBs, are generated, this step encapsulates all the Verilog modules into a top-level netlist. Due to the large size of top-level netlists, example codes have not been included in this report, but can be generated by following the example listed on the front page of our GitHub repository [2].

3.1.8 Verilog and Bitstream Generation for Multi-Mode CLBs

Based on the enriched XML syntax presented in Section 3.1.5, we then upgraded the Verilog and bitstream generators to support multi-mode CLB architectures.

New features added on our Verilog generator:

1. output netlists of fracturable LUTs whose fracture level is customized through XML syntax.
2. output netlists of physical mode of BLEs defined in architecture XML file.

New features of our Bitstream generator:

The bitstream generator has been generalized to support any physical BLEs that can be described by VPR. As the VPR packing engine only maps logic functions into operating modes, we propose an additional packing stage for the physical implementation of BLEs, as depicted in Fig. 12. The packing consists of two steps:

1. back-annotation on physical BLEs, where the packing results of operating modes are mapped to the physical modes. For example, the mapped nets of each primitive blocks highlighted red in Fig. 3(b) can be exactly annotated on the physical modules in Fig. 3(a). Thanks to our proposed XML syntax, the net mapping can be done even though the pb type in operating mode does not exist in the physical mode, unlocking various abstraction of operating modes to achieve better packing results.
2. a detailed routing for physical BLEs, to configure the programmable interconnection. We use the routing engine of the VPR packer to achieve a similar quality of result. As such, whatever the difference in interconnection topology between the operating and physical modes, our approach can guarantee correct mapping results for physical modes. This allows us to generate bitstreams for any complex CLB architectures without losing any generality in the original FPGA description language.

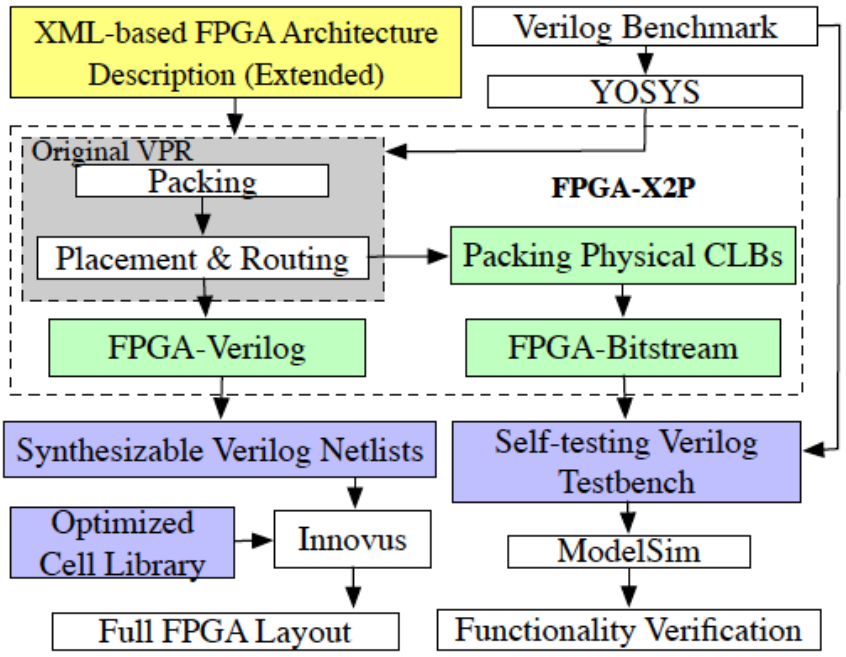


Figure 12: OpenFPGA Framework including XML-to-Prototype and Verilog-to-Bitstream Design Flows

3.2.2 SDC Generation Overview

We have implemented a rich SDC generator in the OpenFPGA framework to deal with both P&R constraints and sign-off timing analysis. Our flow automatically generates two sets of SDC files, as listed in Table 9. 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. The second set of SDC is designed for the timing analysis of a benchmark at the post P&R stage. We have implemented an alpha version of both SDC sets.

Table 9:List of Auto-generated SDC Files

SDC for constraining timing during P&R flow	
File name	Usage
break_loop.sdc	Break the combinational loops in FPGA architecture
clock.sdc	Constrain the clock signals of FPGA architecture
cb.sdc	Pin-to-pin Timing constraints for Connection Blocks that are defined in architecture XML file
sb.sdc	Pin-to-pin Timing constraints for Switch Blocks that are defined in architecture XML file
pb_types.sdc	Pin-to-pin Timing constraints for CLBs that are defined in architecture XML file
routing_channels.sdc	Pin-to-pin Timing constraints for routing wires that are defined in architecture XML file
SDC for timing analysis	
File name	Usage
fpga_top_analysis.sdc	Disable timing for unused resources in implemented FPGAs. The SDC file is used by STA tools, such as Synopsys PrimeTime and Cadence Tempus, to perform timing analysis for a mapped FPGA

3.1.9 SDC Generation for CLBs

VPR uses annotated timing paths to calculate critical paths and optimize the packing. During the Verilog generation, no timing constraints are generated for the CLB. Our SDC file generation aims at using the user-defined values to constrain the CLB, as depicted in Fig. 13. The motivation behind this design choice is that the first use case of this option can be the output of the FLEs doing feedback onto the input of the FLEs. This feedback loop can be implemented as a long wire. Without proper constraining, this wire can be made longer, and its priority in the P&R process can be low even though it is a critical path on our fabric.

To enable timing-driven P&R, the combinational loops in the CLBs should be open. For this purpose, we implemented a new option for multiplexers and Complete interconnection, the loop breaker, as exemplified in Fig. 14. When loop breaker is enabled, our SDC generator adds special buffers to specified ports which may cause combinational loops. By disabling the timing constraints on the special buffers, combinational loops can be avoided during P&R while the timing constraints on multiplexers can still be applied. The timing constraints are removed from the path which is disabled, but the path will be cut into two independent sections, each of them constrained.

We have enriched the XML syntax for loop breaker, which is designed for timing-driven P&R flow. Two new XML tags loop_breaker_delay_before and loop_breaker_delay_after have been introduced, as shown in Fig. 15. Fig. 16 illustrates an example of a multiplexer with loop breaker and associated timing constraints. The loop_breaker_delay_before aims at constraining the input delay of a multiplexer when a loop breaker is enabled, while the loop_breaker_delay_after aims at constraining the output delay. When the tags are enabled, the SDC generator generates two SDC constraints as exemplified in Fig. 17.

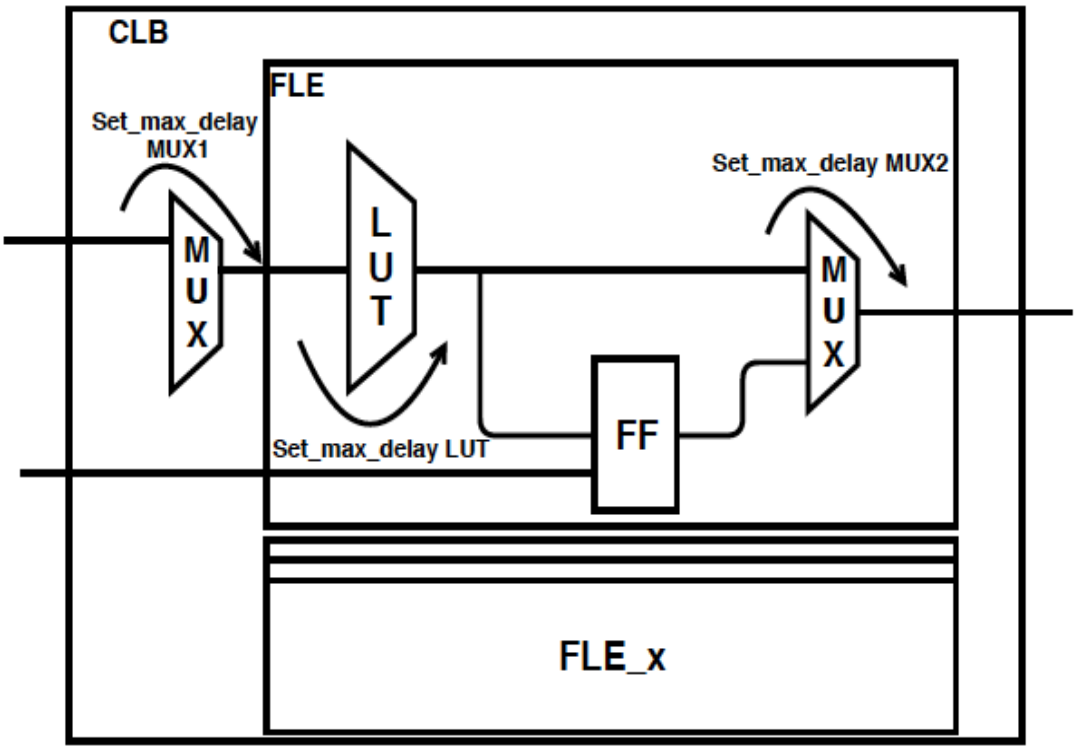


Figure 13: Methodology used to Constrain the CLB
Any user-defined value is used to do the constraining.

```
<complete name="crossbar0" input="clb.In fle[9:0].out" output="fle[9:0].in[0]"
circuit model name="mux 2level" loop breaker="clb.In[5:0] fle[5:0].out">
  <delay constant max="104e-12" in port="clb.In" out port="fle[9:0].in[0]" />
  <delay constant max="104e-12" in port="fle[9:0].out" out port="fle[9:0].in[0]" />
</complete>
```

Figure 14: Examples of Extended XML Syntax for Loop Breakers

```
<complete name="crossbar0" input="clb.I2 clb.I3 fle[9:0].out" output="fle[9:0].in[0]"
circuit model name="mux 2level" loop breaker="fle[9:0].out">
  <delay constant max="220.2e-12" in port="clb.I2 clb.I3" out port="fle[9:0].in[0]"/>
  <delay constant max="75.2e-12" in port="fle[9:0].out" out port="fle[9:0].in[0]"/>
  <loop breaker delay before max="1e-12" min="5e-13"/>
  <loop breaker delay after max="3e-12" min="15e-13"/>
</complete>
```

Figure 15: Example of Loop Breaker with New Syntax Loop Breaker

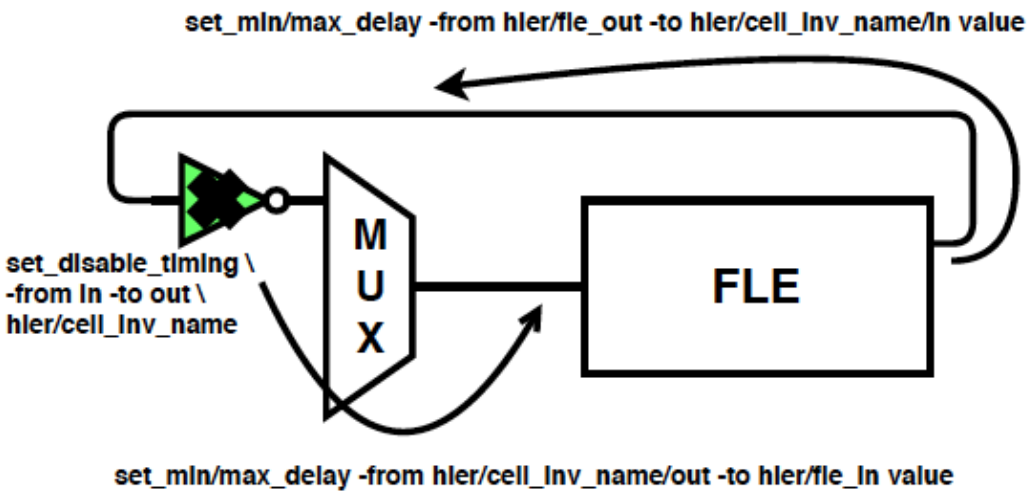


Figure 16: An Example in Constraining the input Delay and Output Delay of a Multiplexers using a Loop Breaker Defined in Fig. 15

```
set min delay -from grid clb 0 /fle 0 /fle out 0 -to grid clb 0 /mux 2level size46 0 /INVD1BWP 26 /I
0.000500
set max delay -from grid clb 0 /fle 0 /fle out 0 -to grid clb 0 /mux 2level size46 0 /INVD1BWP 26 /I
0.001000
set disable timing -from I -to ZN grid clb 0 /mux 2level size46 0 /INVD1BWP 26
set min delay -from grid clb 0 /mux 2level size46 0 /INVD1BWP 26 /ZN -to grid clb 0 /fle 0 /fle in 0
0.001500
set max delay -from grid clb 0 /mux 2level size46 0 /INVD1BWP 26 /ZN -to grid clb 0 /fle 0 /fle in 0
0.003000
```

Figure 17: SDC Example of Loop Breaker with New Syntax Loop Breaker Delay before and Loop Breaker Delay After

3.1.10 SDC Generation for Top-level Sign-Off

The SDC for FPGA top module aims at enabling timing sign-off for benchmarks implemented on FPGA fabrics. For benchmark-dependent analysis, our generator disables the timing analysis for unused resources in FPGA fabrics. Therefore, the intrinsic combinational loops of the FPGA can be disabled, while STA techniques can be applied to identify critical paths of benchmarks. However, during the practice, we notice that disabling unused resources is not enough and the unused inputs of multiplexer should be considered. Take the example in Fig. 18, the unused input of a *Switch Block* (SB) multiplexer (highlighted in red) can lead to unexpected combinational loops. To solve this issue, we improved our SDC to disable all the unused paths in timing analysis. By doing so, we transform all the non-used paths of the design into "invisible" pins, and only the benchmark path is revealed.

Our SDC generator has been significantly improved to provide high-quality and high-coverage timing constraints for our chip effort. Note that these SDCs are designed to be general purpose based on the SDC 2.1 syntax, which is portable between different PnR tools. Detailed usages are documented at [4].

1. **Time unit support:** Previously, our SDC generator used a fixed time unit as second. In practice, it caused many SDCs to be dropped during PnR practice as the backend tools

usually consider ns as the default time unit. To avoid the issues, our SDC generator now supports customizable time unit and delay values are adapted accordingly.

2. **Wildcard SDC generation:** As the FPGA capacity increases exponentially with array size, the SDC files could be tedious, causing long I/O time and large disk space. We have enabled the use of wildcards to compress the SDCs without losing generality. Using wildcards, our SDC files could consist of only hundreds of lines for any size of FPGAs.

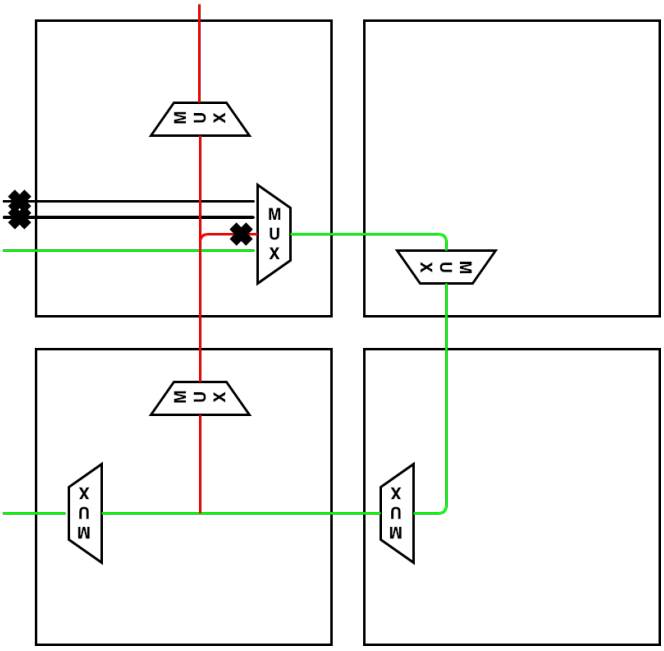


Figure 18: Combinational Loops caused by Unused Inputs of Multiplexers (in red)

3. **Flexible hierarchy:** Previously, our SDCs were designed for a flatten PnR strategy where full hierarchy is applied to each pin to be constrained. However, different PnR strategies require different hierarchies for pins to be constrained. To better support our hierarchical PnR strategy, our generator can output SDCs with or without full hierarchy.

4. **Disable timing for programmable resources:** As our FPGA fabric contains two clock domains (one for the core logic while the other is for configuration protocols), it is essential to cut off the timing arcs between the two clock domains. Our generator can output SDCs to disable the timing arcs through the configuration ports of any programmable resource across the fabric, e.g., LUT, routing multiplexers and I/Os. Note that the timing arcs inside configuration protocol are not disabled so that backend tool can resolve setup/hold violations.

3.1.11 Floor-planning and Layout Generation for Homogeneous FPGAs

To achieve a regular layout for FPGA, a carefully constrained floor-planning is required. Otherwise, a semi-custom design tool may place circuit cells randomly, leading to imbalanced pin-to-pin delay. Our goal is to perform the FPGA floorplanning for auto-generated Verilog

netlists following the island-style, commonly used in commercial products. To arrange the CLB in an array surrounded by CBs and SBs, we were developing and validating a TCL script using Cadence Innovus and TSMC 40nm technology. In order to support flexible and scalable FPGA architectures, the tcl scripts of floorplaning should be automatically generated. Fig. 19 shows a floor-planning generated by our Tcl script and using our auto- generated Verilog netlists that model a 3 ×3 FPGA array. The TCL script consists of two steps:

1. place all the CLBs in the center of the core area The coordinates of each CLB can be determined by estimating the full-chip area derived from total logic cell area and core utilization rate.
2. place the related SBs and CBs around the appropriate CLBs. Since all the CBs and SBs do not have the exact same area (in particular for those on the borders), we intend to consider the pessimistic case: using the maximum SB and CB area for instance.

Since we completed the Verilog generation part in T-2, we then focused on refining the P&R flows with Innovus to achieve properly floorplanned FPGA layouts. We have also sent our FPGA netlists to the OpenROAD team and are working towards the implementation in their flow. We have developed a TCL-based floor-planning script that places the CLBs, CBs and SBs in an island-style similar to modern FPGAs. We tested our back-end flow with the most popular academic FPGA architecture (K = 6; N = 10 - each LUT has an input of 6 and each CLB contains 10 BLEs). Fig. 20(a) and Fig. 20(b) illustrate the floorplan and final layout of a 10 ×10 FPGA using the ASAP 7nm PDK, respectively. To examine the scalability of our approach, we swept the FPGA array size from 2 ×2 to 10 ×10. Table 10 shows the layout area of FPGA sized from 2 ×2 to 10 ×10. Not limited to the ASAP 7nm PDK, our floorplan and back-end scripts have been tested on a few commercial PDKs, i.e., TSMC 40nm, TSMC 180nm and Global Foundry 130nm.

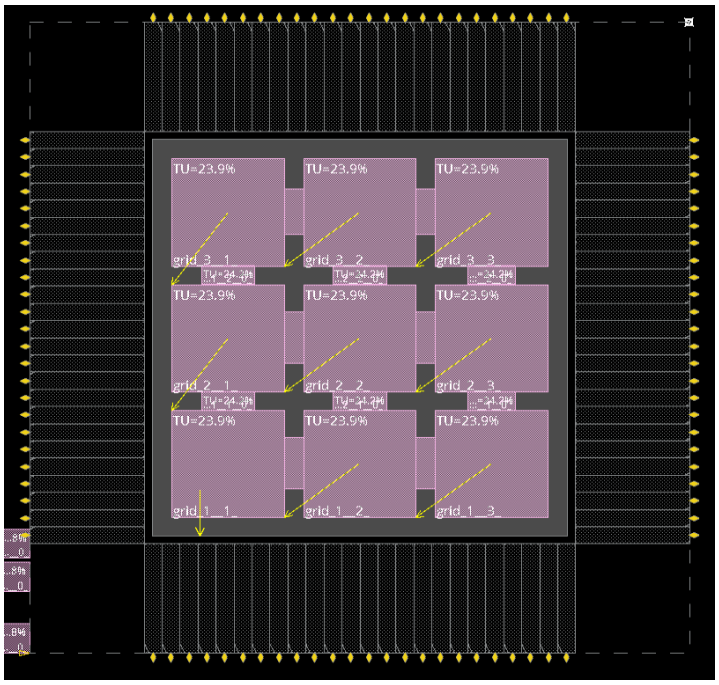
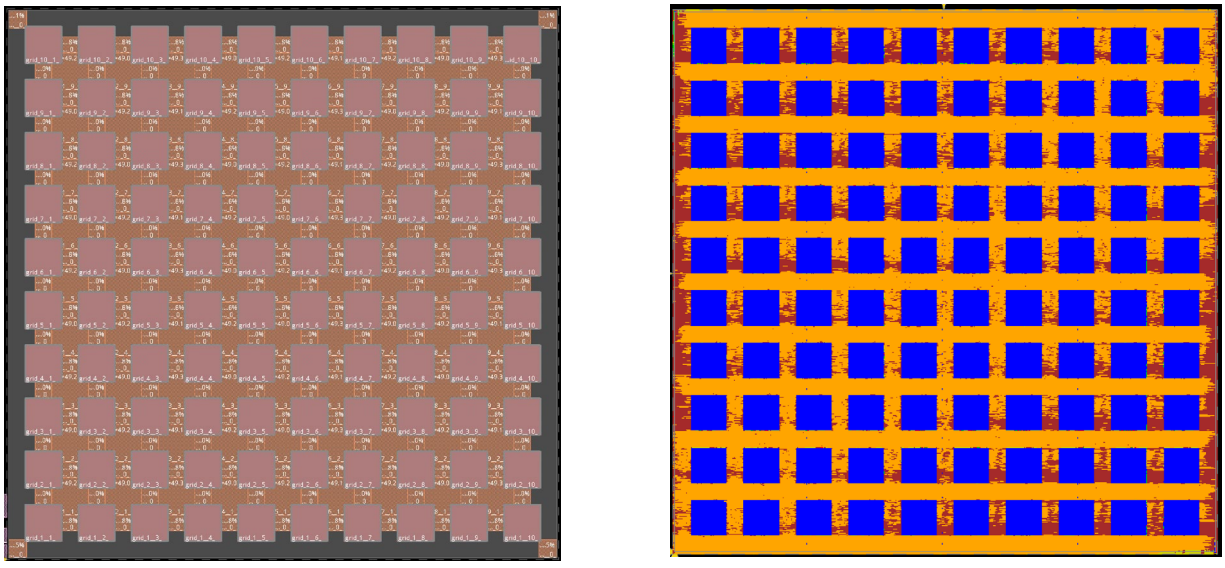


Figure 19: A Example of Floorplan Auto-generated by our Tcl Script

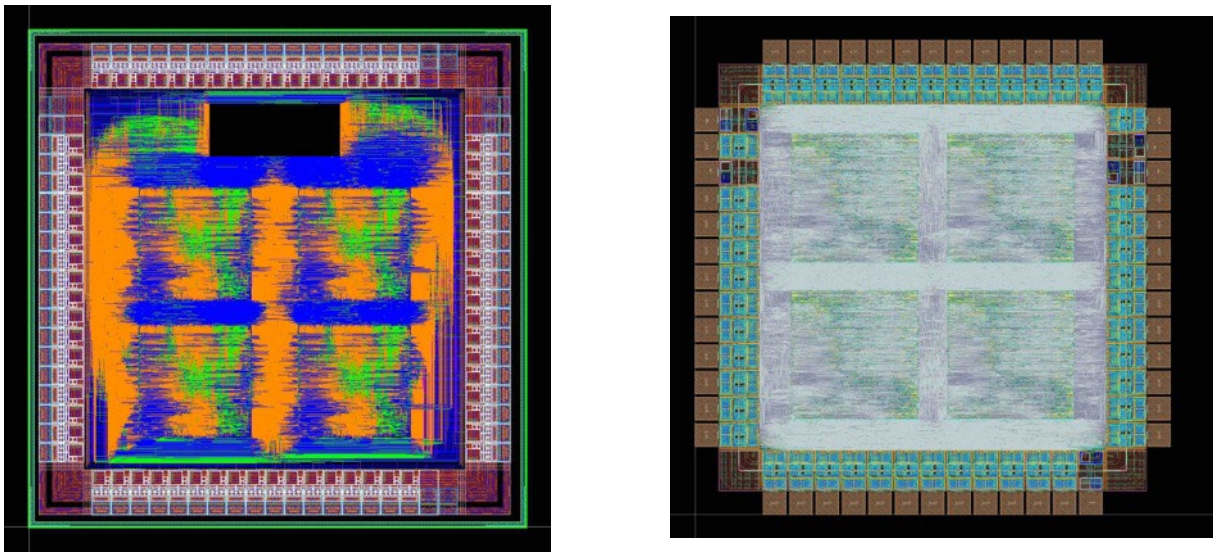
Table 10: Area of ASAP 7nm FPGA Layouts with Array Size Ranging from 2 × 2 to 10 × 10

FPGA size	2 ×2	4 ×4	6 ×6	10 ×10
Area (μm^2)	226,604	1,273,365	2,797,174	7,618,327



(a) Floorplanned **(b)** Layout

Figure 20: A 10 × 10 FPGA Fabric using ASAP 7nm PDK



(a) Using the Global Foundry 130nm technology

(b) Using the TSMC 180nm technology

Figure 21: Layout of 2 × 2 FPGA Fabric

Fig. 21(a) shows a full FPGA layout using the Global Foundry 130nm technology. This chip was sent for fabrication funded by other sources. Note that the tiny black area at the top of the chip is used for another project. Complete post-layout simulations have been performed using our bitstream and testbench generator. Fig. 21(b) presents a full FPGA layout using the TSMC 180nm technology. This chip (with our coming heterogeneous fabric) was sent for fabrication the following year and funded by other sources.

3.3 T-3: Develop Auto-generation for Self-testing Verilog Testbenches

Here, we introduce the technical details of the Verilog testbench generation, which supports automatic functional testing of the generated FPGA Verilog netlists and bitstreams. First, we propose a new XML syntax in order to drive user-defined primitive circuits/FPGA inputs with proper signal stimuli. Then, we present a self-testing Verilog testbench, which can verify both the configuration peripheral circuits and the core logic. However, since the simulation complexity of the configuration phase grows exponentially to the FPGA array size, we also developed a configuration-skip testbenches which focus on validating the core logic of the FPGA fabric.

3.3.1 XML Syntax Supporting Testbench Generation

Building an effective Verilog testbench requires users to understand the functionality of each inputs. To automate the testbench generation, we add the XML syntax shown in Table 11 to the port of circuit module, with which we can identify the proper stimuli for these ports. For instance, when the property is prog of a clock port is set to true, our testbench generation can drive this port with a clock signal designed for scan-chain programming.

Table 11: Description of Additional XML Syntax Defining Ports of a Primitive Block

XML Syntax	Description
is config enable	can be either true or false. Only valid when is global is true. Specify if this port controls a configuration-enable signal. This port is only enabled during FPGA configuration, and always disabled during FPGA operation. All the config enable ports are connected to a global configuration-enable voltage stimuli in testbenches.
default val	default logic value of a port, which is used as initial logic value of this port in testbench generation and bitstream generation. Can be either 0 or 1. We assume each pin of this port has the same default value.
is prog	an be either true or false. Specify if this port is used for FPGA configuration peripheral circuits. All the ports in this purpose are connected to programming stimuli in testbenches.
is set	can be either true or false. Specify if this port controls a set signal (Sometimes required by flip-flops or scan-chains depending on the technology library). Only valid when is global is true. All the set ports are connected to a global set voltage stimuli in testbenches.
is reset	can be either true or false. Specify if this port controls a reset signal. Only valid when is global is true. All the reset ports are connected to a global reset voltage stimuli in testbenches.

3.1.12 Verilog Testbench Generation for Testing FPGA Fabric

Fig. 22 depicts the organization of a Verilog testbench supporting automatic checking. Two top modules, i.e., the generated FPGA fabric and the reference Verilog benchmark, are instantiated. To ex- amine the functionality of both configuration peripheral and core logic, the auto-check Verilog testbench includes two phases:

1. *The configuration phase:* Each memory cell, e.g., SRAM, is programmed serially according to the bitstream. During each programming cycle, a memory cell is configured individually. During this period, all the I/Os of FPGA are kept at logic 0.

2. *The operating phase:* Configuration circuits are powered off and testing input patterns are fed to the FPGA I/Os. Test vectors are generated as waveforms using the signal activities information of the benchmark circuit. The output waveforms of both top-level modules are matched by pairs and checked at any falling edge of the operating clock by an XOR operator. If the flag rises the simulation stop and the tool print the name of the flag who raise.

Fig. 23 shows a successful verification of a toy benchmark consisting of a 32-depth 1-bit FIFO, where the FPGA output `out data out fpga` pad is the same as the benchmark output `out data out` benchmark. As a result, the checking signal `out data out verification` is always '0', indicating that the FPGA implementation is equivalent to the benchmark.

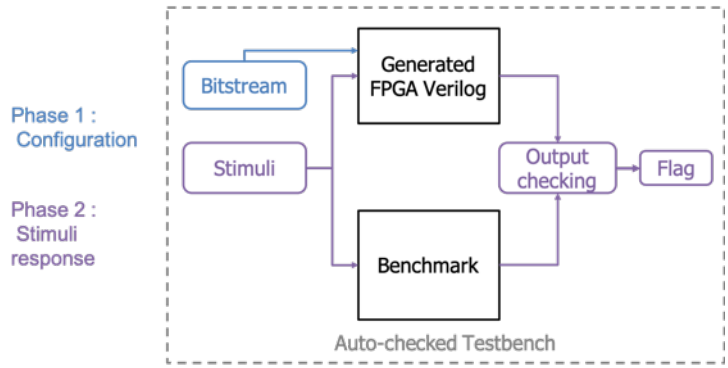


Figure 22: Auto-check Verilog Testbench Technique

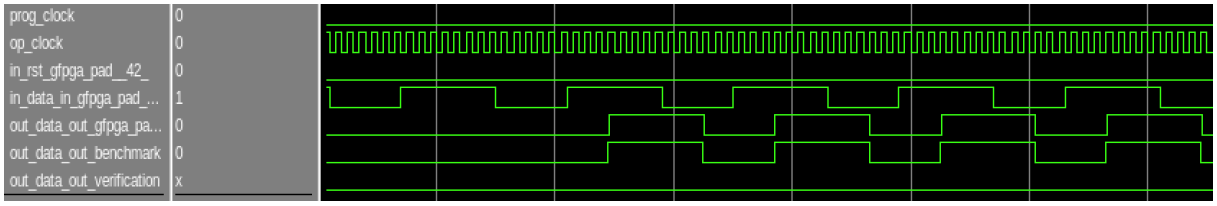


Figure 23: Waveforms Output by a 32-bit FIFO using auto-check Verilog Testbench

3.1.13 Configuration-skip Verilog Testbench Generation

As FPGAs contain a large number of configuration memory bits, HDL simulations for the programming phase is time-consuming. For instance, a PicoRV32 [5] benchmark would require more than a day for the configuration phase simulation. To accelerate the simulation and focus on the core logic, we have developed configuration-skip Verilog testbenches that only include the operating phase. The testbench still include the FPGA IP, the benchmark and the flags as shown in Fig. 22, but the programing phase is bypassed by initializing the configuration memories with the bitstream information at time zero of the simulation. The FPGA module is initialized by reading an .hex file containing the full bitstream. The development of configuration-skip Verilog testbench was finished and tested.

3.1.14 HDL Simulation Script Generation

To provide complete "push-button" HDL simulations, we also provide TCL scripts to interface with Modelsim. The scripts are generated along with the Verilog testbenches, which can import all Verilog netlists into Modelsim, compile and launch simulations.

3.1.15 Functional Verification Techniques

To support HDL simulation based functional verification, we have developed timing-annotation and signal initialization techniques in our Verilog testbench generation. The techniques are developed to assist convergence in HDL simulations due to combinational loops. Fig. 24 illustrates two typical cases of combinational loops in FPGA architectures. The loops indeed exist even in real FPGAs but come from unused logic and routing resources, and thus do not impact performance. However, the loops cause undeterminable signals ('X') in HDL simulations, leading to verification errors.

To handle the combinational loops in global routing architecture, we added signal initializations in Verilog netlists, which are enabled only in HDL simulations. Fig 25 shows an example of a Verilog module INV with signal initialization. To avoid covering any potential problems in FPGA architectures, the signal forcing deposits a value to the output, which can be overwritten by any other driving signals. The initial values to be forced can be customized by the XML property `default_val`. Our HDL simulation showed that the signal initialization resolve the undeterminable signals in the global routing architectures.

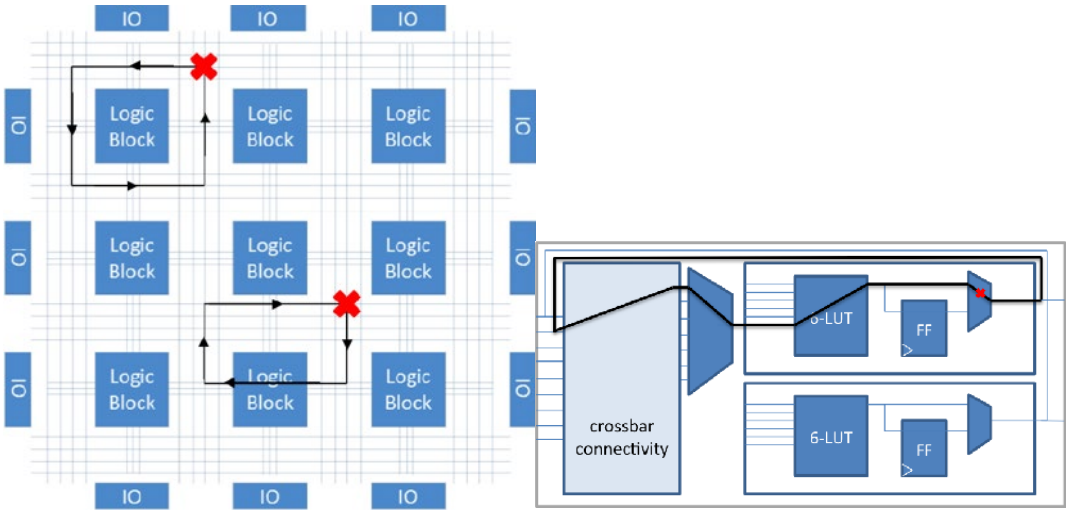


Figure 24: Combination Loops in FPGA Architectures
(a) Global routing; and (b) Local routing [6].

```
module INV (
  input [0:0] in,
  output [0:0] out);
  assign out = ~in;
  `ifdef ENABLE TIMING
//------ BEGIN Pin-to-pin Timing constraints -----
  specify
    (in[0] => out[0]) = (0.01, 0.01);
  endspecify
//------ END Pin-to-pin Timing constraints -----
  `endif
  `ifdef  INITIALIZATION
//------ BEGIN driver initialization -----
  initial begin
    $signal force("in", 0, 0, 1, , 1);
  end
//------ END driver initialization -----
  `endif
endmodule
```

Figure 25: Example of Timing-Annotation and Signal Initialization in Verilog Netlists

To handle the combinational loops in local routing architecture, we use timing annotations, as shown in Fig. 25. This approach breaks the pure simulation zero-delay combinational loop, so the simulator can evaluate the combinational logic in two steps, as it happens in the reality. To customize the delay values, we developed novel XML syntax under XML node `circuit`

model, as exemplified in Fig. 26. Using both timing annotation and signal initialization, the undeterminable signals can be resolved for the entire FPGA architectures in many practical benchmarks. To enable the two techniques, we also added two new command-line options to VPR, `--fpga verilog include timing` and `--fpga verilog init sim`.

3.1.16 Formal Verification Techniques

Formal verification using Synopsys formality is in place and helped us identified the critical bugs/limitations in the multi-mode support of VPR (as they resulted in incorrect bitstrams and therefore faulty benchmark implementations...). Formal verification are implemented and we have a formal pass on 8 of the MCNC benchmarks. Synopsys formality has some difficulties in automatically matching all the internal register on FPGAs bigger than 10×10 CLBs. We patched this issue by automatically generating verification scripts that "user-match" the different internal registers. All the verification was done on a classical fracturable LUT6 architecture, a more complex architecture (lookalike of the Stratix 4 - see Fig. 3) with fully connected local routing and the same architecture with a half-connected local routing. Some possible future issues might come as Yosys tends to "change" the benchmarks (like converting asynchronous resets to synchronous resets, *etc.*)

```
<circuit model type="inv buf" name="INVTX1" prefix="INVTX1" is default="1">
  <design technology type="cmos" topology="inverter" size="1" tapered="off"/>
  <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>
```

Figure 26: Example of Customizing Timing-annotation in Architecture XML Language

3.1.17 Using Standard Testing Patterns in HDL Simulation

Due the bugs found in Formality, we looked for yet another way to verify these implemented benchmarks and we tried to use an ATPG (TetraMAX). But this tool requires structural Verilog netlists while our benchmarks are behavioral. We tried to translate the netlists using Design-Compiler and Synplify. However, Synplify generates netlists incompatible with TetraMAX requirement (another support ticket confirms it) by using the. option in a *defparam* command. Design-Compiler generates netlists including standard-cell in behavioral Verilog, we solved this issue by creating a small standard-cell library in structural Verilog. Succeeding in this way could resolve another issue coming from the benchmarks written in another language than Verilog (as VHDL or SystemVerilog). Indeed, our tool requires a blif file which is generated by Yosys which needs a Verilog file as input. If we control the standard-cell library, we can better handle Yosys in its blif generation.

3.1.18 Open-source Verilog Simulator Support

All functional verification efforts were done using Icarus Verilog. We have added Icarus as an option to our flow to perform functional verifications. To overcome the issue of Icarus where our signal initialization is always overwritten by the initial X state, we set a delay of one minimum timescale to let the initialization overwrite the x state. We have updated the testbench generation engine to meet the Icarus Verilog requirements. Functional verification also had a new testbench for pre-programmed FPGA. This test-bench is completely random and its goal is to save simulation time by skipping the programming part.

3.1.19 Support Bitstream File Loading in Testbench Generator

This change was motivated by our experience in developing bitstream downloader for chip testing as well as QuickLogic's eFPGA development. Previously, OpenFPGA's Verilog testbench generator hardcoded bitstream when writing testbench netlists. This has caused large file sizes, and the testbench could not be used as a reference when developing bitstream downloaders. We have reworked OpenFPGA to address the issue:

1. First, OpenFPGA's bitstream generator can now output a bitstream file (.bit), being compatible to Verilog standards;

Second, the testbenches generated by OpenFPGA now use the readmemb command to read a bitstream file and use a *Finite-State Machine* (FSM) to download bitstream. As such, the bitstream downloading logic in testbenches are synthesizable (for Xilinx FPGAs). Engineers can now easily implement a bitstream downloader based on the testbenches.

3.1.20 Commands to Call Various Testbench Generators

To enable atom-level testbench generation, we have deprecated the command `write verilog testbench`, which carries too many functionalities. Now, the command `write verilog testbench` is replaced by four new commands (see details in [7].):

1. The `write full testbench` is designed to generate full testbenches, corresponding to the option `--print top testbench` in the deprecated command;
2. The `write preconfig fabric netlist` is designed to output preconfigured FPGA netlists, corresponding to the option: `--print formal verification top netlist` in the deprecated command;
3. The `write preconfig testbench` is designed to output preconfigured FPGA testbenches for the preconfigured FPGA netlist, in the deprecated command, it is corresponding to the option `--print preconfig top testbench;`
4. The `write simulation task info` should be created to output the exchangeable simulation in a file, corresponding to the option `--print simulation ini` in the deprecated command.

Support default net type in testbench generator: only the fabric generator of OpenFPGA supports `default_net_type` while the testbench generators do not. This has caused compatibility issues in some of the HDL simulators, such as Modelsim. Therefore, we upgraded the testbench generators to support this feature. This smooths out the verification in our tape-out projects, which are based on Modelsim. See details in [8].

Streamline Testbench Organization: Previously, the auto-generated testbenches from OpenFPGA relied on preprocessing flags, i.e., AUTOCHECKED SIMULATION, ENABLE FORMAL VERIFICATION and FORMAL SIMULATION, to switch between different simulations. This has caused issues for users since it is complex to manipulate the flags due to the strong dependencies. It is easy to fail verification due to the misuse of these flags. In addition, these flags impose the full version of testbenches to be outputted, which caused large file sizes on hard disk. Therefore, we have removed the use of flags and enabled outputting trimmed testbenches. Users can easily generate multiple versions of testbenches, by calling the testbench generators multiple times in a OpenFPGA shell script with different options. See details in [9].

3.1.21 Micro Benchmarks

We have added a series of micro benchmarks that are scalable and hence can fully utilize FPGA fabrics at different sizes. These benchmarks include counters (aim to test LUTs and FFs), mac units (aim to test DSP blocks), and adders (aim to test adders). See details in [10, 11]. We successfully functionally verified all the MCNC benchmarks, 14 of the EPFL benchmarks and the PicoRV32 [5]. Through our testing, we detected a bug in Formality, which has been confirmed by the Synopsys team after creation of a support ticket. We suppose that Formality reported false failures on failed benchmarks.

3.2 T-4: Design a Bitstream Generator and Download Manager

To generate complete bitstreams for a customizable FPGA fabric, our methodology is to (1) extract the mapping information from VPR packing, placement and routing results; and then (2) convert the mapping results to configuration bits by considering the topology of programmable circuits as defined in XML. In this section, we first introduce how we extract configuration of programmable resources from VPR results (see Section 3.4.1), and how to translate mapping results according to the programmable circuitry (See Section 3.4.2).

3.2.1 Back-annotation on VPR Mapping Results

To acquire bitstream, we must extract mapping results from the VPR data structures. In principle, VPR stores mapping results in two separated data structures: (1) a local *Routing Resource* (RR) graph is created for each CLB to contain packing results; and (2) a global *Routing Resource* (RR) graph for CBs and SBs which is created to contain global routing results. Each type of RR graphs are left untouched after each stage. However, the logic equivalence of FPGA modules, such as LUT inputs, allows many routing optimizations at different stages, which meanwhile breaks the consistency of the two types of RR graphs. Therefore, to achieve a complete and correct mapping results, we applied the following modifications:

3.2.1.1 RR Graph Synchronization

CLBs typically have a fully-connected local routing architecture, which allows any CLB input to access any LUT or BLE input. Thanks to this feature, VPR can re-arrange the mapping results on the CLB inputs during the global routing stage to enable high-level optimization without causing any packed CLBs to be unroutable. Fig. 27 illustrates an example comparing the net mapping to CLB inputs at post-packing and post-routing stage. However, we identified that VPR does not back-annotate the net mapping to the local RR graph, leading to out-of-date routing results inside CLBs. This does not affect the performance evaluation results of VPR but do cause incorrect bitstream generation. For example, this would cause the configuration of local routing multiplexers to be wrong. We fixed this limitation to guarantee a proper matching between the RR graphs by re-routing the local routing architecture according to global routing results.

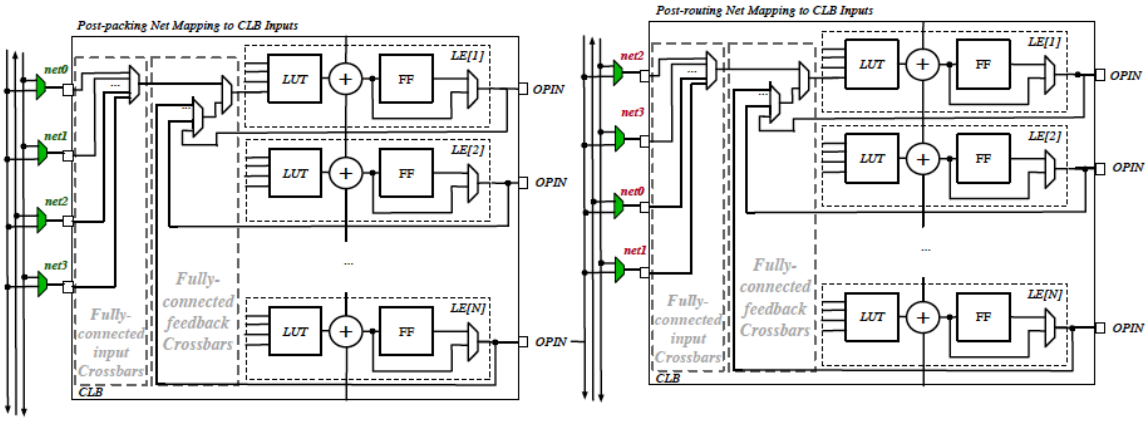


Figure 27: Mismatched Net Mapping Between VPR Post-packing and Post-routing Results

3.2.1.2 Truth Table Adaption for LUT Pin Mapping

A *k*-input LUT can map any *k*-input function single-output logic functions, which create a large optimization space. To avoid routing congestion, VPR can freely move an input of a logic function to any LUT input, as depicted in Fig. 28. While legitimate, such a re-assignment requires an adaption on the original truth table, otherwise the LUT functionality will be different from the mapping logic function. This feature was not initially supported by VPR and we updated the code to now guarantee correct LUT configuration bits assignments, as illustrated in Fig. 28.

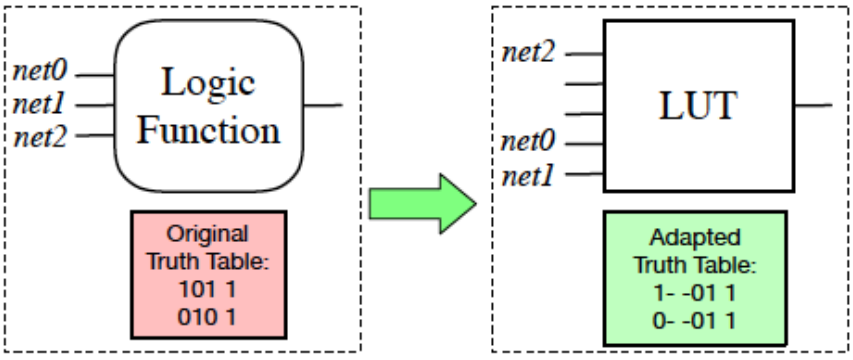


Figure 28: Swapped Pins when Mapping a Logic Function to a LUT

These modifications guarantee an accurate (compared to the hardware) representation of the mapped benchmarks by the VPR data structures, as validated by our self-testing testbenches.

3.2.2 Bitstream Decoding

After a successful back-annotation of the VPR data structure, our bitstream generator traverse the two types of RR graphs and decode configuration bitstream according to the circuit topology of LUTs and multiplexers.

3.2.2.1 Bitstream Decoding for CLB Mapping Results

To support the hierarchy of the CLB architecture, we developed a recursive Depth-First Search (DFS) algorithm whose pseudo code is shown in Algorithm 3. The algorithm first recursively visits of all the child pb type under current node. When a LUT primitive pb type is reached, we decode the adapted truth table into its bitstream. After visiting each child pb type, the algorithm decodes the bitstream for each multiplexer in the local routing architecture. Our bitstream generation supports one-level, two-level and tree-like multiplexers, covering most frequently used types in modern FPGA architectures. As such, our algorithm can support any CLB architecture that can be described by VPR.

```
Function rec_gen_bitstream_pb_type(current_pb_type):
    foreach child_pb_type ∈ current_pb_type do
        rec_gen_bitstream_pb_type(child_pb_type);
    end
    if (TRUE == is_primitive_pb_type_lut(current_pb_type)) then
        gen_bitstream_lut(current_pb_type);
        return;
    end
    gen_bitstream_mux(current_pb_type);
end
```

Algorithm 3: Bitstream generation engine for CLB (pseudo-code).

3.2.2.2 Bitstream Decoding for CB/SB Mapping Results

Algorithm 4 depicts the engine to generate the bitstream of the CBs. We enumerate all the coordinates of CBs in VPR and identify if each node in the rr graph belongs to the CBs. For each node in the CBs, we evaluate if it corresponds to a multiplexer (and consider its fan-in). Then, our engine generates the bitstream for multiplexers according to its structure, such as one-level, two-level and tree-like. The bitstream generation for the SBs share the sample principle as Algorithm 4.

```
rr graph: VPR Routing Resource Graph.
fpga size: Array size of a FPGA architecture.
Function gen bitstream cb(rr graph:):
    foreach x ∈fpga size do
        foreach y ∈fpga size do
            foreach node ∈rr graph: do
                if (FALSE == is rr node in cb(node)) then
                    continue;
                end
                if (is rr node mux(node)) then
                    gen bitstream mux(node);
                end
            end
        end
    end
end
```

Algorithm 4: Bitstream generation engine for CB (pseudo-code).

Fig. 29 shows an example of outputted bitstream in text file. Our bitstream generator supports a scan-chain-based FPGA fabric, and the bitstreams have been validated using self-testing test benches as shown in Section 3.3.

```
1, // Configuration bit No.: 12, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 10
1, // Configuration bit No.: 11, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 9
1, // Configuration bit No.: 10, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 8
1, // Configuration bit No.: 9, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 7
1, // Configuration bit No.: 8, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 6
1, // Configuration bit No.: 7, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 5
1, // Configuration bit No.: 6, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 4
1, // Configuration bit No.: 5, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 3
1, // Configuration bit No.: 4, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 2
1, // Configuration bit No.: 3, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 1
0, // Configuration bit No.: 2, SRAM value: 0, circuit model name:
mux 1level tapbuf, circuit model index: 0
0, // Configuration bit No.: 1, SRAM value: 0, circuit model name:
mux 1level tapbuf, circuit model index: 0
1, // Configuration bit No.: 0, SRAM value: 1, circuit model name:
mux 1level tapbuf, circuit model index: 0
```

Figure 29: Example of Outputted Bitstream

3.2.3 Yosys Script Tuning for BRAMs

We have tuned Yosys scripts to support various BRAMs, in particular for LeWiz benchmarks. We have managed to make yosys map the dual-port dual-clock ansynchronous FIFOs to the dual-

port dual-clock BRAMs. However, we detected two potential problems:

1. the special type of BRAM is not supported by ARM IP compiler. As a solution, we investigated two alternatives:
   (a) Use LUT RAMs instead of the BRAMs. This could be a good option when LeWiz does not have any access to ARM IPs.
   (b) Use alternative SRAM providers. It could be an easy path since we already have working benchmarks.

   To ease the integration of LeWiz benchmarks, we provided Yosys scripts and sufficient guidelines for LeWiz, including the supported BRAM/LUT RAMs verilog modules.
2. A few combinational loops are still reported by Yosys and downstream tools. We analyzed Yosys reports and provided useful debugging information for LeWiz.

3.2.4 LUT RAM Support

We initiated the evaluation of LUT RAMs that are already supported with our XML syntax and Verilog generator. To manipulate a LUT as BRAM, a control logic module can be added aside to the LUT, as illustrated in Fig. 30. The RAM control logic can be user-defined Verilog module and be treated as a blackbox for Verilog generator, as shown in Fig. 31. The RAM module can be added to CLB architecture using the regular FPGA architecture description language. We have completed a FPGA architecture file, which include two types of CLBs (one for regular CLB and the other for CLB RAMs, similiar to Xilinx products). Each type of CLBs is arranged in columns and interleaved. Testing and verfication complemented this effort.

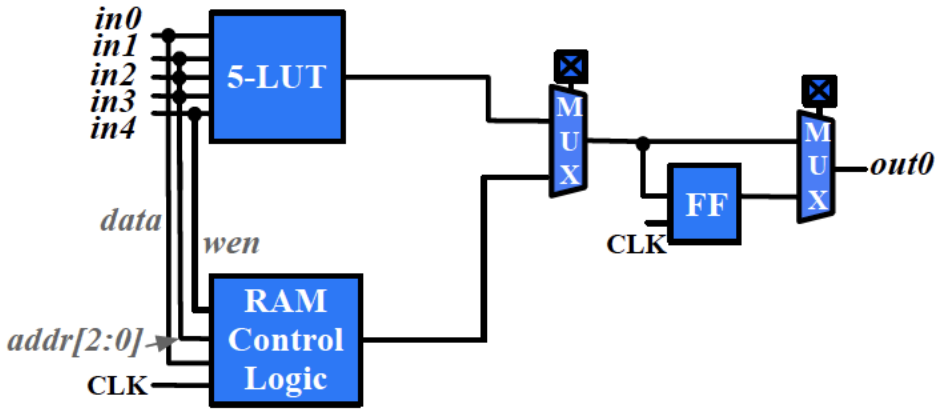


Figure 30: Schematic of LUT RAM

```
<!-- Circuit model definition for RAM control logic -->
<circuit model type="hard logic" name="ram ctrl" prefix="ram ctrl"
verilog netlist="ram.v">
  <!-- Port map corresponds to the schematic in Fig. 30 -->
  <port type="input" prefix="wen" size="1"/>
  <port type="input" prefix="addr" size="2"/>
  <port type="input" prefix="data" size="1"/>
  <port type="clock" prefix="clk" size="1"/>
  <port type="output" prefix="out" size="1"/>
</circuit model>
```

Figure 31: XML Example for Describing a CLB RAM

3.2.5 Refactored Flow-run Scripts

We have refactored our flow-run scripts to interface more HDL simulators, as depicted in Fig. 32. To be easy to extend, we enhanced FPGA-X2P to output an `.ini` file which includes exchangeable simulation information. This file includes:

1. a list of Verilog netlists that are required to run HDL simulations;
2. the module names required to build scripts for HDL simulators;
3. recommended timescale, precision and duration.

Our python scripts can parse the .ini file and built customized scripts to interface various simulation tools. As such, we offload the trivial script generation from FPGA-X2P to python scripts, to avoid frequent modification on FPGA-X2P due to any update in downstream tools. By default, our main script run fpga task.py supports iVerilog simulator. Alternatively, users can run Modelsim simulations by calling another python script run modelsim.py, which can generate .do files for different version of Modelsim based on the exchangeable simulation information. Note that FPGA-X2P generates standard Verilog netlists and testbenches which are applicable to both iVerilog and Modelsim. None of the python scripts will modify the Verilog netlists.

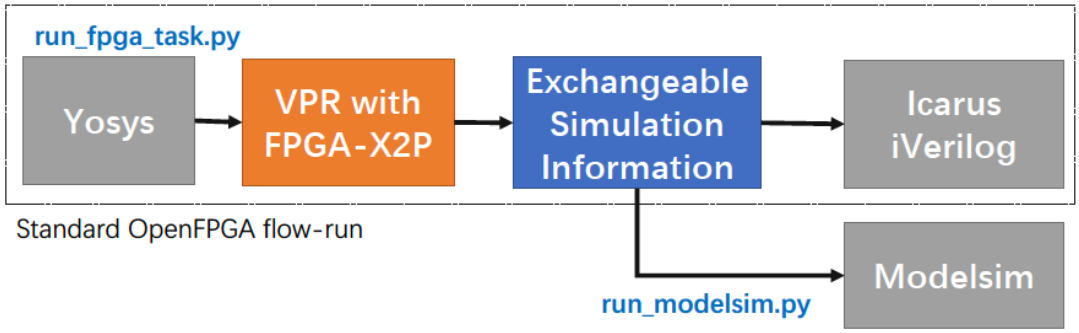


Figure 32: Flow-run Script and Supported Tool Chains

The bitstream generator can output two types of bitstreams, as illustrated in Fig. 33:

1. **Generic bitstreams**, where configuration bits are organized out-of-order in a database.

The bitstreams can freely be outputted to a given standard file format, i.e., FASM[1]. We output to a XML format, which is easy to debug. Generic bitstreams allow industry and other contributors to work with OpenFPGA while keeping proprietary information hidden.

2. **Fabric-dependent bitstreams**, where configuration bits are organized 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. Industrial contributors can easily develop their own fabric-dependent bitstream generators and keep them as proprietary codes.

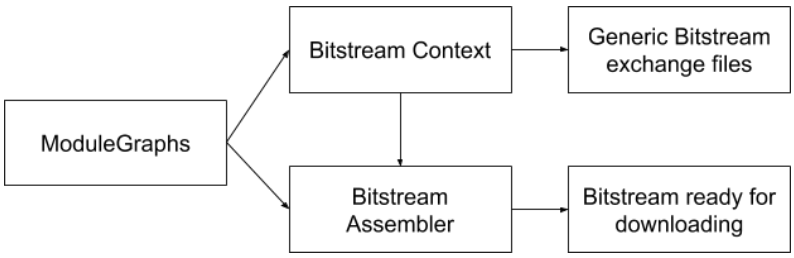


Figure 33: Module Graph to Drive Fabic-independent and Fabric-dependent Bitstream Generation

3.2.6 Interchangeable Bitstream File

OpenFPGA supports bitstream build-up through an external file input. The bitstream file follows an XML format, whose details are available in the online documentation [12]. This allows users to build synthetic bitstreams which may not be synthesizable by VPR but useful for testing, as illustrated in Fig. 34.

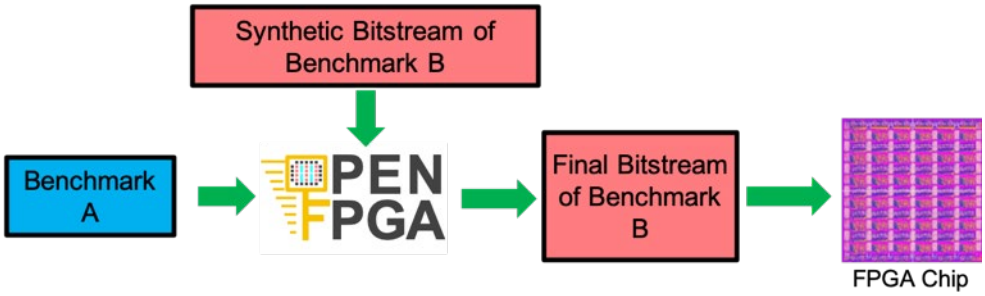


Figure 34: Synthetic Bitstream Loading Enabled by the Interchangeable Bitstream File

3.2.7 Bitstream Overloading through .eblif File

To enhance the arithmetic support in modern FPGAs, LUTs with fracturable outputs are also used to implement adder functions. The carry input and carry output ports of these LUTs are connected in a chain, in the same concept as the 7-series look-ahead carry chain. An example

[1] https://docs.verilogtorouting.org/en/latest/utils/fasm/#fasm-metadata

can be found in the QLSOFA architecture [13]. However, a LUT mapped to an adder function (3 input and 2 output) does not comply with the conventional LUT definition, which only accepts 1 output. As a result, the LUT has to be modeled as `.subckt` in `.blif` format. The `.subckt` model cannot contain the truth table information for the LUTs, which are required to generate bitstream to configure the LUTs. This is due to the fact that a `.subckt` model is typically considered as a black box. The truth table information can be defined through the `.attr` and `.param` annotation in `.eblif` file format.

OpenFPGA now supports the LUT bitstream to overloaded from the `.attr` and .param lines in `.eblif` files. Users can customize which pb type in the VPR architecture can be overloaded, through a bitstream setting file (see file format section in documentation [14]). Also, new commands for read/write bitstream setting files are added to the OpenFPGA shell. The feature potentially can be exploited to build synthetic benchmarks, to ease testing.

3.2.8 Runtime and Memory Improvement

We have optimized the runtime and memory usage of OpenFPGA on the bitstream generation for large fabrics, e.g., 100k-LUT FPGAs. In Table 12, we see a significant reduction on runtime and memory usage. We believe the runtime and memory usage is acceptable for end-users to fast iteration on their implementations.

Table 12: Runtime and Memory Comparison Before and After Optimization on the 100k-LUT FPGA

	Before Optimization	After Optimization
Runtime	~90 minutes	9 minutes
Memory Usage	~20Gb	4.5Gb

3.2.9 Patching and Changes

Critical patch on bitstream generator: NYU has offered benchmarks with logic obfuscation to be tested on OpenFPGA. During the process, a critical bug on OpenFPGA's bitstream generator has been detected. Even though the bug occurred only with one benchmark, it is a generic bug that could happen on complicated benchmarks. We have patched our bitstream generator and used NYU's benchmark in CI [15].

Report bitstream distribution: During our chip measurement, we realized that knowing the bitstream size per block in an FPGA helps when developing and debugging bitstream downloader. To smooth the chip testing process, OpenFPGA has been upgraded to report the number of configuration bits per block to an XML file. See details in [16].

Customizable path support for routing multiplexers: Previously, OpenFPGA's bitstream generators had a built-in strategy when generating bitstream for unused routing multiplexers. Driven by the verification requirements from QuickLogic's eFPGAs, we upgraded the OpenFPGA's bitstream generator to allow users to customize which input to be used for each routing multiplexer. See details in [17].

3.3 T-5a: Update Verilog Netlist Generator

In this task, we have reforged the Verilog and the bitstream generation engines. A major benefit of the new Verilog generation engine is that we can quickly add writers to output netlist in a different format, e.g., SPICE, VHDL, SystemVerilog *etc*. This allows us to interface versatile backend tools in an easy and clean way.

3.3.1 Integration to VPR8

We have completed the adaption on all the OpenFPGA data structures in the context of VPR8 environment [18, 19], as highlighted green in Fig. 35. Our regression tests have been adapted to the new OpenFPGA shell interface and deployed in the Travis continuous integration [20]. Current regression tests are based on a small set of micro benchmarks. We enriched our benchmark sets by adding representative and scalable HDL designs, such as *Finite State Machines* (FSM).

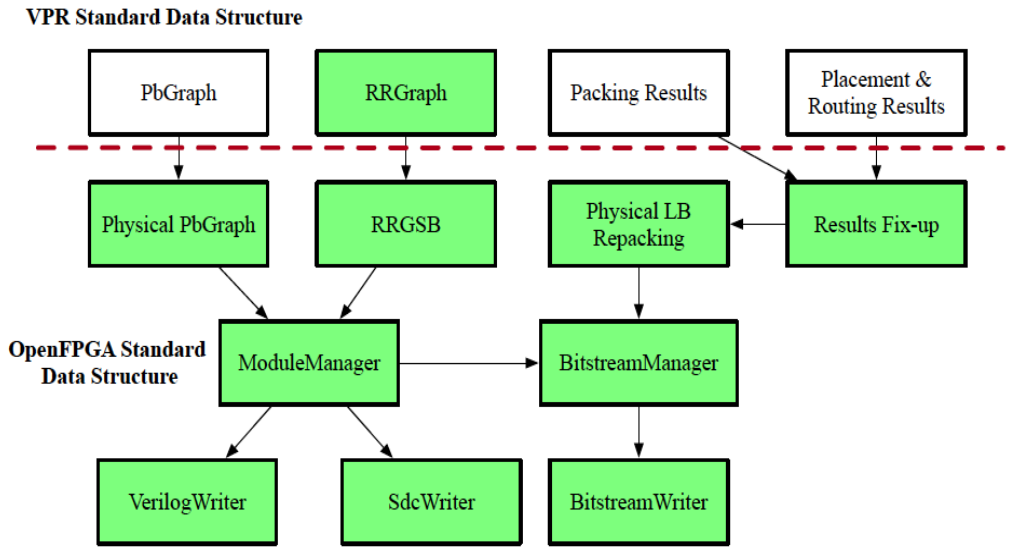


Figure 35: Progress on Integration to VPR8 in Terms of Modularized Data Structures

3.3.1.1 User Interface

OpenFPGA now adopts a shell-like interface, being similar to commercial tools. As depicted in Fig. 36, OpenFPGA can run in two modes: interactive and script. The interactive mode allows users to execute tools at runtime, which is easier to debug than previous command-line options. OpenFPGA shell can also accept scripts crafted by users to launch flows. For example, users can create different scripts for hardware development, i.e., generate fabric Verilog netlists or for implementing users' Verilog designs, i.e., bitstream generation. This brings a larger design space for users to exploit OpenFPGA's capability and eases the use of various tools inside OpenFPGA.

Table 13 shows a detailed comparison between the OpenFPGA with VPR7 version and OpenFPGA with VPR8 version. In addition to the significant changes in user interface, the new version is fully modularized and keep a loose integration with VPR. As such, we can avoid a hard fork on VPR and easily embrace the latest VPR versions with least manual efforts. Being

fully modularized, it is very easy to integrate new tools as well develop new features. On the other side, as OpenFPGA efficiently reuses VPR libraries and formalizes data structures, we natively support:

1. multiple I/O types, which means that OpenFPGA can support transceivers and Serdes blocks,
2. multiple-column IPs, which means that OpenFPGA can support embedded processors as well as other large IPs.

Table 13: Technical Comparison on OpenFPGA with VPR7 and VPR8

Item	OpenFPGA + VPR7	OpenFPGA + VPR8
User Interface	Command-line	Shell-like
Accept scripts	✗	✓
Time to integrate a new version of VPR	8+ weeks	<1 week
Multiple IO types	✗	✓
Multi-column IPs	✗	✓

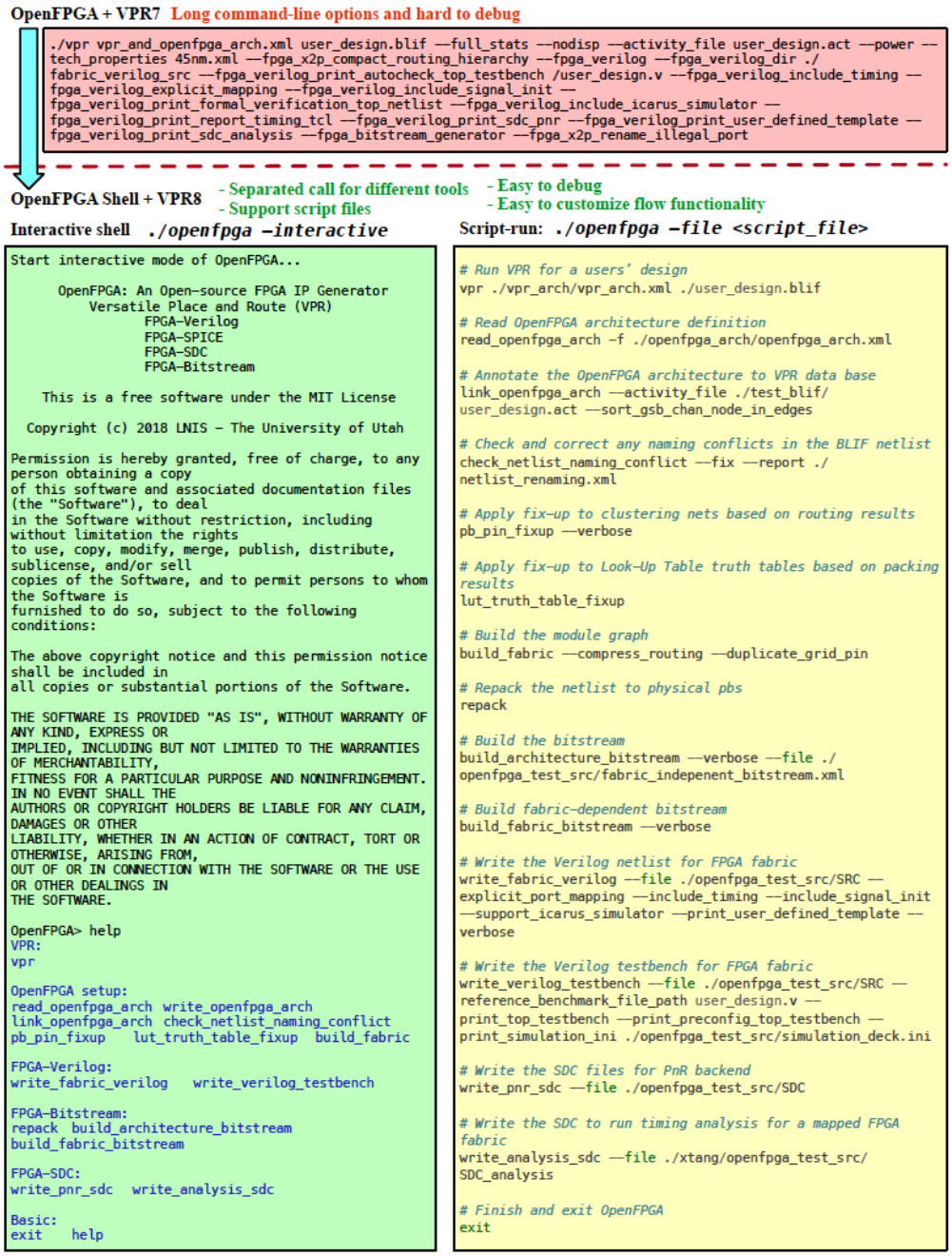


Figure 36: Comparison on user Interface between OpenFPGA with VPR7 and OpenFPGA with VPR8.

3.3.1.2 Routing Resource Graph Modularization

VPR uses discrete data structures to model a *Routing Resource Graph* (RRGraph), which causes a large code complexity. For instance, to get the switch information of a node in the RRGraph, current data structures `rr` node requires a set of utility functions as the data is stored in other data structures. In addition, to traverse the graph, users have to visit at least two data structures, which are `rr` node and `rr` edge. To overcome the limitations, a new class RRGraph has been

implemented, which includes nodes, edges, switches and related information in a unified data structure. This allows us to simplify the inputs of VPR routers as well as consist of a good base to modularize/standardize VPR data structures. Around the new class RRGraph, we have developed a series of functions:

3. we built a function `convert_rr_graph` to convert the old data structures to the new class RRGraph. To examine the correctness of `convert_rr_graph`, we have implemented a new writer to output a RRGraph to a XML file (standard VPR output format). By comparing the outputted XML files generated by classical and new writers, we have guaranteed the correctness of RRGraph conversion.

4. we have adapted the downstream routers, including both breadth-first router and timing-driven routers to use the RRGraph data structure. The modified VPR has passed both basic and strong regression tests, preserving the correctness as well as high-quality of results.

3.3.1.3 Netlist Compression

Previously, OpenFPGA was using single-bit ports in Verilog netlist generation, which causes large file sizes and impacts the I/O time of backend tools. We have merged these single-bit ports to buses to compress netlist sizes. For instance, the size of top-level netlist of a 2×2 FPGA drops from 15,975 to 1,683 lines, which is a 90% shrink. This can help reduce the I/O time and open more opportunities in backend strategy since modern detailed routing algorithms can optimize bus nets than single-bit nets.

3.3.2 Support for Explicit Mapping of Standard Cells

To ease the Verilog generation, our engine uses a fixed port sequence for most primitive circuit cells. However, the standard cells provided in various PDKs may follow a different port order. To be generic to any standard cell library, we have added an explicit port mapping feature in our Verilog generator. The explicit port mapping is applied to all the primitive circuits, including inverters, buffers, transmission gates, flip-flops, SRAMs and I/O pads. Fig. 37 shows the novel XML syntax to support this feature. We added the XML property dump explicit port map to switch on/off outputting Verilog netlists with explicit port mapping.

```
<circuit model type="inv buf" name="inv1x" prefix="inv1x"
      dump explicit port map="true" verilog netlist="std cell.v">
   <design technology type="cmos" topology="inverter" size="1"/>
   <port type="input" prefix="in" lib name="IN" size="1"/>
   <port type="output" prefix="out" lib name="OUT" size="1"/>
</circuit model>
<circuit model type="pass gate" name="tgate" prefix="tgate"
      dump explicit port map="true" verilog netlist="std cell.v">
   <design technology type="cmos" topology="transmission gate"/>
   <port type="input" prefix="in" lib name="IN" size="1"/>
   <port type="input" prefix="sram" lib name="SEL" size="1"/>
   <port type="input" prefix="sramb" lib name="SELB" size="1"/>
   <port type="output" prefix="out" lib name="OUT" size="1"/>
</circuit model>
```

Figure 37: XML Examples of Defining Explicit Port mapping for Basic Elements

The XML property can be applied to any primitive cell, allowing each circuit module to have a customizable port mapping. The existing XML property prefix in each port represent the port name defined in FPGA architecture and it will be used each time the circuit model is instantiated. To specify the port name defined in standard library, we added a new XML property lib name to each port. If not specified, the lib name will be same as the prefix by default. Fig. 38 shows an example where the explicit port mapping is applied to the inverters and transmission gates of a multiplexer.

```
// Structural Verilog for CMOS MUX basis module: mux 1level size2 basis
module mux 1level size2 basis (
      input [0:1] in,
      output out,
      input [0:0] mem,
      input [0:0] mem inv);
   // Structure-level description
   TGATE TGATE 0 (.IN(in[0]), .SEL(mem[0]), .SELB(mem inv[0]), .OUT(out));
   TGATE TGATE 1 (.IN(in[1]), .SEL(mem inv[0]), .SELB(mem[0]), .OUT(out));
   endmodule
// END Structural Verilog CMOS MUX basis module: mux 1level size2 basis
// CMOS MUX info: circuit model name=mux 1level, size=2, structure: one-level
module mux 1level size2 (
      input wire [0:1] in,
      output wire out,
      input wire [0:0] sram,
      input wire [0:0] sram inv);
   wire [0:1] mux2 l1 in;
   wire [0:0] mux2 l0 in;
   mux 1level size2 basis mux basis (mux2 l1 in[0:1], mux2 l0 in[0],
      sram[0:0], sram inv[0:0]);
   inv1x inv0 (.IN(in[0]), .OUT(mux2 l1 in[0]));
   inv1x inv1 (.IN(in[1]), .OUT(mux2 l1 in[1]));
   inv1x inv out (.IN(mux2 l0 in[0]), .OUT(out));
endmodule
```

Figure 38: Example of Auto-generated Verilog Netlists: a 1-level 2-input Multiplexer with Explicit Port Mapping to Standard Cells

Previously, we had limited support on the direct mapping to standard cells for the configurable memories. We have extensively enhanced OpenFPGA to support direct mapping to various standard cells as a configurable memory element. The cell list covers most variants of memory cells in *Global Foundries* (GF) 12nm PDK, as well as any other popular PDK. We can support:

- D-type flip-flop, with/without reset/set/enable signals,
- Scan-chain D-type flip-flop, with/without reset/set/enable signals,
- 6-transistor SRAM, with/without reset/set signals,
- latch, with/without reset/set signals.

3.3.2.1 Standard Cell Mapping to Multiplexers

To reduce the development risks of our test chip, we have extended our standard-cell-mapping support to 2-input multiplexers. This allows users to build any multiplexers using base standard cells (instead of custom T-gates), giving more flexibility in designing FPGA fabrics. Especially for our test chip, it can avoid a complex custom layout for one-level and two-level multiplexers, considering the high risk and complicated design rules of GF 14nm technology. Fig. 39 shows an example of XML codes that describe a standard cell MUX2. In principle, standard cells are defined in a `circuit` model in the type of `gate`. The functionality of standard cells is specified in the XML property `topology`. Fig. 39 shows an example of how to use the MUX2 cell in multiplexer `circuit` model. When the `pass gate logic` is linked to a MUX2 cell, the multiplexers will be built with MUX2 cells instead of transmission-gates/pass-transistors.

```
<!-- Define a circuit model for a standard cell MUX2 -->
<circuit model type="gate" name="MX2 X1N A9PP84TR C14" prefix="MUX2">
  <design technology structure="cmos" topology="MUX2"/>
  <input buffer exist="off"/>
  <output buffer exist="off"/>
  <port type="input" prefix="a" lib name="A" size="1"/>
  <port type="input" prefix="b" lib name="B" size="1"/>
  <port type="input" prefix="s" lib name="S0" size="1"/>
  <port type="output" prefix="out" lib name="Y" size="1"/>
</circuit model>
```

Figure 39: XML Example for Describing a Standard cell MUX2

```
<!-- Enable local encoders for multiplexers -->
<circuit model type="mux" name="mux 1lvl" prefix="mux 1lvl">
  <design technology structure="one-level" local encoder="true"/>
  <input buffer exist="on" circuit model name="INV1X"/>
  <output buffer exist="on" circuit model name="INV1X"/>
  <!-- Link to the circuit model defined in Fig. 39 -->
  <pass gate logic circuit model name="MUX2"/>
  <port type="input" prefix="in"/>
  <port type="output" prefix="out"/>
  <port type="sram" prefix="sram"/>
</circuit model>
```

Figure 40:XML Example for Describing a One-level Multiplexer using Local Encoders and Standard Cell MUX2

3.3.2.2 Local Encoder Support for Routing Multiplexers

A potential destructive risk exists during the configuration phase of configuration-chain FPGAs. It comes from DC paths through routing multiplexers, which can temporarily exists during the configuration phase. Fig. 41(a) shows an example, where the red dashed line represents such a DC path. It happens when two transmission-gates/pass-transistors are turned on at the same time, e.g., $S[0] = 1$ and $S[1] = 1$ in Fig. 41(a). Such case will only occur during configuration phase when the bitstream is loaded serially through the scan-chains, as long as that there are two adjacent 1 in the bitstream. Considering the large number of routing multiplexers, the total DC current could be high enough to destroy the chip. Therefore, we added local encoders for routing multiplexers, as illustrated in Fig. 41(b). As shown in Fig. 39, the feature can be enabled by a XML syntax `local decoder` in the `circuit` model of multiplexers. When enabled, a local encoder can be added to the multiplexer circuits defined in this `circuit` model. The local encoder can interface the control inputs and the configuration SRAMs with a one-hot encoding, guaranteeing that only a single path is open at a time. Note that for multi-level multiplexers (see Fig. 42), each level has an independent local encoder, which can keep the circuit logic as simple as those for one-level multiplexers. This will help reduce the number of SRAM cells used in the FPGA as well as configuration time (especially for scan-chain configuration protocols).

3.3.2.3 Flexible Routing Multiplexer Design

OpenFPGA allows users to define the existence of input and output buffers for routing multiplexers. This is motivated by the fact that standard/custom 2-input multiplexer cells may already contain buffers internally and not need to add input/output buffers when cascading these cells. The following XML syntax (see Fig. 43) is supported to enable/disable input and output buffers for a routing multiplexer.

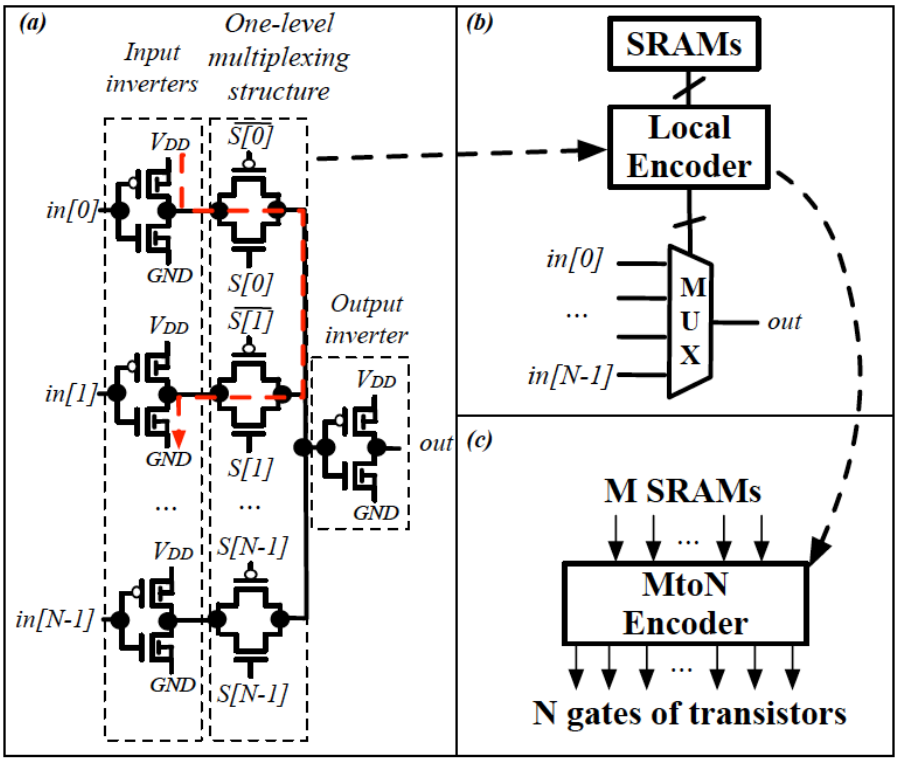


Figure 41: (a) Schematic of a One-level N-input Routing Multiplexer; (b) Block Diagram of a Routing Multiplexer with a Local Encoder; (c) Detailed Local Encoder

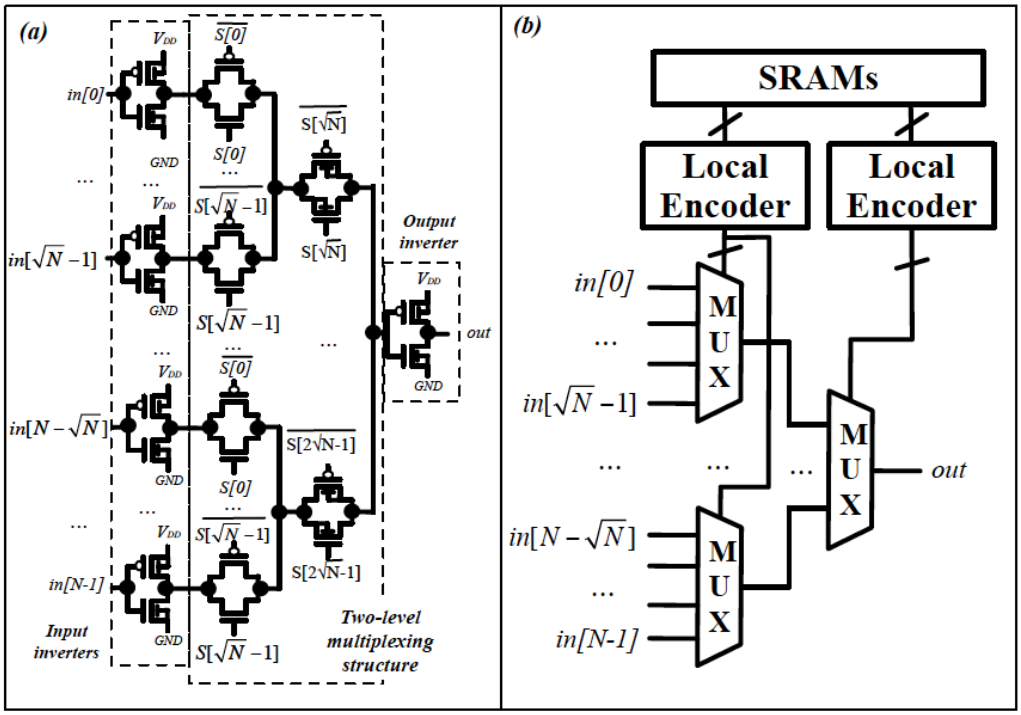


Figure 42: (a) Schematic of a Two-level N-input Routing Multiplexer; (b) Block Diagram of a Routing Multiplexer with Local Encoders

```
<!-- Flexible MUX design in XML -->
<circuit model type="mux" name="mux 2level" prefix="mux 2level">
  <design technology type="cmos" structure="multi level" num level="2"/>
<!-- Input buffers are enabled -->
  <input buffer exist="true" circuit model name="buf4"/>
<!-- Output buffers are enabled -->
  <output buffer exist="false"/>
  <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>
```

Figure 43: Examples of Input/Output Buffer Customization in Routing Multiplexers

Multiplexer Netlist Reorganization: Now the Verilog netlists of routing multiplexers are split into two separated netlists [21]:

5. muxes.v, that contain the top-level modules, and
6. mux primitives.v, that contain the primitive modules to built the top-level multiplexers.

We have also improved the code generator to enable top-level modules share primitives as much as they can [22]. We have seen a significant reduction on the netlist size for our Skywater tape-outs (mux primitives.v is reduced by 50+% from 1,000 lines to 400 lines). This also eases the integration of custom cells since the netlist organization is simpler.

3.3.2.4 Extended Global Signal Support

We have also enhanced the support on global signal customization, especially for clocks, during Verilog netlist generations. Previously, OpenFPGA created feedthroughs for all the global signals, as illustrated in Fig. 44(a). This caused a strong limitation in supporting programmable clock network, which can be customized in VPR's FPGA architecture description language. Therefore, we now allow users to connect global signals to the input ports defined in VPR architecture file. Fig. 44(b) depicts an example where a programmable local clock network can be customized inside a tile and a global clock signal is connected to each tile. The enhancement enables a basic multi-clock support, where clocks are either driven by the global clock signal or a local signal inside a tile (driven by a LUT or other circuit elements).

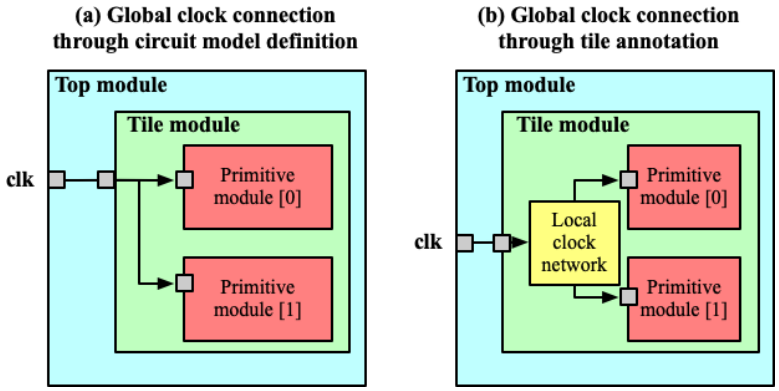


Figure 44: Difference Between Global Port Definition Through Circuit Model and Tile Annotation

3.3.2.5 Extended I/O Support

To provide native EDA support for the Skywater tape-out, we have upgraded OpenFPGA to support versatile I/O cells in embedded FPGAs. Beyond the classical GPIO cell, OpenFPGA now supports input buffer, output buffer and tri-state buffers as I/O. This have covered all the features on I/O cells required by QuickLogic eFPGA devices and the Caravel SoC [23].

Report I/O mapping: Driven by Quicklogic's needs for verification and chip testing, we realized that a file containing I/O mapping information can ease the debugging significantly, particularly, which primary input/output of a HDL design is mapped to which FPGA I/O. Therefore, OpenFPGA has been upgraded to report the I/O mapping into an XML file. See details in [24].

3.3.2.6 Support of I/O Grids in the Center Part of FPGA

In the short term, this feature [25] enables QL to generate Verilog netlists for their AP3 architecture. See an illustrative example in Fig. 45(b). In the long run, this feature allows us to develop embedded FPGA IPs compatible with flip-chip technology and 3D packaging options.

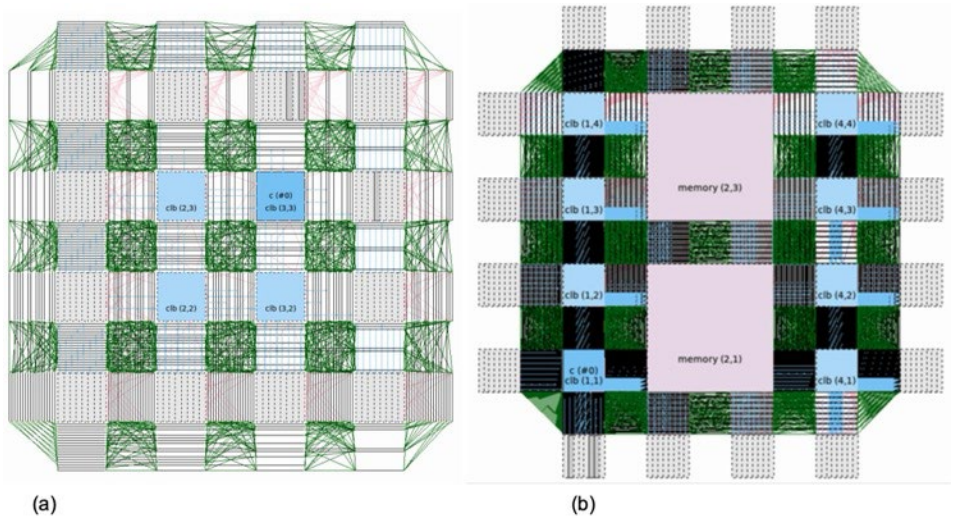


Figure 45: (a) an Example of an FPGA Architecture Where I/O Grids are in the Center Part of FPGA Fabric, sur Rounded by the Routing Architecture; (b) a Standard FPGA Architecture where I/O are on the Border of FPGA without Surrounding Routing Architecture

3.3.3 Support of Enhanced LUT Designs

We enhanced the LUT design support for versatile LUT circuit designs. We previously supported standard fracturable LUT design [26], where a fracturable LUT contains internal configurable memories to switch between operating modes, e.g., dual LUT5 and quad LUT4. Driven by QuickLogic's needs, we now support native fracturable LUT designs [27], where a fracturable LUT relies on external signals (VDD/GND) to switch between operating modes. To help users understand the difference, we have added illustrative figures to online documentation about the versatile LUT design topology. Fig.46 compares the schematic-level difference between the standard and native fracturable LUT designs.

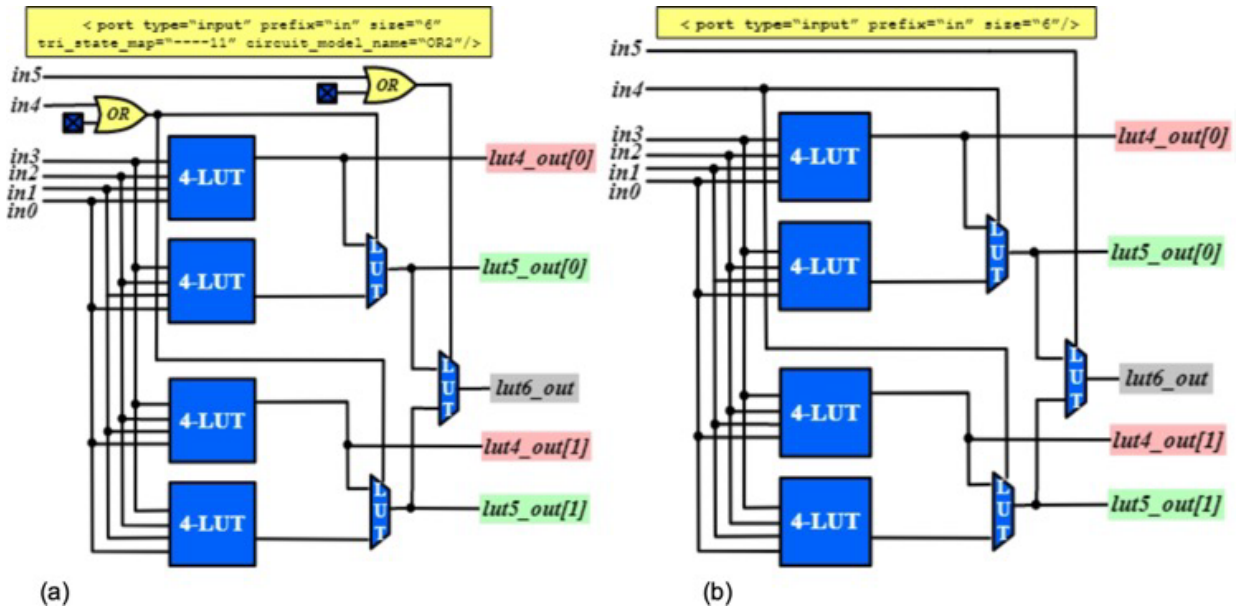


Figure 46: (a) a Standard Fracturable LUT Design where Internal Configurable Memories and OR Gates are used to Switch Between Operating Modes; and (b) a Native Fracturable LUT Design which Operating Modes Switching are Determined by External Signals at in5 and in

3.3.3.1 Logic Element Enhancement

We created and tested architectures to support memories as logic element's mode. These logic elements are known as LUT-RAM or SLICEM and supported by commercial FPGAs. Fig. 47 shows an example of an architecture using LUT4. Already tested and supported modes are shown in green, while in orange is an example of a mode under development using more than one logic element.

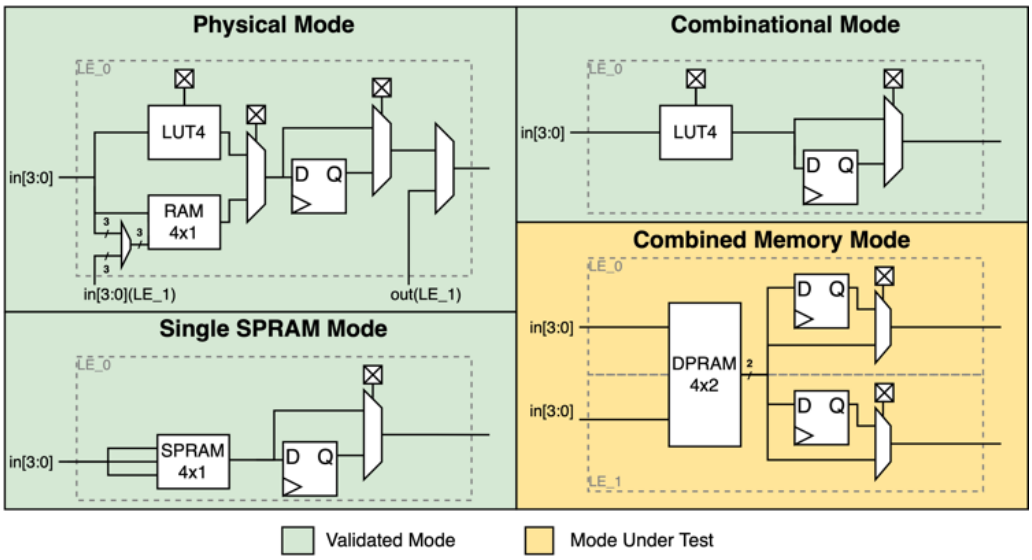
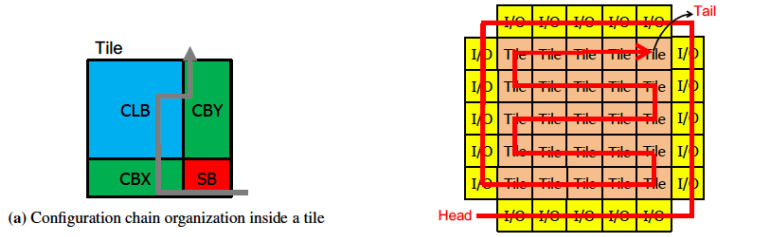


Figure 47: Example of LUT-RAM Architecture with a Multi-logic Element Mode

3.3.4 Configuration Protocols

We have improved the configuration chain organization to be more friendly to PnR. As a result, OpenFPGA can automatically generate the bitstream used to program the configuration circuits, according to the selected implementations.

3.3.4.1 Scan-chain Configuration Protocol



(a) Configuration chain organization inside a tile

(b) Configuration chain organization in full FPGA fabric

Figure 48: Improved Configuration Chain Organization

Configuration chain can first pass through all the routing, logic and I/O blocks. This indeed challenged our PnR process, where the configuration chain contained many long metal wires when across different type of resources. This can introduce large parasitics and thus slow down configuration speed. To speed up that configuration, Fig. 48 depicts the scan-chain flip-flop organization, where the configuration bits can go from tile to tile. Inside each tile, the configuration chain will always start from a *Switch Block* (SB), go through a *X-direction Connection Block* (CBX), a *Configuration Logic Block* (CLB), and end to a *Y-direction Connection Block* (CBY). Therefore, the scan-chain is tightly organized across tiles, and thus eliminates all the long metal wires and significantly speed up configuration clock frequency. Moreover, the configure-enable signal activates the data output of the configuration chain when the configuration is done. This feature allows users to avoid having to constantly change the configuration of programmable resources during bitstream loading, as illustrated in Fig. 49 and integrated to the project [28].

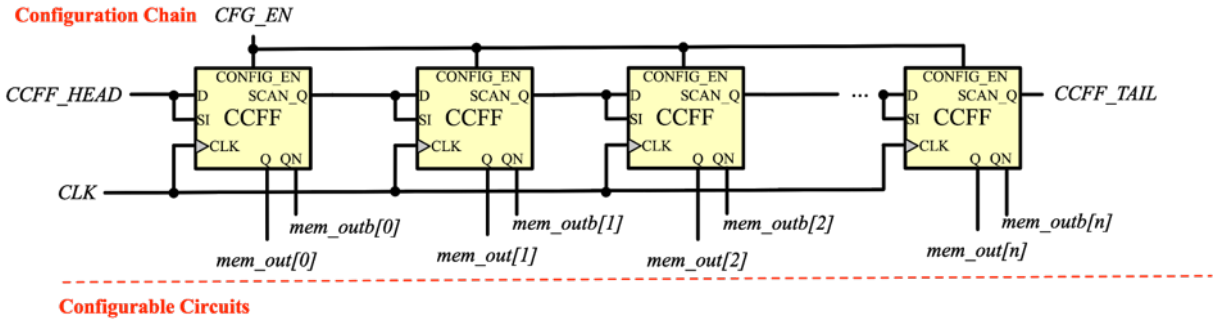


Figure 49: An Illustrative Example of a Configuration Chain using Configure able Signal (CONFIG EN) and Separated Data Output to Drive Memory Elements and Datapath Circuits

3.3.4.2 Memory-based Configuration Protocol

As the memory decoders require SRAMs rather than scan-chain flip-flop, we added a new XML syntax to describe the ports of SRAMs, which enables a direct mapping to standard cells. As illustrated in Fig. 50, SRAM cells belonging to the same row share a BL signal while each column is controlled by a WL signal. All the BL and WL signals are controlled by two decoders. Each SRAM cell can be individually programmed when its associated BL and WL are enabled by manipulating the two decoders. The XML snippet in Fig.51 corresponds to the SRAM implementation described in Fig. 50(c). In addition to this example, our XML syntax also supports the inverted WL port, which can be described by defining a port type of wlb. The XML syntax `default val` is here to indicate to the bitstream generator which digital signals will be forced to the BL and WL ports when configuring a SRAM. In the example, each BL signal will keep at logic 0 unless a SRAM configuration is required.

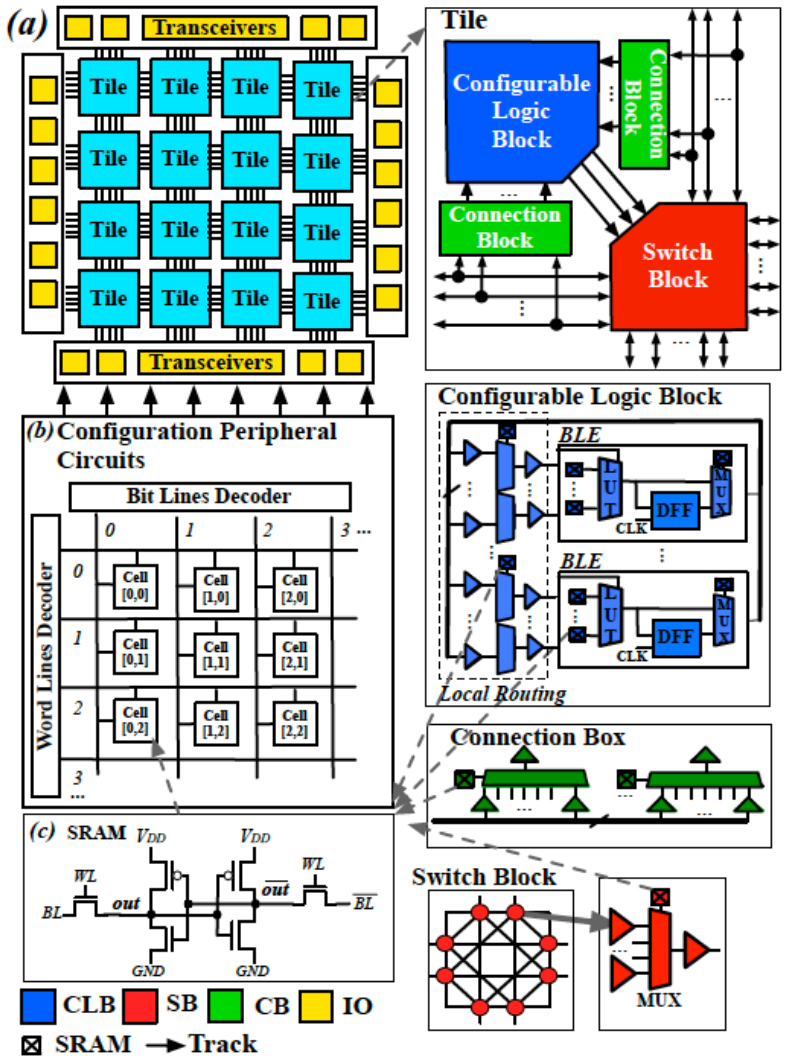


Figure 50: FPGA Architecture with Address Decoders to Configure SRAMs
(a) tile-based FPGA organization; (b) memory decoders and SRAM cells; (c) transistor-level schematic of a 6-transistor SRAM.

```
<!-- Declare a 6-transistor SRAM module with BL and WL ports-->
<circuit model type="sram" name="sram6T" verilog netlist="sram.v">
<!-- Declare regular output ports -->
  <port type="output" prefix="out" lib name="Q" size="1"/>
  <port type="output" prefix="outb" lib_name="QB" size="1"/>
<!-- Declare BL and WL ports -->
  <port type="bl" prefix="bl" size="1" lib name="BL" default val="0"/>
  <port type="blb" prefix="blb" size="1" lib name="BLB" default val="1"/>
  <port type="wl" prefix="wl" size="1" lib name="WL" default val="0"/>
</circuit model>
```

Figure 51: XML Syntax for the SRAM Configuration Circuit

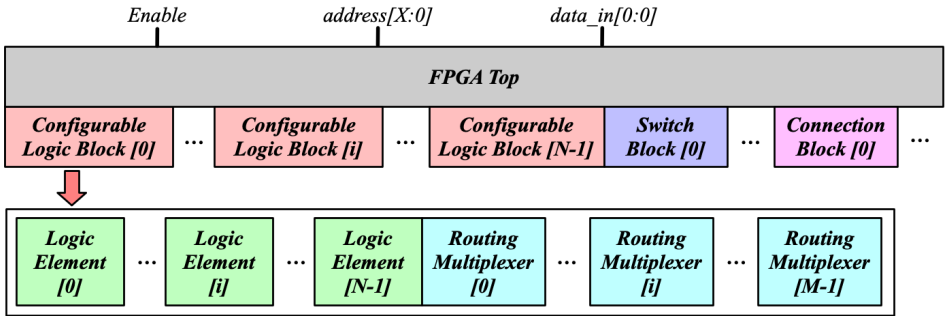
As memory bits of FPGAs can be accessed by different types of configuration circuits, leading to difference in the full-chip area and also other merits. The architecture description language is also extended to model the configuration circuits. Fig. 52 describes the configuration circuits specified for Verilog and Bitstream generator under the XML node sram. In the example, the XML tag verilog specifies the type of configuration circuit (XML property organization). The supported configuration circuits include memory-bank style (XML syntax memory-bank) and scan-chains (XML syntax scan-chain). The memory model accessed by the configuration circuits can be declared in the XML property `circuit` model name, which is linked to a defined `circuit` model devoted to the transistor-level designs of a SRAM and a scan-chain flip-flop.

```
<!-- Specify the use of memory decoders to configure SRAMs -->
<sram>
  <verilog organization="memory-bank" circuit model name="sram6T"/>
</sram>
```

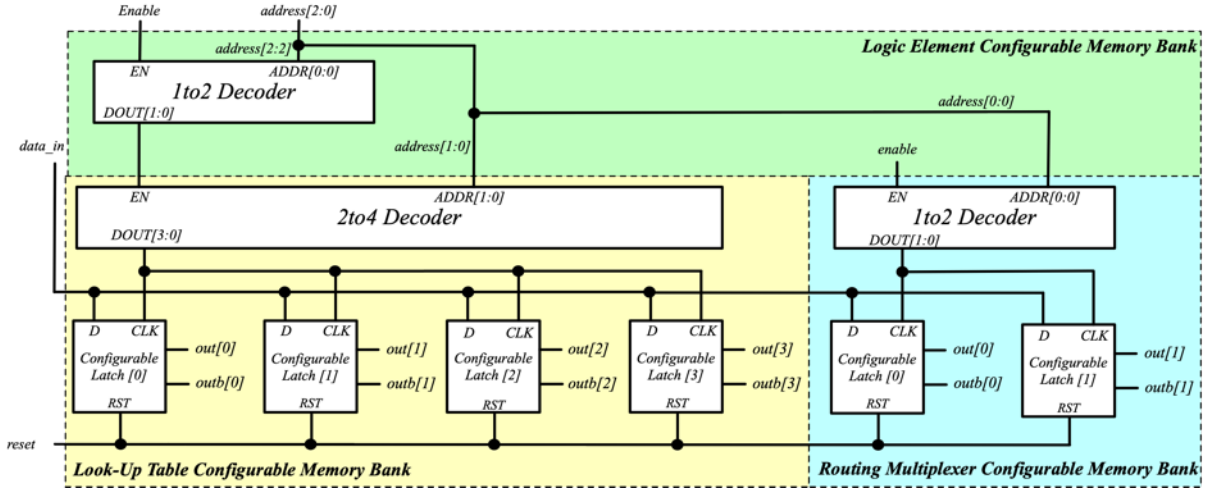
Figure 52: XML Syntax for the SRAM Configuration Circuit for Bitstream Testing

3.3.4.3 Frame-based Configuration Protocol

Configurable memories are organized by frames, as illustrated in Fig. 53. 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 applied hierarchically. Each frame may consist of a number of sub frames, each of which follows the similar organization.



(a) Hierarchy of the frame-based configuration protocol



(b) An example of frame-based configuration protocol

Figure 53: Frame-based Configuration Protocol

3.3.4.4 Multi-region Configuration Protocol

OpenFPGA supports a single configuration chain across the fabric (see Fig. 54(a)), where the full bitstream has to be loaded serially (1 bit per clock cycle). Many drawbacks have been seen during our tape-out practice, e.g., (1) challenging the timing signoff especially hold time violation for large FPGAs, (2) long verification time, which may take days. Therefore, we enhanced this configuration circuitry by supporting multiple heads, where bitstream can be loaded in parallel per clock cycle, as illustrated in Fig. 54(b). Moreover, the multi-head configuration chain is fully customizable.

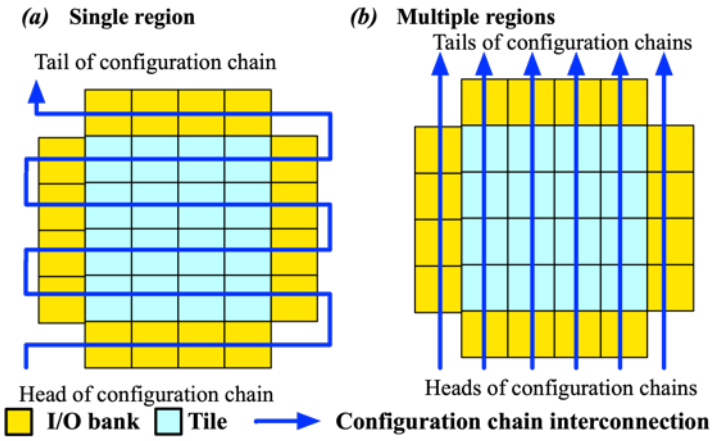


Figure 54: Flexible Configuration Chain Support

Configurable memories are organized in an array, where each element can be accessed by an unique address to the BL/WL decoders, as illustrated in Fig. 55. We have extended the multi-region configuration support to the two configuration protocols, where a bitstream can be loaded to each region in parallel per clock cycle, as illustrated in Fig. 55(b). Similar to the multi-head configuration chain, the multi-region memory bank and frame-based configuration protocols are customizable through the XML syntax [29] and the fabric keys [30].

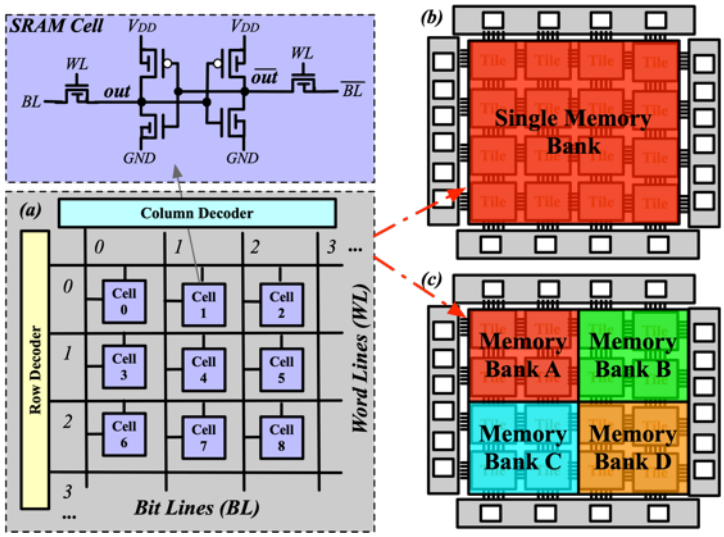
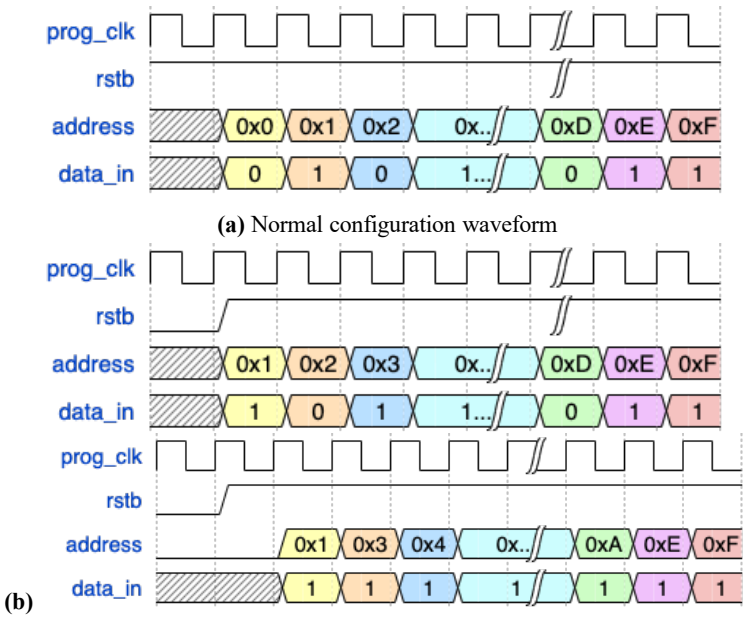


Figure 55: Example of (a) a Memory Organization using Memory Decoders; (b) Single Memory Bank Across the Fabric; and (c) Multiple Memory Banks Across the Fabric

3.3.4.5 Fast Configuration-chain Support

The fast configuration support has been extended to configuration chain protocol, which was previously applicable to only the memory bank and frame-based protocols. When enabled, the Verilog testbench will skip all the zeros at the beginning of the bitstream when loading to FPGAs. Note that in this case, the configurable memories must have reset functionality. It can also skip all the ones at the beginning if the configurable memories have set functionality. As a result, we see a 15%-92% simulation runtime reduction depending on architecture and

implemented circuits thanks to the fast configuration technique, the verification time is now ~30 minutes at post-PnR with SDF, which was previously ~9 hours. The fast simulation can be executed before a full-coverage simulation, as a quick sign-off method. It helps us to spot problems early during back-end and iterate faster.



(a) Normal configuration waveform

(b)

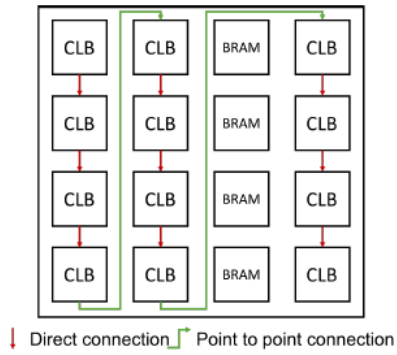
(c) Fast configuration waveform: heading logic '0' is skipped

(d) Fast configuration waveform: logic '0' is skipped

Figure 56: Configuration Wave-forms

3.3.5 Inter-CLB Connection Enhancement

To implement a test scan-chains across the test chip, we have enriched the direct connection patterns across CLBs. VPR's support on inter-CLB direct connection is limited to CLBs in the same column or row, highlighted red in Fig. 57. However, to enable superior Design-For-Test, our test chip requires a scan-chain going across all the CLBs, which was beyond the capabilities of VPR. Therefore, we have added a new XML syntax to the "direct" connection definition in the VPR architecture file (see Fig. 58). Our goal is to support cross-column/row inter-CLB direct connection (highlighed green in Fig. 57(b)) in the architecture XML language and the Verilog code auto-generation. Through keywords `interconnection type`, `x dir` and `y dir`, users can define eight different connecting patterns, covering all the directions in cross-column/row connections between CLBs. A detailed truth table is listed in Fig. 57(b). So far, our Verilog generator has been upgraded to auto-generate these direct connections. This feature is used in our test chip to create a scan-chain for testing. Verification over the scan-chain has been done successfully. This feature has been pushed onto our GitHub branch master.



(a) Supported inter-CLB connecting patterns.

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

(b) Truth table of XML syntax to define the cross-column connecting patterns.

Figure 57: Detailed Inter-CLB Connecting Patterns

```
<!-- Define a cross-column inter-CLB connection -->
<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>
```

Figure 58: XML Example for Describing an Inter-CLB Connection

3.3.6 Tileable Routing Architecture Support

To achieve million-of-LUT FPGAs (and match large commercial products), the routing architecture should be tileable. Such a functionality also allows us to fully exploit the advantage of hierarchical P&R flows. The coming release of OpenFPGA already supports tileable CLBs, which are the same across full FPGA fabrics. However, as stated on VPR documentation, the routing architecture supported by VPR is not tileable[2]. This does block us to achieve a tile-based FPGA as shown in Fig. 50. To study the difference of routing architecture, we have implemented an analysis tool in OpenFPGA, which compares the connection blocks and switch blocks. On the Stratix-4 like FPGA (20×20) with Length-4 routing wires, we find that 57 of 441 switch blocks are unique, while 30 of 289 connection blocks are unique. Our results showed that the P&R flow can be significantly simplified if we only output the unique blocks in Verilog and instantiate them across the fabric. All the features are available on the feature branch `tileable routing`, which can be turned on by an option `--compact routing hierarchy`.

We have upgraded VPR with a tileable *Routing Resource Graph* (RRG) generator, which can generate regular routing architecture for both homogeneous and heterogeneous FPGAs. To validate the regularity of our tileable RRG, we use graph matching to identify the replicated CBs and SBs in both tileable and academic architectures. For a fair comparison, we consider only tileable array sizes for both architectures, ranging from 16×16 to 96×96. Table 14 compares the number of unique tiles in both architectures. Even though the academic architecture from VPR is generally not tileable, there are still many tiles which are repeatable. However, the number of

[2] http://docs.verilogtorouting.org/en/latest/arch/reference/?highlight=tileable

unique tiles remain too large, requiring complete manual layouts for a $\sim 9 \times 9$ array. Note that FPGA regularity could be much worse, when longer wire segments (> 16) are introduced. In contrast, the tileable architecture has a constant number(= 9) of unique tiles, whatever the array sizes are. The results prove the proposed RRG can minimize the number of tiles to be laid out, showing an improvement of $9.3\times$ compared to academic architectures. To propose an apple-to-apple comparison, we compare the layout area of the two architectures using a synthesizable FPGA flow. The array sizes are fixed to 32×16 which is minimum tileable array size that fit the largest MCNC big20 benchmark. The academic FPGA consumes $14.016 mm^2$ while the tileable FPGA requires $14.008 mm^2$, showing the same area efficiency.

Table 14: Number of Unique tiles in FPGAs

. of Unique Tiles	Homogeneous		Heterogeneous	
Array Size	VPR	Tileable	VPR	Tileable
16 ×16	80	9	148	27
32 ×32	84	9	174	27
64 ×64	84	9	178	27
128 ×128	84	9	174	27

More than tileable FPGA, our RRG generator also supports different switch block patterns for (a) the routing tracks that start/end in a tile and (b) the routing tracks that pass a tile. To customize the switch block patterns for passing tracks, we added two new XML tags to the VPR architecture description language, as shown in Fig. 59. The XML property sub type specifies the interconnection pattern for passing tracks, while sub fs denotes the connectivity of passing tracks. Supporting the most popular patterns, type and sub type could be any combination of Subset, Universal and Wilton. In addition, sub fs can be different than the connectivity of starting/ending tracks fs, enabling a larger architecture exploration space than before. Fig. 60 depicts the corresponding switch block in an illustrative case, where uni-directional length-2 wires are interconnected in a routing channel width of W = 4. Note that when the length of wires is larger than 2, the number of passing tracks will be much larger than the number of start tracks in a switch block. Our RRG generator uses a similar round-robin scheme than VPR, in order to balance the multiplexer size.

```
<!-- Switch block with a mix of Subset and Universal patterns -->
<switch block type="subset" fs="3" sub type="universal" sub fs="3"/>
```

Figure 59: Examples of Extended XML Syntax for Switch Block Patterns

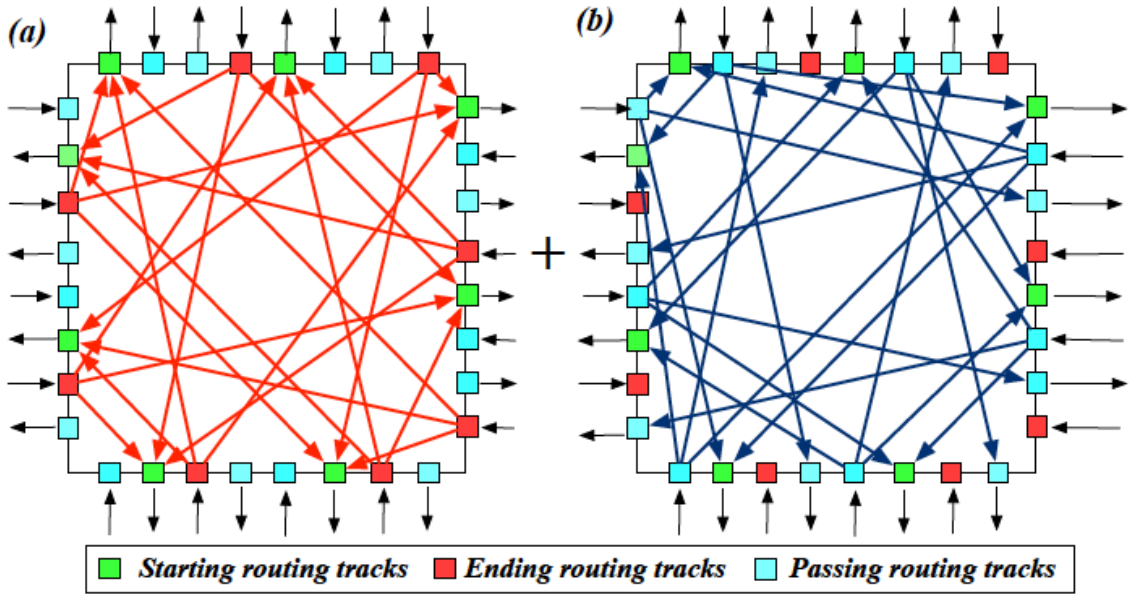


Figure 60: An Illustrative Example of Mixed Switch Block Patterns
(a) Starting/Ending Tracks Follow the Subset Pattern; (b) Passing Tracks Follow the Universal Pattern

We evaluate the performance of mixed switch blocks patterns in the context of a Stratix IV-like FPGA architecture, by considering the most representative patterns, i.e., Subset [31], Universal [32], Wilton [33] and Imran [34]. Fig. 61 compares the area, delay and minimum routing channel width Wmin between the academic and tileable architectures. In academic architecture, Wilton and Universal patterns are the best choices, being 2% smaller in Wmin and 1% better in area-delay product than Subset. The conclusion is consistent with previous works and also explains well why Wilton SB is popular in many FPGA research papers. Differently, in tileable architecture, Subset patterns are the most routable but with a 3% overhead in area-delay product than the baseline. When considering mixed switch block patterns, using Universal for starting/ending tracks and Subset for passing tracks (Universal×Subset) leads to the best area-delay product. On average, the tileable FPGAs improve 13% in Wmin than the academic counterparts. This is due to that the passing tracks use Subset/Universal/Wilton are more routable than those using round- robin schemes. When compared to the best academic architecture, the Universal×Subset tileable FPGA has a 2% area-delay product improvement, showing the promise of tileable architectures.

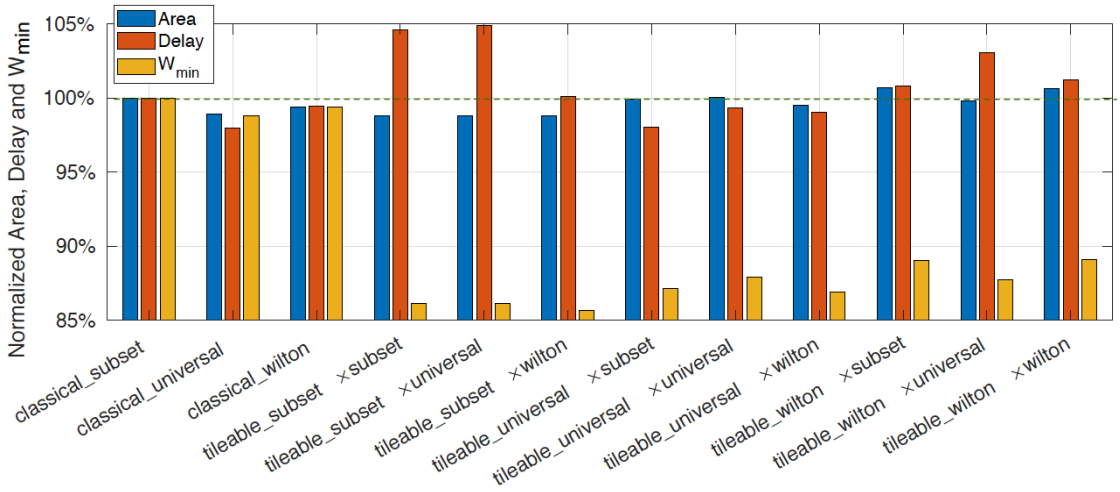


Figure 61: Area, Delay and Channel Width Comparison between Academic and Tileable FPGAs using different Switch Block Patterns Averaged over MCNC big-20 and VTR Benchmarks

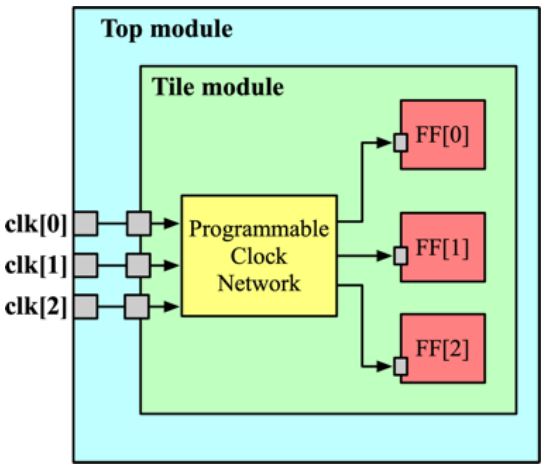


Figure 62: An Illustration of Multi-clock Implementation in FPGA Fabric

3.3.6.1 Consistent port naming with architecture description

In this enhancement, the naming of grid module pins is changed to be closely related to architecture definitions. For instance, previously, the name of an input I[0:0] in CLB module was named as top width 0 height 0 pin 0 . Currently, the name of a input I[0:0] is named as top width 0 height 0 subtile 0 pin I 0 , which is easier for chip designers to traceback architecture XML changes in Verilog netlists. See details in [35].

3.3.7 Enriched Heterogeneous FPGA Architecture Examples

Thanks to VPR8, OpenFPGA now supports more versatile heterogeneous FPGA architectures. Therefore, we have added the following architecture examples to our portfolio on GitHub. Note that these features are very useful in modern FPGA architectures which are typically pad limited and integrate hard IPs.

1. Different I/O capacities on each side of a FPGA fabric. For instance, each I/O block at left side contains 2 GPIO cells while each I/O block at left side contains 4 GPIO cells. Example architecture is available at [36].
2. I/Os only appear on specific sides. For instance, only top and bottom sides of an FPGA have I/O blocks. Example architecture is available at [37].
3. AIB interface replace I/Os at a side of FPGAs. Example architecture is available at [38].
4. Wide BRAM each of which spans two columns of tiles. Example architecture is available at [39].

The Yosys synthesis script of OpenFPGA has been upgraded to infer multipliers and block RAMs from input HDL designs. The inference on multipliers and block RAMs is based on the available programmable resources in FPGA architectures, e.g., 18-bit multiplier and 8k-bit RAM. The Yosys synthesis script has been tested on the VTR benchmarks as well as a newly added micro benchmark 8-bit multiply-accumulate function to validate bitstream generation. See details in [40].

3.3.7.1 Packable Mode of pb_type

This is an addon feature we have developed in the VPR8 of OpenFPGA. Designers will describe physical implementation of logic blocks in VPR architecture XML, which are not packable. The packable option will allow packer to skip the physical implementation so that VPR packer will not waste runtime on unpackable parts of logic blocks. Documentation has been updated on this feature [41].

3.3.7.2 Versatile Multi-mode Flip-Flop Support

To match the state-of-the-art FPGA architectures, OpenFPGA supports flip-flop designs that can operate in different modes. For example, a flip-flop can be reset with an active-low or active-high signal. Using the versatile flip-flop design, we see a reduced number of LUTs required (some logic function can be absorbed in the flip-flop) in practical benchmarks, which typically require many different types of flip-flop designs. See details in [42].

3.3.7.3 Multi-clock support

We have developed the multi-clock support in OpenFPGA. New XML syntax has been introduced to allow users to define any number of clocks in the FPGA architecture. Fig. 62 shows an illustrative example where a programmable clock network can be defined inside a tile, using the VPR architecture description language. The bitstream generator has been upgraded to accept pin constraints from users so that clock nets can be mapped to the proper clock pins. The Verilog testbench generator has also been upgraded to simulate a multi-clock FPGA fabric using different clock frequencies. Full details are available in the documentation [43–45].

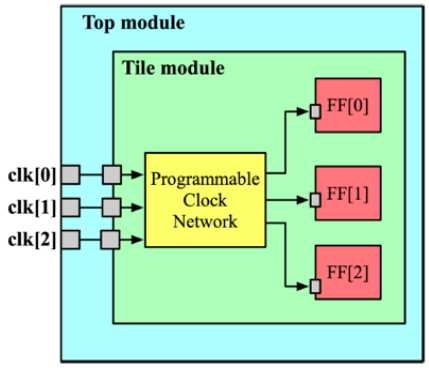


Figure 63: Comparison on Area Per Tile

3.3.8 Hardware Performance Gap

We performed a detailed comparison on the hardware performance between OpenFPGA and previous works [46, 47] (see [46] which is [9] in Fig. 63).

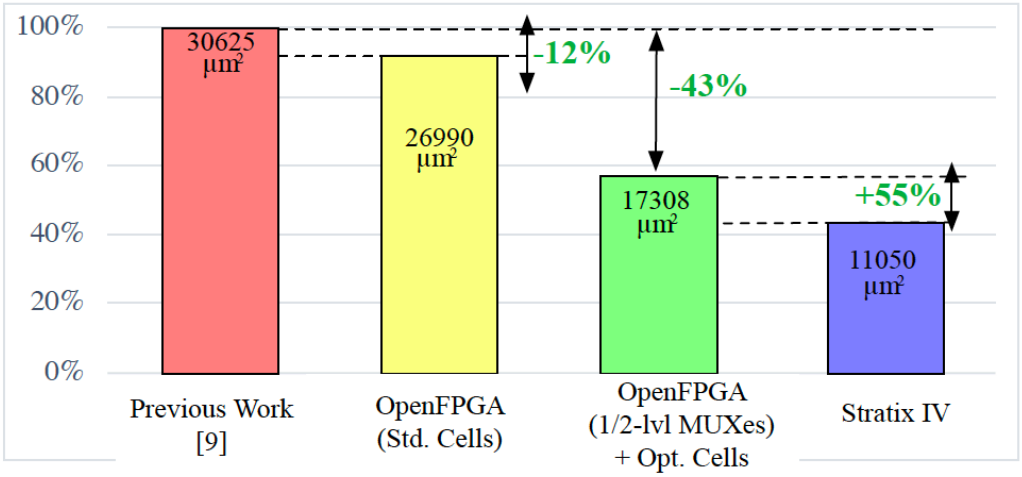


Figure 64: Comparison on Area Per Tile

3.3.8.1 Tile Area

We compare in Fig. 63 the tile area (including a CLB, two CBs and one SB) of OpenFPGA layouts with previous academic results and commercial products. To study the impact of cell libraries and multiplexing structures, we generate two additional layouts: (1) a standard-cell-based layout like [46] but using OpenFPGA. (2) an optimized layout where multiplexers and SCFFs are implemented with our customized cells shown in Fig. 64. Note that the reproduced layout is 11% smaller than the previous work [46]. This can be explained by a high density of our layout, where our utilization rate reaches 90% in P&R while previous work can only achieve 80%. Our results show that to match commercial state-of-the-art, the cell library has to be optimized for SCFF, one-level and two-level multiplexing structures, which can further reduce the tile area by 42%. Thanks to the compact multiplexing structure, our FPGA, even designed with a semi-custom approach, has only a 60% overhead in tile area when compared to a well- optimized commercial product. Note that, the area difference is partially due to the use of transmission

gates rather than pass-transistors, which are naturally larger in area. We considered transmission gates as they are potentially faster, are well suited to the standard cell methodology and more robust at advanced technology nodes [48].

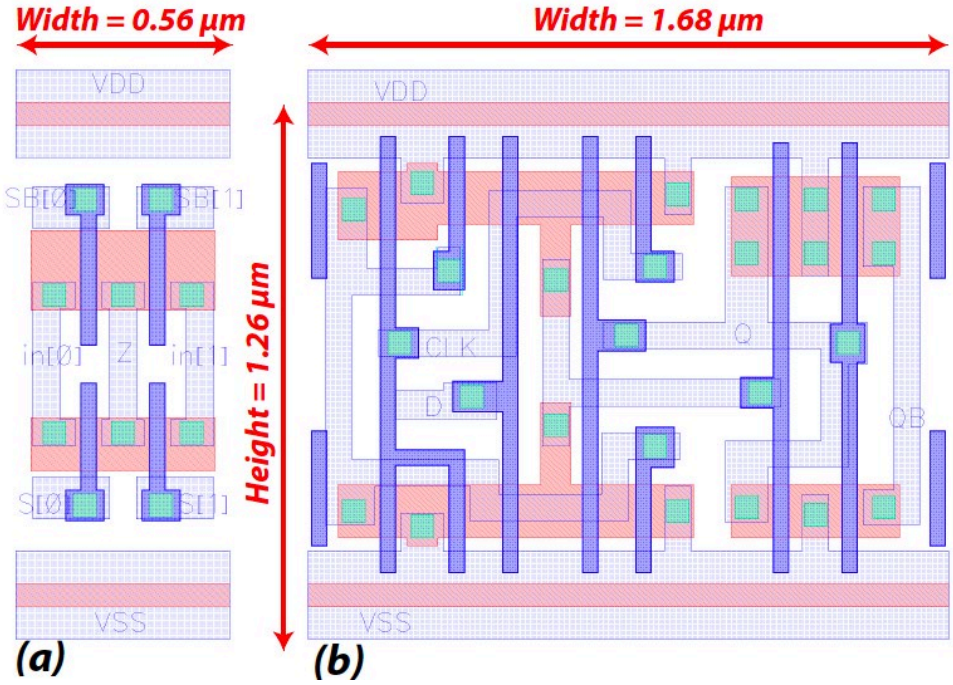


Figure 65: Full Custom Layout View of: (a) 1-level 2-Input Multiplexer; (b) SCFF

3.3.8.2 Path Delays

In Table 15, we compare the delay of critical routing paths in FPGA architectures. Each critical path of any implementation is a combination of the routing paths, the delay comparison showed the intrinsic performance gap between FPGAs. Our results prove the necessity in using one-level and two-level multiplexing structures as well as an optimized cell library. The performance of our FPGA can indeed be close to commercial FPGAs with an average difference of 30%. When compared to the best available academic results, the delay reduction is 3 × on average. Note that advanced multiplexing structures and layouts impact the delay in a two-fold way. First, the performance of one-level/two-level multiplexers is naturally higher than the tree-like in particular for large input sizes. Second, the tile area can be significantly reduced, leading to a shorter metal length between two SBs. As a result, the delay of long routing tracks can be further improved.

Table 15: Path Delay Comparison between Previous Works [46], OpenFPGA and Stratix IV

Path Type Delay unit: *ns*	Previous Work (TT) [46]	OpenFPGA (TT)	OpenFPGA (SS)	Stratix IV [47]
5-LUT	0.46 (+70%)	0.14 (-48%)	0.26 (-3%)	0.27 (100%)
6-LUT	0.5 (+78%)	0.15 (-46%)	0.27 (-3%)	0.28 (100%)
1-bit Adder[1]	0.7 (-9%)	0.54 (-10%)	1.00 (+30%)	0.77 (100%)
20-bit Adder[1]	1.63 (+32%)	1.10 (-6%)	2.12 (+72)%	1.23 (100%)
Local Routing[2]	0.27 (+58%)	0.12 (-30%)	0.23 (+35%)	0.17 (100%)
L-4 track[3]	2.53 (+328%)	0.40 (-32%)	0.75 (+27%)	0.59 (100%)
L-16 track[3]	4.02 (+294%)	0.78 (-1%)	1.55 (+52%)	1.02 (100%)
Average Diff.	+122%	-25%	+10%	100%

[1] Adder paths start from a CLB input and end at a CLB output with local routing included.
[2] Local routing path starts from a BLE output and ends at a BLE input.
[3] Track delay starts from a CLB output and ends at a CLB input.

3.3.9 Hardware+Software Performance Gap

We also compared the delay of our optimized FPGA fabric to previous works and Stratix-IV. As shown in Fig. 65, benchmark implementations on our FPGA fabric (SS) benefit from an improvement of 39% on speed than the academic FPGA (TT). In this case, as previous works use the same CAD tool as OpenFPGA, the performance gain results from the efficient hardware. When compared to the commercial FPGA, the performance gap is on average 1.75×. The gap comes from sources: (1) our hardware lags in performance with an average of 30%. When critical paths consist of multiple routing paths listed in Table 15, the delay difference will aggregate. Therefore, the longer the critical path is, the larger the performance gap will be. (2) previous studies have shown a large gap between VPR CAD algorithms and commercial counterparts [49]. The performance gap may be as large as 55% on average, fully shadowing any efficiency on hardware. This indicate that developing efficient CAD algorithms that can match industry quality should be a frontier for the open-source FPGA research community.

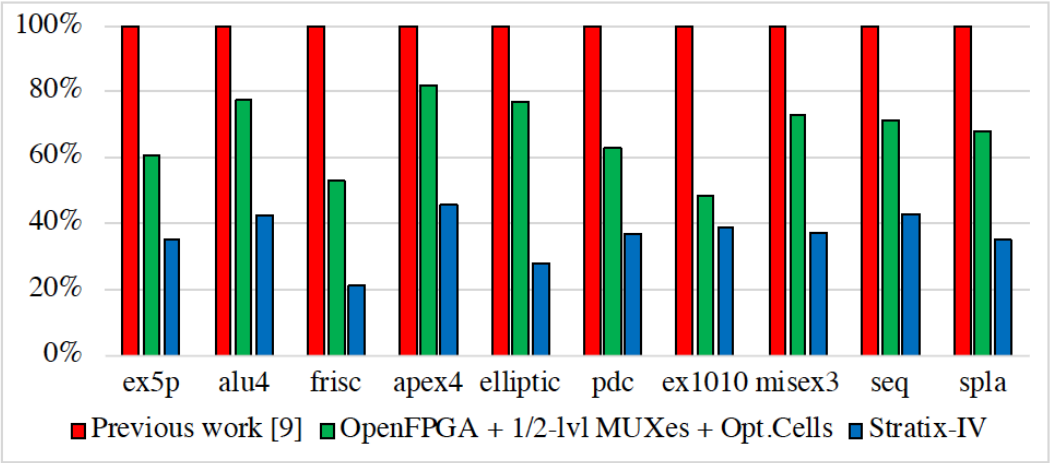


Figure 66: Delay Comparison between OpenFPGA and Previous Works [46] using Selected MCNC Benchmarks

3.4 Limitations Seen in FPGA Back-end Practice

During the development, we noticed that the number of timing violations increases as FPGA size increases due to the large clock network to be routed in the top-level. Even though the violations can be fixed manually, we were looking for a cleaner PnR strategy to avoid these violations. In addition, as the number of clock sinks increases exponentially with FPGA array size, the runtime of Clock Tree Synthesis (CTS) increases explosively, which could be 12+ hours and even never finished for large FPGAs, dominating the backend runtime. Fig. 66 shows that the physical design of a 1500-LUT FPGA cannot be finished due to the exponentially growing complexity in the top-level PnR. Therefore, we have developed several improvements to address the issues, as introduced in Section 3.6.2 and Section 3.6.3.

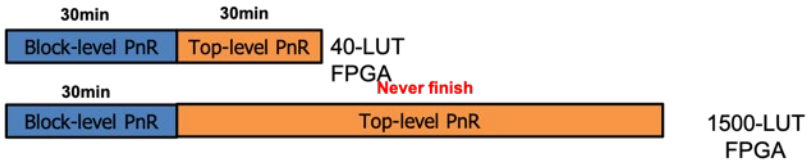


Figure 67: Exponentially Increasing Complexity in Top-level PnR

3.4.1 Frame View-based Top-level PnR

We now use the frame view of CLB/SB/CB modules in the top-level PnR, which are represented in `.lib` and `.lef` files, as illustrated in Fig. 67. Previously, our hierarchical flow import full databases of these modules to the top-level PnR, which causes high complexity in the *Clock Tree Synthesis* (CTS) and *Static Timing Analysis* (STA). Using the frame view, we only expose necessary module-level information to the top-level CTS and STA and hence accelerate the runtime. In addition, we also reduce the number of nets to be routed in the top-level, by applying an abutment-based floorplanning (see Fig. 68). The elaborated floorplanning allows blocks to be placed tightly and hence eliminate the need to connect the pins between adjacent blocks.

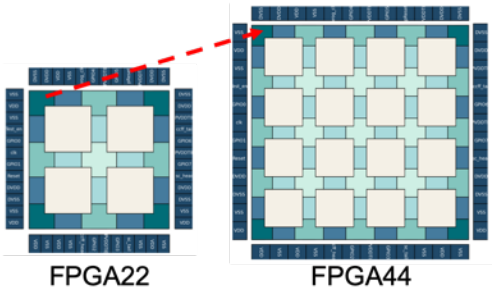


Figure 68: Reuse Block-level PnR Results as a Frame-view in Large FPGA Fabrics

Using the frame views, we have scaled up our PnR strategy for larger FPGA sizes, e.g., 12×12 and 20×20. Our practice showed that the runtime of a 12×12 FPGA takes only 3 hours while the PnR was not finished before due to the complexity of CTS. However, due to the synthesized CTS algorithms, the clock network is not organized as a H-tree or other regular topology, and this causes a large clock variation even in neighbouring blocks. Fig. 69(a) illustrates the clock latency distribution of the 12 × 12 FPGA.

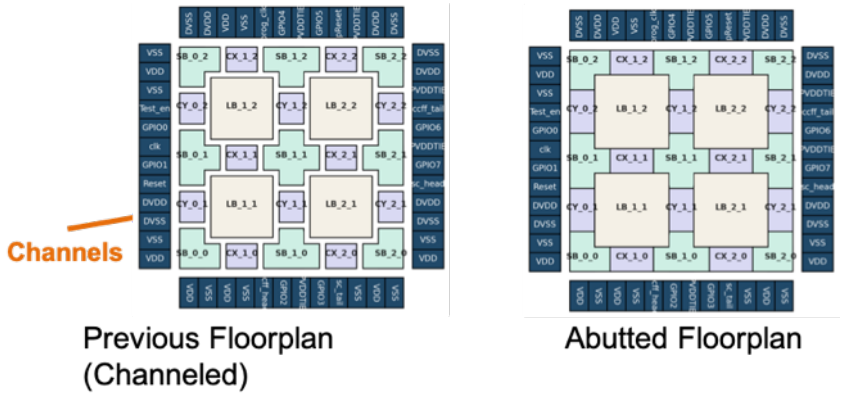
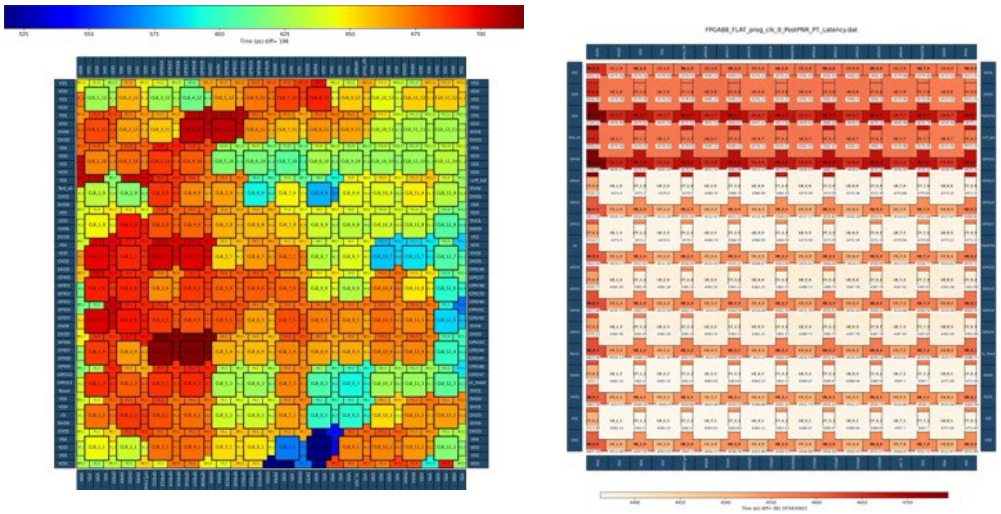


Figure 69: Improvement in Floorplanning the Top-level PnR: use of Abutted Blocks



(a) Clock latency across fabric using CTS algorithms. **(b)** Clock latency across fabric using dedicated clock network in Fig. 70.

Figure 70: Clock Latency/Skew Comparison between CTS Algorithms and Dedicated Clock Network

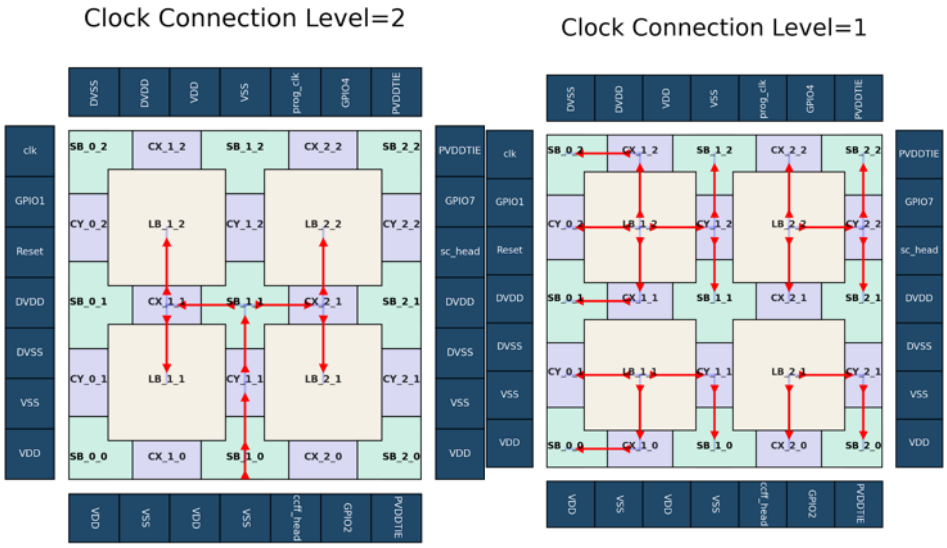
3.4.2 Customized Clock Tree Network

We have proposed a regular clock network to achieve linear CTS runtime and clean timing for any FPGA fabric. Upon analysis, it was observed that most of the time is taken by the top-level PnR which consists of the global signal routing (high fan-out nets, such as reset) and CTS. Rather than using the CTS, we have developed a dedicated clock network before top-level routing. As illustrated in Fig. 70, the clock network is organized in a hierarchical H-tree style. Our PnR scripts have been upgraded to allocate these dedicated routing channels and clock buffers. This reduced the top-level routing runtime significantly, making it a phase to just glue different modules together. Fig. 69 compares the clock latency distribution of these network using CTS and dedicated clock network. The dedicated clock network has a more balanced delay distribution and leads to shorter PnR run-time.

3.4.3 Optimization on Pre-routed Clock Trees

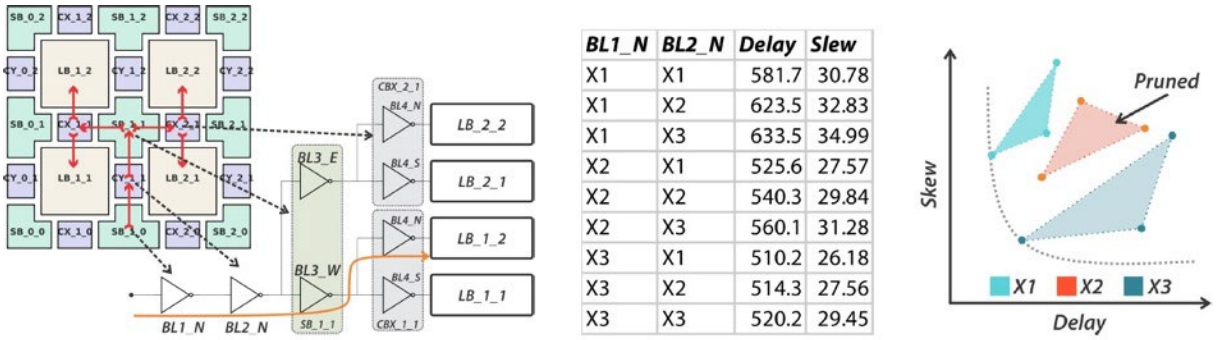
To optimize clock network latency considering Post-PnR parasitics, we implemented an optimization approach to size the buffers in the tree. We used the Van-Ginneken buffer insertion algorithm with a solution pruning policy [50]. To illustrate the buffer sizing strategy, Fig. 71 shows an example where a pre-routed H-Tree-based clock network is built for a 2×2 FPGA. To optimize latency for each path from the source to a sink, we consider a set of 10 buffers and 10 inverters as candidates for buffer placement. For example, in Fig. 71(a), we highlight such a path in orange, where BL1 N, BL2 N, BL3 W and BL4 N are the possible buffer placement location. We start from the clock source node in Fig. 71(a) and enumerate solutions for all the combinations of buffers for locations BL1 N and BL2 N. The STA engine is used to compute delay and slew at the output of the BL1 N, as shown in Fig. 71(b). We keep only the Pareto optimal solution, as shown in Fig. 71(c). The same process will be performed for the locations

BL2 N and BL3 W, until it reaches a sink node. The best latency solution is selected from the Pareto points, based on the latency and slew constraints that are defined by users.



(a) Connection from the chip I/O to modules. **(b)** Connection between adjacent modules.

Figure 71: Dedicated Clock Network Organization



BL1_N	BL2_N	Delay	Slew
X1	X1	581.7	30.78
X1	X2	623.5	32.83
X1	X3	633.5	34.99
X2	X1	525.6	27.57
X2	X2	540.3	29.84
X2	X3	560.1	31.28
X3	X1	510.2	26.18
X3	X2	514.3	27.56
X3	X3	520.2	29.45

(a) Pre-routed clock tree structure **(b)** Rendered solutions **(c)** Pareto optimal solutions for *BL1_N* location

Figure 72: Post-PnR Buffer Sizing Strategy

By evaluating the QoR of the pre-routed clock network and compared to the clock tree synthesis results from Synopsys ICC2. As shown in Table 16, we achieved 4 × improvement in skew with only 15% increase in the latency for the 8 ×8 design. We created a CTS algorithm to fit our model of hierarchical FPGAs synthesized through multi-instantiated blocks. The general principle is to minimize the wire length, therefore parasitic, while maintaining a symmetrical (note this does not mean H-tree) routing tree at each level of the clock tree network. Through the use of PrimeTime (interchangeable with any STA tool), we are able to perform a gradient approximation of the loss function, latency in this scenario, by applying statistical analysis to the loss function measurements themselves. The algorithm outline has been determined, and we investigated parasitic estimation and tunable gain parameters. We have adapted post OpenFPGA scripts to alter the FPGA structure to include the new feed-throughs needed for clock tree construction. Allowing ICC2 tool to perform the detail routing, our initial results showed that the

clock tree has a skew of 4 ps with a latency of 0.98 ns without performing buffer placement optimization via *Simultaneous Perturbation Stochastic Approximation* (SPSA). Results showed that *Simulated Annealing* (SAN) algorithm was consistently out performing SPSA algorithm which led to a redesigned clock tree synthesis algorithm, moving away from a SPSA-based algorithm. Finally, we investigated StarRC parasitic extraction to reduce errors in pseudo design for delta capacitance interpolation. The error was reduced from 25% to 5% through load averaging technique as shown in Fig. 72. We redesigned the flow to use closed-source tools, ICC2 and PrimeTime, to reduce the development effort.

Table 16: Clock Network Latency and Skew Comparison

FPGA	SYNOPSYS ICC2		PRE-ROUTED SOLUTION	
Design	LATENCY (PS)	SKEW (PS)	LATENCY (PS)	SKEW (PS)
2 ×2	122.52	30.6	134	10.11
8 ×8	252.56	50.4	304	12.45
16 ×16	N/A *	N/A *	2658	19.87
32 ×32	N/A *	N/A *	8954	50.5
64 ×64	N/A *	N/A *	21012	70.56

*Results are not applicable due to the P&R is never finished.

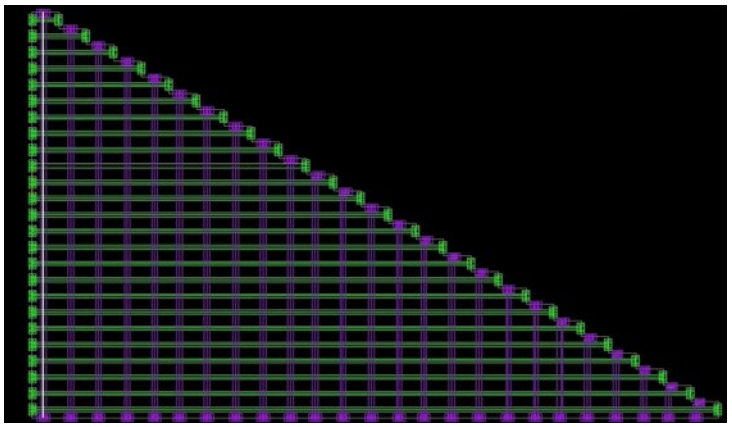


Figure 73: Template Design for Delta Capacitance Interpolation

Python code was extended for Pareto pruning to implement pruning and improvement in polling structure of code, with a reduced run-time from 45 minutes to 30 minutes. The size pruning characteristic was implemented and is unique for size and position, thus further reducing dimensionality of iteration. Our initial results show improvement from base clock tree of 1.0 us to 0.51 us or roughly a 50% improvement. Our tests show that wire capacitance estimation error is in the range of 15-20%. The new wire characterization design is shown in Fig. 72. Results show that starting from an un-optimized clock tree of 1 ns latency we can reduce it to 0.41 ns latency. Following principles of symmetrical trees adapted for FPGAs, we have developed a new symmetrical tree configuration that minimizes skew. Then, results indicate that our skew is 0.05 ns for a 2 ×2 FPGA.

The CTS revision 1.0 flow has been integrated in our PnR flow by adding additional make targets into the module PnR phase of our hierarchical CTS algorithm. Revised portions can be summarized through Fig. 73. The approach implements a semi-automated physical design flow

that takes advantage of the highly repetitive structure common to all homogeneous FPGAs. Highlighted subsections are modified or inserted from our previous work to implement our semi-automated clock tree synthesis algorithm.

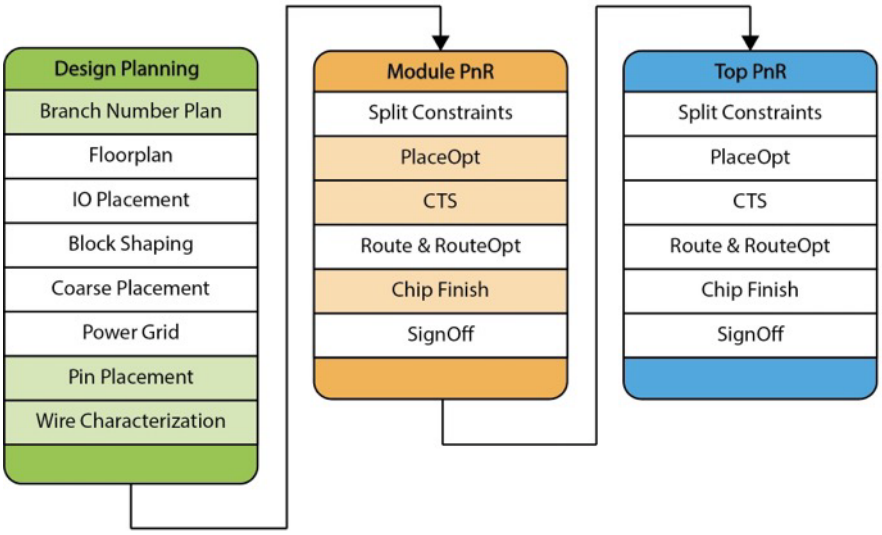


Figure 74: Hierarchical Place and Route Flow

Table 17 presents results compiled for 2×2 FPGA. Three comparisons are used in experimental methodology for skew and latency comparisons: First, we consider our base hierarchical place and route algorithm with no automation for CTS. Second, we consider a buffer removal scheme with routing corridors implemented to reduce skew. Third, we consider our full CTS algorithm with routing corridors and Pareto pruning for latency reduction.

Table 17: 2×2 FPGA CTS Results on Skywater 130nm

	Basic Flow	Structural Optimization and Buffer Removal	Pareto Optimal Solution
Latency (ns)	1.2	0.87	0.41
Skew (ns)	0.4	0.006	0.006
Runtime Overhead	0 minutes	10 minutes	150 minutes

Clock Tree Synthesis (CTS) Automation

CTS revision 2 flow has been integrated in our place and route flow. Revision 2 fully parallelizes the algorithm by utilizing a manager and worker-based schema. The manager first generates a list of tasks and each worker, running an independent PrimeTime subprocess, runs the task assigned to it through a task queue. Furthermore, a new pruning schema has been generated. In the new process we group nodes based on common ancestor path, and generate a independent pruning iteration for each ancestor tree. Results for the pruning method can be found in Table 18:

Table 18: Final Task Enumeration

	No Pruning	**Independent Pruning**
Number of Final Tasks	31067	767
Runtime (min)	>60	4

Table 19 presents results compiled for 2×2 FPGA. Three comparisons are used to demonstrate the benefit of multi-processing in the algorithm. First we consider results with a single thread and subprocess, and we sequentially iterate through all the tasks leading to a runtime of roughly 200 minutes. Second, when using the multi-threaded and multi-process implementation we were able to reduce the runtime to 15 minutes. Third, to demonstrate the usefulness of our CTS algorithm while utilizing our design flow, we show results gained through using ICC2 CTS with no feed-through generated.

Table 19: Table 19: Clock Tree Synthesis Comparison Results

	Pareto CTS Rev.1	Pareto CTS Rev.2	ICC2 CTS
Latency (ns)	0.41	0.41	0.56
Skew (ns)	0.05	0.05	0.06
Runtime (min)	198	15	1

CTS algorithm has been extended to larger designs consisting of 2^n *Configurable Logic Blocks* (CLB). In order to accomplish this, our design planning scripts had to be updated to restructure a 2 ×2 design to include the feed-through pins and top level connections. Then, we modeled the entire clock tree and its connections upon the lower-most *Switch Block* (SB) module. Placing the pins there allows all other module pin placements to be derived from this reference placement. Given clock tree logical connectivity for the larger designs, we needed to update our physical design scripts to accommodate more general pin and corridor configurations. Therefore, our routing corridor insertion script has been updated to realize routing corridors for different amounts of pins, ranging from 2 to 6 pins per corridor. For our 32 ×32 FPGA, which contains ten clock levels, the switch block, Fig. 74, contains the most corridor configurations. Results have shown that using the + corridor for pin configurations 4 or larger does not effect skew results which allows some simplification in the process and scalability of our designs. Currently we have scaled our design successfully to an 8 ×8 FPGA and finished the 32 ×32 FPGA design. For the 8 ×8 design, our algorithm has been compared to narrow channel CTS through IC Compiler 2's (ICC2) algorithm. Through our structural benefits and our efficient buffer sizing and positing algorithm we have generated a clock tree that has a latency of 1.42 ns, whereas ICC2 was only able to implement a clock tree of 2.52 ns, showing the benefit of our automated CTS algorithm.

3.4.4 Improved Back-end Run-time

Based on the back-end strategies, i.e., frame view-based (see Section 3.5.2) and customized clock network (see Section 3.5.3), we have managed to scale up our PnR to 40k-LUT FPGA fabrics whose runtime is 18 hours. We have studied the runtime with regards to FPGA sizes and compared to our previous best practice, as shown in Table 20. Meanwhile, we have managed to improve the buffer sizing in the clock network. The clock latency in a 2×2 FPGA is now 0.9ns, which was 1.8ns, being close to the synthesized CTS results, which is 0.8ns.

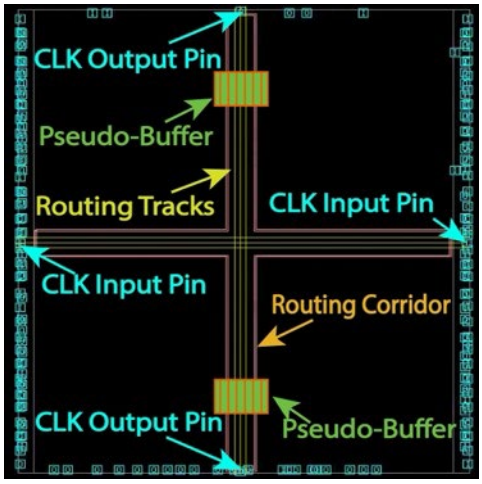


Figure 75: Corridor Configurations

Table 20: Top-level Backend Runtime Reduction Before and After using Customized Clock Network

FPGA Size	No. of LUTs	Previous Runtime	textcolor
2 ×2	40	30 Minutes	4 Minutes
8 ×8	640	480 Minutes	10 Minutes
16 ×16	2,560	Never Finished	20 Minutes
32 ×32	10,240	Never Finished	30 Minutes
64 ×64	40,960	Never Finished	18 Hours*

*Mainly due to some unoptimized script to handle big file size

3.1.1 Open-source RTL-to-GDS Workflow

We automated the processes of placing macro-level I/O signals and top-level macro cells. We also worked through some minor issues in other steps, such as PDN generation to generate a DRC-clean and LVS-clean 2×2 design, as shown in Fig. 75. We also started extending the open-source workflow to a 16×16 design, to demonstrate its scalability.

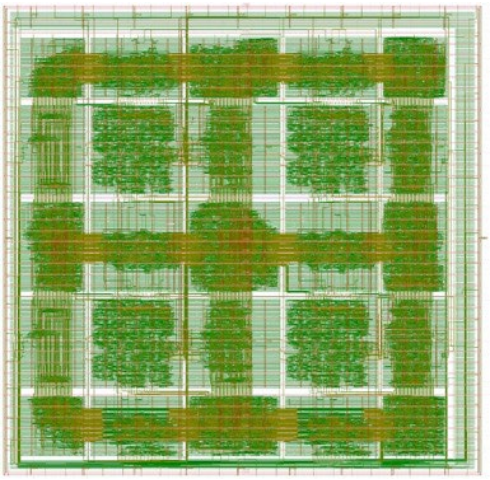
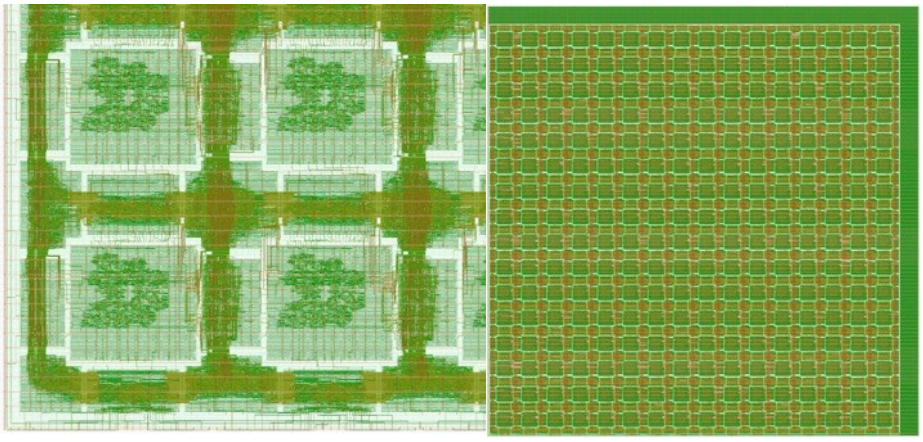


Figure 76: 2x2 DRC-clean FPGA Fabric Generated by OpenROAD/OpenLane

We created a private fork of the OpenLane repository [51] to share within the organization, and updated the 2×2 project to work with a newer version of the toolchain. We also created a project for a 16×16 design, and implemented rectilinear macro shapes in both the 2×2 and 16×16 designs (Fig. 76(a), and Fig. 76(b)). All of the projects are DRC/LVS-clean according to MAGIC [52]. We evaluated the layout, to see if it can fit the MPW shuttle offered by eFabless.



(a) Bottom-left corner of the 16×16 layout. **(b)** Full GDSII layout view.

Figure 77: Layout View of the 16 × 16 FPGA Fabric Generated by OpenROAD Flow

3.4.5 Support of Heterogeneous Blocks in the Physical Design Flow

The physical design RTL-to-GDS automated flow for 4×4 heterogeneous FPGAs (Fig. 77) generated via OpenFPGA has been completed. Techniques including moving post-processed homogeneous feed- through insertion to a corresponding heterogeneous module to maintain connectivity, reshaping modules to maintain narrow channel floorplan implementation, and redundant port pruning have been incorporated. The pin placement flow can be compatible for scalable architectures, including homogeneous and heterogeneous modules. The strategy can be summarized by first placing common pins on the periphery of multi instantiated blocks during the base 2×2 (homogeneous) or 4×4 (heterogeneous) floorplans which are exported to larger fabrics. When scaling to larger fabrics sizes, the common ports such as clock tree ports, are inserted before module level PnR such that it may be exported to scaled FPGAs. An automated synthesis flow has been incorporated to the physical design flow to help facilitate total design package. Users will be required to provide SDC constraints along with other key features which are to be incorporated on custom design. These features include sleep transistors, clock gating, and other enhancement features.

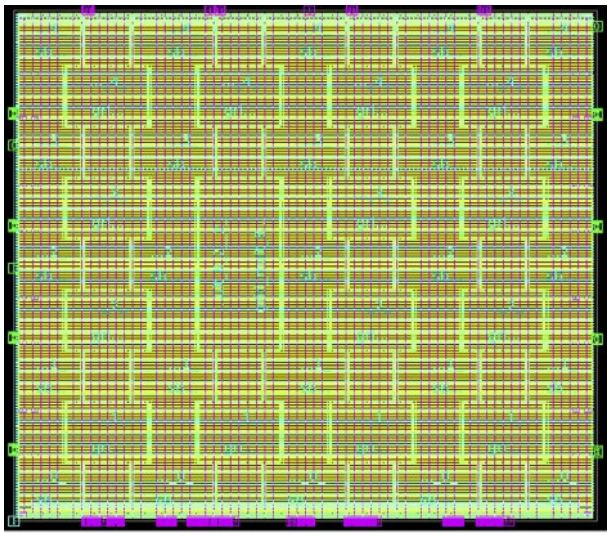


Figure 78: Skywater 4×4 Heterogeneous FPGA using 8×8 Multipliers Generated by Synopsys IC Compiler II

3.4.6 Hierarchical Physical Placement

We started focusing on different physical design techniques to optimize the performance of the FPGA design and provide different tiling structures for floor-planning the FPGA. In that context, we started exploring how a physical hierarchy can be manipulated to achieve the intended floor-plan. After preliminary experimentation with the SpyDrNet [53] project from BYU, we started extending the functionality to accommodate physical restructuring. We implemented and tested syntax such as (1) merge instances (2) merge multi-instance (3) merge wires (4) merge pins (5) optimize pins (6) optimize connectivity (7) create feed-throughs. We also integrated global signals and clock signal embedding in the spydrnet-physical plugin.

As shown in Fig. 78, the Tile02 strategy has been implemented, which combines the routing resources from each side of the logic block. Splitting of routing modules has been achieved using the Metis partitioning library [54]. The routing modules are represented as a connectivity graph. Each mux is described as a node, and each net is defined as an edge. Constant weights are assigned to each intermediate net, and higher weights are assigned to nets connecting the input and output ports. This representation allows splitting routing modules equally in two-part to be merged in two different tiles. We have generalized the Tile02 strategy for different customizable fabrics supported with OpenFPGA. We built a testbench and a verification methodology to evaluate the correctness of the reorganization. The verification methodology transforms the scripts generated by the OpenFPGA to accommodate restructuring-related changes.

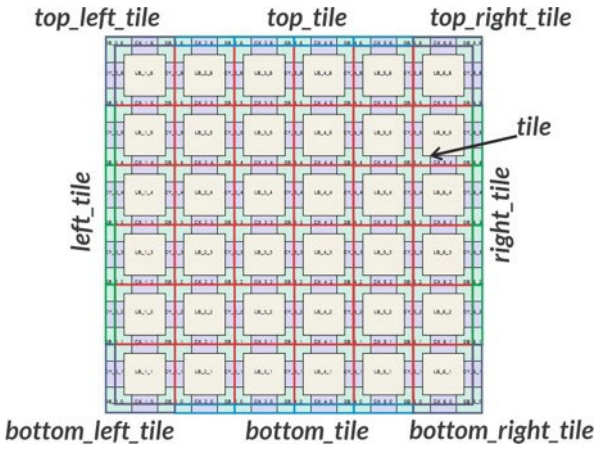


Figure 79: Plan of the Tile02 Strategy

In parallel, we developed a clock tree and global signal design framework using SpyDrNet-Physical; it allows FPGA designers to plan the top-level signal connectivity in homogeneous and heterogeneous structures [3]. The signal embedding is validated using formal verification (using Synopsys formality). Figure 79 shows the clock tree design for 4 ×4 FPGA fabric, due to a more granular repetitive structure, it is considered a 9 ×9 2D grid. The clock tree is a hierarchically/layered connectivity to satisfy buffering needs. The L0 pattern in the figure shows the lowest level and L1 a layer above the lowest level. The combined view is represented in the combined pattern. The dotted arrows represent a connection going to the bottom layer or coming from the top layer.

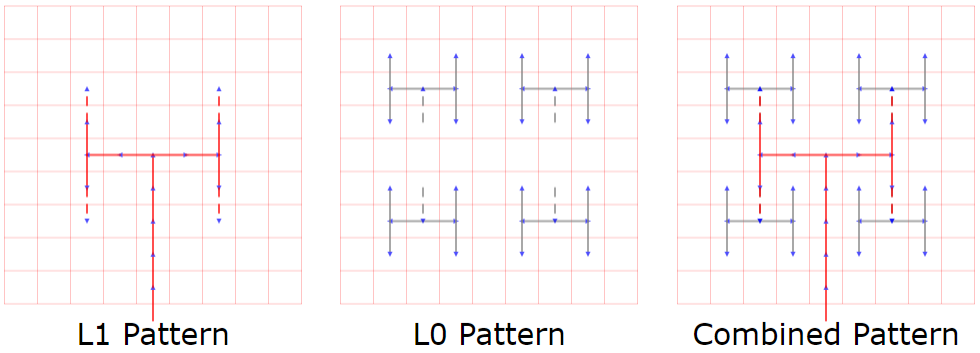


Figure 80: 4x4 FPGA Clock Tree Designs

We also focused on the restructured netlist verification strategy. After restructuring the physical hierarchy of the FPGA netlist, the logical paths of the configuration chains are no longer valid. A typical module restructuring operation is shown in Fig. 80. Fig. 81 shows the FPGA with neighboring connection boxes merged to modify the physical hierarchy. Invalid logical paths break the pre-configured testbench for the verification. A pre-configured testbench is essential to speed up the functional verification process. We developed a script that hierarchically extracts the configuration chain to reinstate the valid configuration path. A module-level chain traversal removes the module-level paths, and a top- level configuration chain traversal pulls the sequence of the module connectivity. These features have been embedded into the SpyDrNet-Physical library. The early validation experiments helped to detect many design bugs that occurred as a part of the restructuring process.

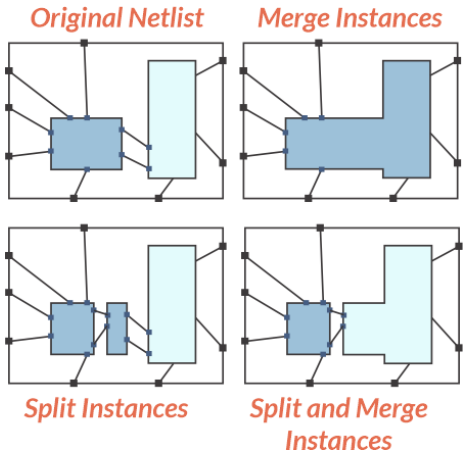


Figure 81: Physical Restructuring Operation

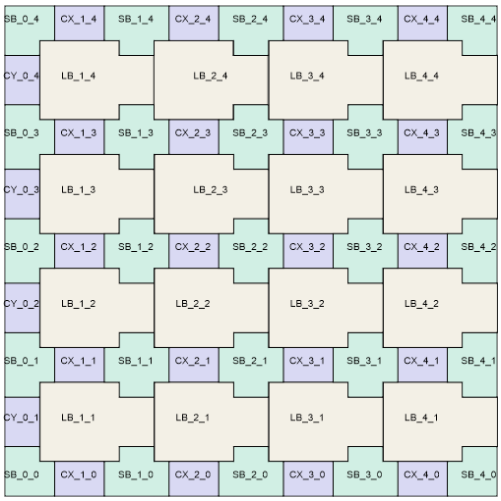


Figure 82: Merging the Connection Box and Complex Logic

Continuing our work on the performance extraction of different FPGA tiling strategies, using K6 N10 architecture with a 50% populated crossbar. Different tiling strategies present different opportunities for tile area optimization, as presented in Fig. 82. According to our performance evaluation, compared to the classic tiling strategy, we obtained an 11% and 10% deduction in the area using area- optimized tiling and generic tiling schemes, respectively. The timing segment variations also significantly differ in each implementation, and we observed a 15% and 20% timing variations reduction using area-optimized tiles and generic tiles, respectively. As expected, the runtime of hierarchical physical design correlates to the number of instances on the top-level runtime. In this comparison, the classic tiling requires 161 minutes to finish, area-optimized tiling requires 111 minutes, and generic tiling requires 140 minutes to finish, as shown in Fig. 83.

(a) Classic tiling scheme tile GDS view (b) Area-Optimized tiling scheme tile GDS view (c) Generic tiling scheme tile GDS view

Figure 83: Different Tiling Strategies for 4x4 FPGA Fabric

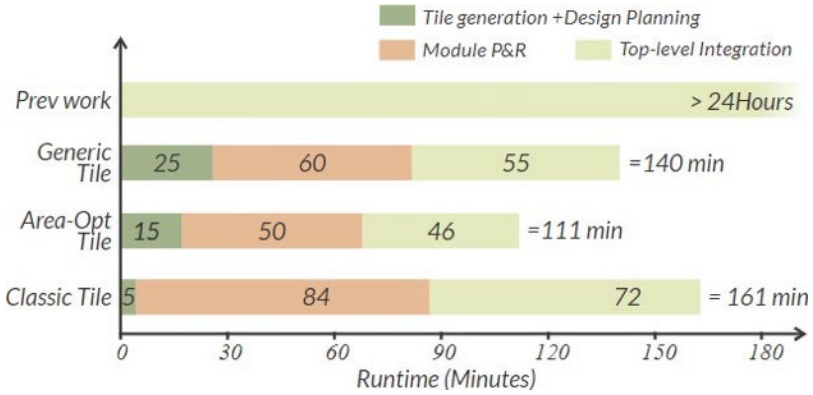


Figure 84: Runtime for Different Timing Strategies

Post-silicon Configuration Chain Testing

We worked with the efabless team to perform post-silicon testing (for the 8 ×8 FPGA chip submitted to the chip-ignite program). We resolved all the discrepancies in the Verilog netlist and successfully performed the power-up, *Configuration Chain Flip-Flop* (CCFF), and *Scan Chain Flip-Flop* (SCFF) tests in verification. We performed these tests on the silicon due to timing issues in the chip-ignite submission; post-silicon testing requires a few hacks. The idea is to develop an automated flow to mimic the custom design techniques using automated tool-chains. We targeted shift register-based protocol and latch-based memory bank protocol. In shift register-based protocol, taking advantage of the configuration chain reordering, multi-bit flip flops, and clock routing with a negative skew. All the above techniques help improve the area efficiency without impacting the datapath performance. In the case of latch-based memory bank protocol, we partitioned the problem into two parts, first looking at the word and bit line placement with respect to the initial latch cell placement and second exploring the optimal addressing scheme.

3.1.2 Fabric Key

OpenFPGA now accepts a fabric key, which 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 FPGA chips even with an open-source tool. Fig. 84

illustrates a practical scenario of using a fabric key against attackers. Designers created a fabric key when taping out the FPGA chip and share it with end users. With the key, end users can generate correct bitstream and download it to the chip. On the other side, an attacker, e.g., who got access to the chip through untrusted foundry or other channels, have no access to the key and can only generate wrong bitstreams. To make things worse, attackers will not know the correctness of the bitstream unless he downloads it to the FPGA chip. As the key complexity increases with the FPGA sizes, the difficulty to crack the key using a brute-force method increases.

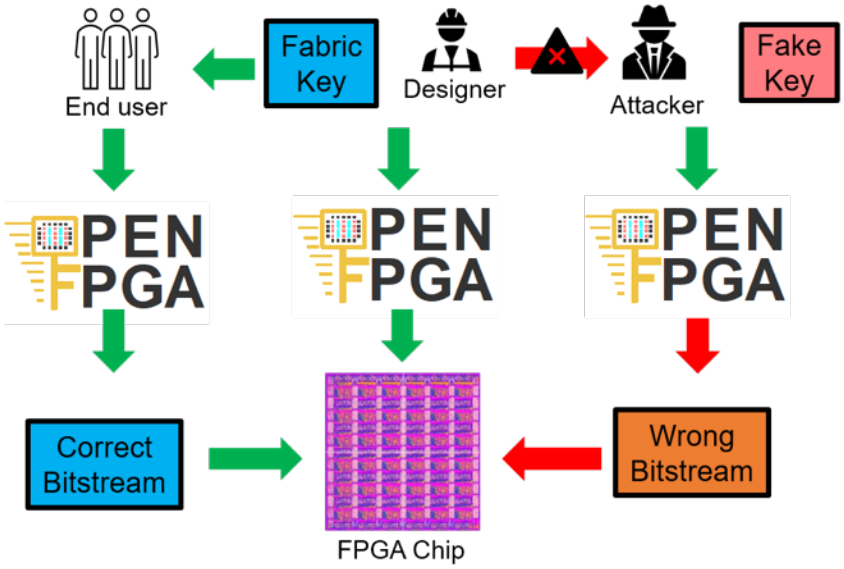


Figure 85: Fabric Key Provides Bitstream Obfuscation for Attackers

Fabric key is in XML format to be consistent with the VPR file formats. It can be switched to binary format or encrypted file formats upon DoD's need. Detailed usages are documented at [30]. Test cases on fabric key generation, loading and verification have been covered in our regression tests, as presented in Section 3.8.4).

3.4.7 Secured Bitstream Configuration

To configure the FPGA with a secured bitstream, we have proposed a generic architecture system, as shown in Fig. 85. The secured bitstreams may be downloaded to the FPGA through a JTAG port and then decrypted on-the-fly by the decryption module. JTAG instructions, defined in the IEEE-1149.1 standard, have been selected to integrate the configuration protocol into the encrypted bitstreams. The configuration protocols are necessary to configure the *Test Access Point* (TAP) of a JTAG interface. A JTAG TAP consists of a *Finite State Machine* (FSM), one instruction register, and multiple data registers, and is responsible for interpreting JTAG signals. This can allow encrypted bitstreams to be transferred to the decryption module contained in the FPGA periphery.

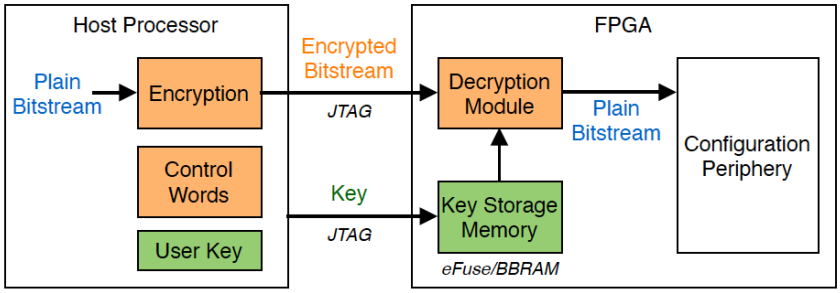


Figure 86: Block Diagram and Working Principle of Secured Bitstream Configuration

3.4.8 Programming Management Unit (PMU)

The hardware description for a JTAG interface that meets the basic requirements of IEEE-1149.1 standard has been implemented in a stand-alone repository, along with full test coverage for the JTAG interface [55]. A more refined and detailed architecture, called the *Programming Management Unit* (PMU), has been proposed to support advanced configuration for secured programming of FPGAs, as presented in Fig. 86. The PMU can consist of a custom FSM which decodes a bitstream header and sends data to the targeted destination. It also contains the AES decryption module, decryption key and related architecture to support data movement between the components used to program FPGA core, non-volatile memory, and AES key. The PMU is capable of storing a cryptographic key in a dedicated non-volatile memory and loading a bitstream to an FPGA core and decrypted the bitstream on-the-fly by the AES module, as the specification requested. This PMU provides a new encoding scheme embedding instructions and corresponding data register, in accordance with the JTAG interface standard, as presented in Fig. 87. The encoding scheme has been implemented and consists of information related to bitstream destination and bitstream data length, as well as a header and footer to configure the JTAG interface. New instructions and data register target a configuration block contained in the FPGA periphery with an encoded bitstream.

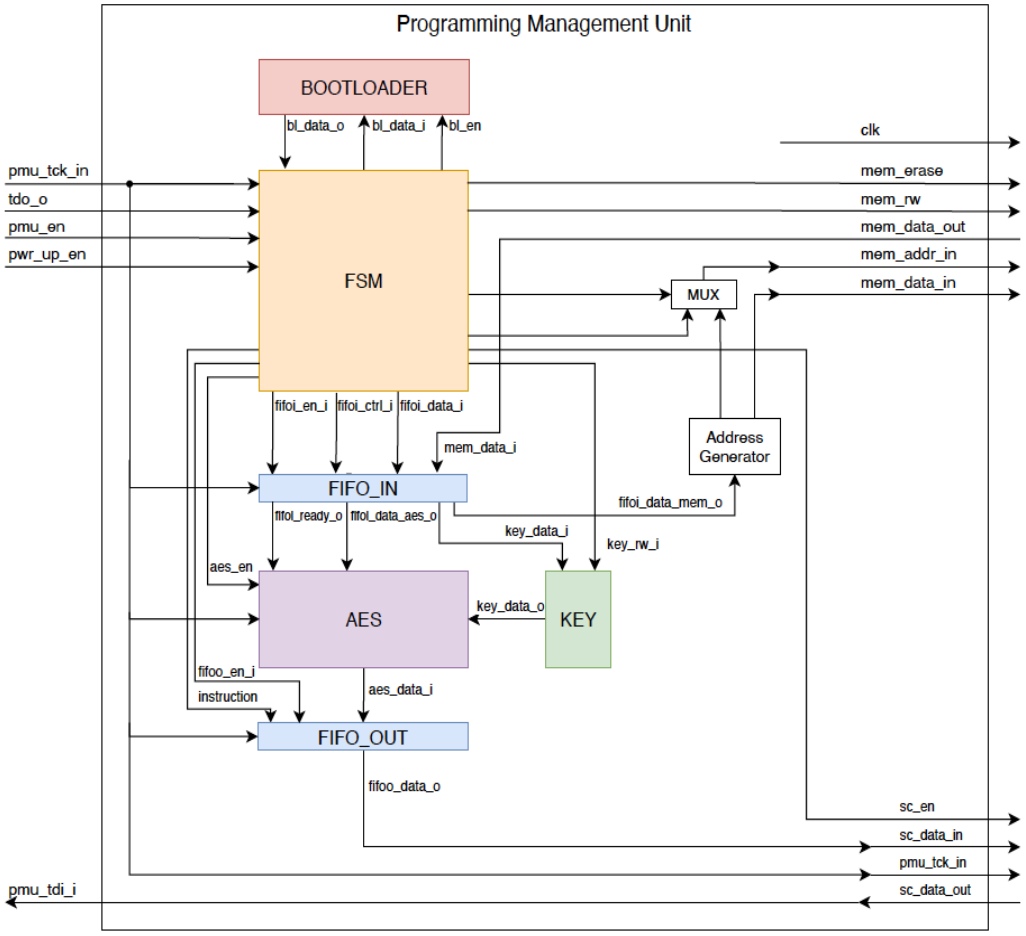


Figure 87: Programming Management Unit (PMU) Architecture Overview

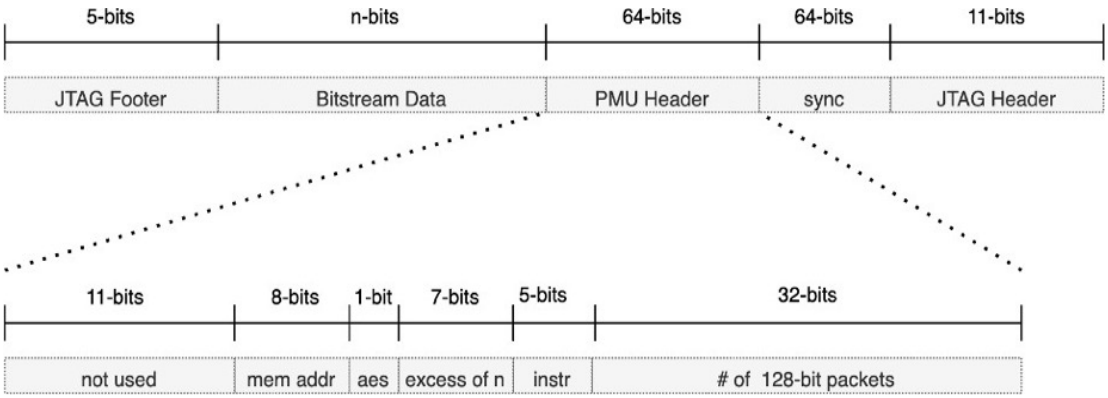


Figure 88: PMU Encoding Scheme for Instructions and Data

3.4.9 PMU Version 1

PMU version one is presented in Fig. 88. This began with version one which incorporates a JTAG TAP controller that is purely capable of loading an FPGA core with a bitstream, and a checksum module. The checksum module sums packets of incoming data and performs a checksum algorithm with the purpose ensuring data integrity when the FPGA bitstream is being

sent from the host PC to the FPGA core. This step-by-step approach facilitates the assessment of security achieved within each version, by addressing a threat and then implementing a countermeasure for that specific threat. This provides a structured approach to define what security threats the proposed architecture is resilient to. Version one of the PMU acts as a baseline with no security features and subsequent versions demonstrate addressed threats and countermeasures as improvements to the baseline. Additionally, progress with respect to debugging and experimentation with open-source tool Siliconcompiler [56]: LEF, DEF, and GDS files can be generated for Verilog designs and were demonstrated by generating a 2×2 SOFA FPGA fabric using the Siliconcompiler flow.

A revision to *Programming Management Unit* (PMU) Version one was made to improve the latency and size of the previously proposed checksum implementation. The new checksum module consists of a eight bit linear feedback shift register that relies on modulo arithmetic over a finite field of two elements: zero and one, to compute a checksum. The schematic for the CRC-8 is show in Fig. 89.

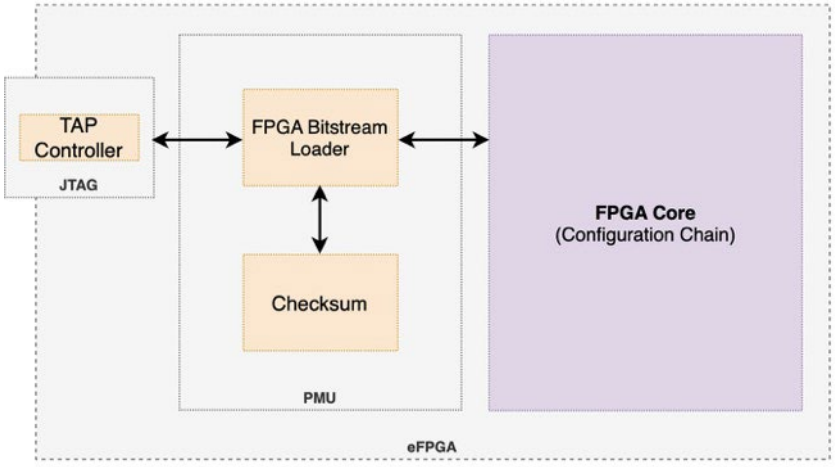


Figure 89: PMU Version 1 with Checksum

Figure 90: Bit Cyclic Redundancy Check

The purpose of incorporating a checksum in the PMU design is to ensure data integrity when the FPGA bitstream is being transferred from host PC to FPGA core. Revisions were also made to PMU threat model to be more accurate with respect to the security threats addressed by the PMU. The PMU threat model is a road map for design methodology that defines security threats and the countermeasures implemented in PMU to give an accurate representation of the security achieved by the PMU.

Implementation of the PMU version 2.0 RTL incorporates an AES module according to the PMU threat model defined in collaboration with NYU. Since the AES module incorporated in PMU is sourced from other existing projects, prior AES blocks failed to synthesize. Therefore,

another AES block design has been sourced that will have better out of box compatibility with the existing PMU design and RTL synthesis tools and was incorporated into the PMU. Additionally, to include the PMU in a test chip, further efforts have been made to collaborate with the SiliconCompiler team with the goal of submitting a 2x2 SOFA FPGA to eFabless MPW-6. A result of this collaboration is a flattened 2x2 FPGA design that has been correctly integrated in the template floor plan provided by eFabless caravel project wrapper. As well as the capability to execute a hierarchical design flow using SiliconCompiler tool. A flattened 2x2 SOFA FPGA placed in the caravel user project wrapper is shown in Fig. 90.

PMU version two and three as described by the PMU threat model defined in collaboration with NYU were completed. The latest version of the PMU consists of an FSM to interpret the PMU encoding scheme, a CRC8 module for data integrity checking, an SHA-256 core to authenticate the PMU user, and an AES-128 core to decrypt a bitstream on the fly as it is being loaded into an FPGA core. RTL and functional test coverage for all the intended operations of PMU are complete and significant efforts were dedicated to submitting a PMU test-chip to the eFabless MPW-6 tape-out. The PMU test chip is shown in Fig. 91 that contains one flattened PMU with AES and SHA cores, as well as a flattened 2×2 FPGA and several spy-pads in order to properly test the behavior of the FPGA and PMU post fabrication. This was possible due to the on-going collaboration with SiliconCompiler team [56]. However, despite our best efforts a PMU design which passed the MPW pre-checks was not submitted before the MPW-6 tape-out deadline.

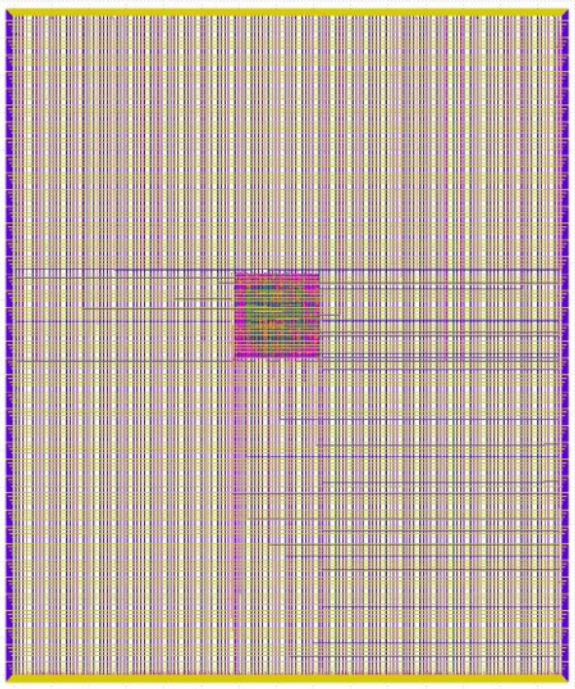


Figure 91: 2x2 SOFA FPGA Placed in Caravel Floorplan Template

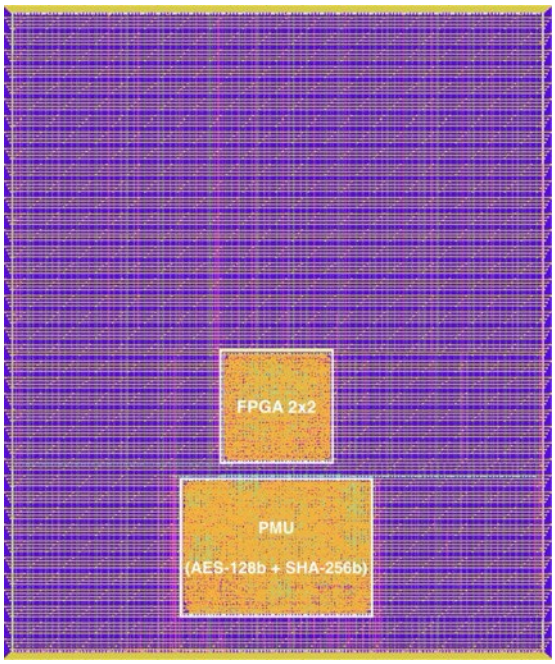


Figure 92: 2 × 2 SOFA FPGA and PMU Test-chip Sub Mission to eFabless MPW-6

3.5 Custom Multiplexer Cells

Since, standard-cell-based multiplexers lead to a significant power and timing overhead when compared to commercial FPGAs [57], we developed two custom cells which are 2-input and 3-input transmission-gate-based multiplexers with inverted inputs. These are compatible with Skywater 130nm HD cell library, in order to fabricate them during the Skywater tape-out (see Section 3.11.5). The custom cells have been used in the SOFA-CHD eFPGA IP [58], which has been sent for fabrication.

We evaluated the performance of the routing multiplexers built with the custom cells, when considering 4-input and 6-input routing multiplexers. Our baseline is the standard-cell-based multiplexers using the Skywater 130nm High-Density MUX2 cells. The custom routing multiplexers are built with one-level structure [59]. As shown in Table 21, the 4-to-1 custom multiplexer implementation reduced power consumption by 22% and t_{pd} by 31% with an equal area requirement. As shown in Table 22, the 6-to-1 custom multiplexer implementation reduced the power consumption by 10.5% and t_{pd} by 27% with a 20% area overhead. Post-layout simulation shows that using the custom cells, the performance of an eFPGA IP can be improved by close to 2×. See details in [60].

Table 21: Performance Comparison between 4-to-1 Multiplexer Implementations

	Custom Cell Multiplexer	Skywater Multiplexer
Power (μW)	2.37	3.03
Timing (ps)	211.10	304.30
Area (μm^2)	33.78	33.78

Table 22: Performance Comparison between 6-to-1 Multiplexer Implementations

	Custom Cell Multiplexer	Skywater Multiplexer
Power (μW)	2.96	3.31
Timing (ps)	272.60	374.20
Area (μm^2)	61.31	48.80

3.5.1 Custom SRAM Cell

We designed a custom SRAM cell (see Fig. 92) compatible with the Skywater 130nm HD cell library. We characterized it and used it as a basis for LUT-RAM and SRAM based FPGAs. This cell is also part of the Smart-Redundacy effort and will lead to ECC checking inside the memory.

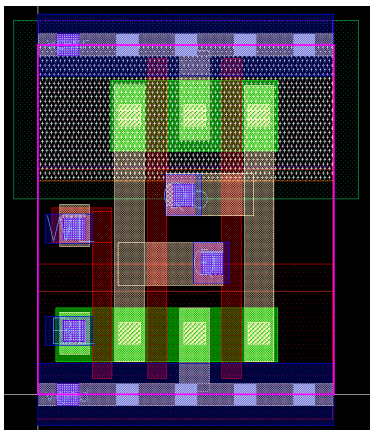


Figure 93: Layout View of a SRAM Cell for Memory Element Generation

3.5.2 Custom Cells for Radiation Hardening

An FPGA is mainly composed of buffers, multiplexers (MUX), *Look-Up Table* (LUT), *Flip-Flop* (FF), and SRAM. Data storage cells as SRAM and Flip-Flop are sensitive to *Single Event Upset* (SEU) that alter the correct behavior of the design. To address this problem, configuration memories are modified to integrate an error correction based on the Hamming code, and connected to a solver correcting errors. Flip-Flops are triplicated and connected to a voter. LUT's architectures are modified for redundancy between two redundant logic elements, delayed LUT's output, covering buffer, and multiplexer transient events.

We designed a standard-cell DICE cell using the 130nm Skywater PDK (Fig. 93) to store FPGA's configuration memories. We proposed an hardened cell we floor-planned the memory to allow ECC checking as sensors using the organization in Fig. 94. We have showed that the SEU detection time at least 1,200 × faster than bitstream scrubbing for a 52% CLB area overhead (Fig. 95). This area overhead could be well compensated because the scrubbing logic is removed and the sensing method can be reused for Smart-Redundancy [61], as described in the next sections.

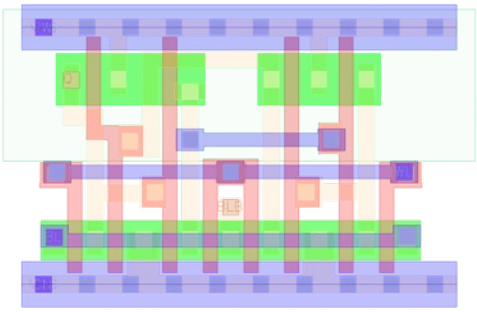


Figure 94: Standard-cell like DICE Cell Layout using Google-Skywater 130nm PDK used as Configuration Memory for Radiation Resistant Application

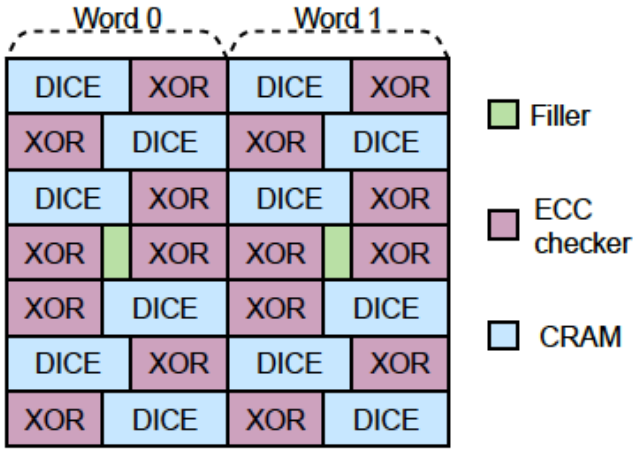


Figure 95: Repeated Pattern used to Allow In-Memory ECC Checking as SEU Sensor and to Avoid Three Bits Errors

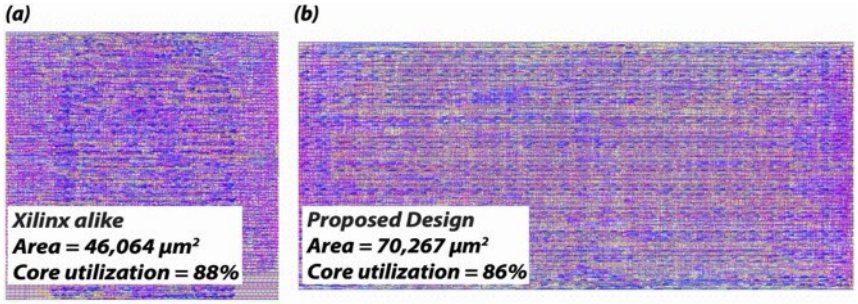


Figure 96: CLBs Layout for (a) Reference and (b) Work showing the Impact of In-Memory Checking on the Area and Block Shaping for Similar Core Utilization

3.5.3 Smart-Redundancy for Radiation-hardened FPGAs

We have proposed the Smart-Redundancy techniques and its architectures that handle *Single-Event Transients* (SETs) and *Single-Event Upsets* (SEUs) differently. As illustrated in Fig. 96, Smart- Redundancy still uses two *Logic Elements* (LEs) known as *Master LE* (MLE) (in green) and *Redundant LE* (RLE) (in yellow) with common inputs and tri-state outputs.

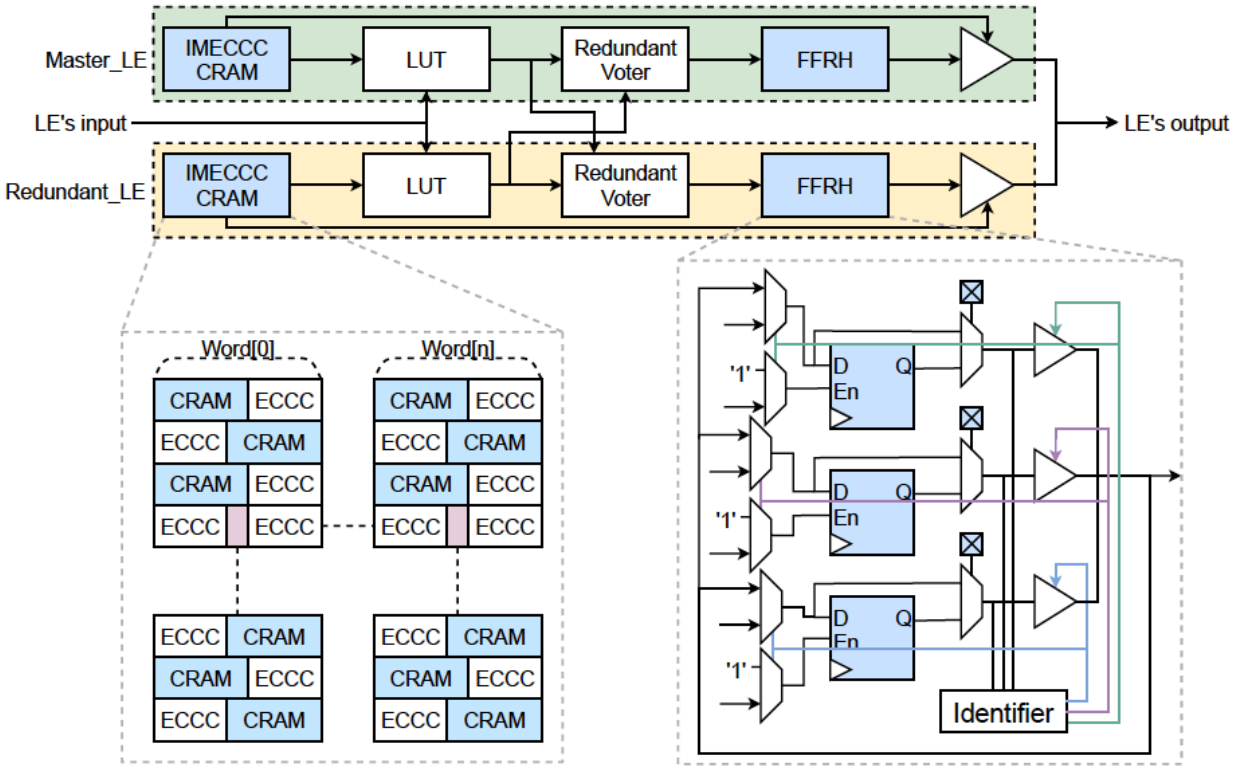


Figure 97: Block Diagram of the Improved Smart-Redundancy Architecture to Integrate IMECCC Sensors

CRAM is subdivided in words with integrated ECC checkers (ECCC) used as all-or-none sensors. FF are hardened with local TMR and auto-correction.

In Memory Error Correction Code Checking **(IMECCC):** The integration of IMECCC sensors covers only the *Configuration Random Access Memory* (CRAM)'s SEUs. It requires a word-like organization to insert the parity bits from the ECC code. The bottom-left of this figure shows the floor-plan pattern, block of three CRAM cells in staggered rows, used to integrate the sensors in the CRAM. IMECCC sensors detect SEUs and SETs in the CRAM.

Radiation-Hardened Flip-Flop **(FFRH):** The FFRH module triplicates the FF and enables auto- correction via the identifier module. FFRH's identifier detects SEUs and SETs in the FFs, as well as SET from the redundant voter module. The redundant voter module handles SETs in the LUT with local redundancy between MLE and RLE. We assessed the most efficient type of local redundancy, between temporal triple modular redundancy and dual modular redundancy, to integrate and study the implication in terms of performance.

The studied architectures enable very fast configuration repair. We want to compensate for the area increase caused by the sensors and local redundancy with less LE utilization than state-of-the-art mitigation methods. These architectures also significantly simplify the implementation flow over the state-of- the-art. Indeed, the architectures natively integrate redundancy while the state-of-the-art requires external tools to apply redundancy on the design to implement in the FPGA.

3.5.4 Local Clock Filtering for Radiation-hardened FPGAs

We then continued by focusing on the *Single-Event Transients* (SETs) coming from the clock tree and affecting the lower branches. To do so, we developed a local clock filter to instantiate in every FPGA element. As illustrated in Fig. 97(a), this module uses programmable phase shift sub-modules to triplicate the clock signal. Then, those triplicated clocks fed voters that intend to control locally triplicated *Flip-Flops* (FFs), as introduced before. This local clock filter uses 180° phase shift through programmable delays. In other words, it delays the clock signal by half a period of the initial clock. Such delays cause a sufficient SET propagation time for the voters to filter it. The programmable delays are handled by buffers and selectors, as illustrated in Fig. 97(b). The hardware composing this module can also be a victim of SETs. However, the impact is not different from an event coming from the clock tree and is filtered by the system.

The router module is used as a delay selector. Such selection is ensured with tri-state buffers controlled by triplicated CRAM, as illustrated in Fig. 97(c). The triplicated CRAMs are close to the FFRH presented in the last report. CRAMs are protected from *Single Event Upsets* SEUs and SETs by the rewriting logic controlled by the identifier sub-module. The identifier is natively resistant enough not to defeat the triplicated CRAM it protects. The proposed module perfectly fits our Smart-Redundancy methodology work by avoiding single-point failure in the signals routing.

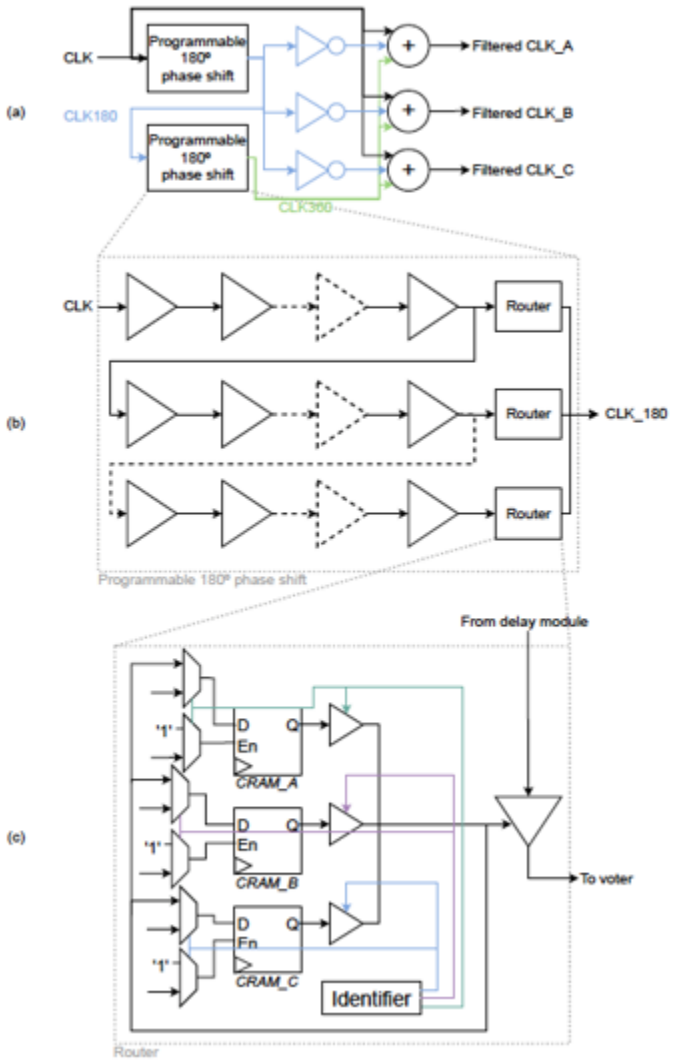


Figure 98: (a) is a Block Diagram that shows the Filtering Through Triplication from Locally delayed Clock Signal (b) Illustrates how the Delays Signals are Generated using Buffers and Selectors (c) is the Schematic of the Router Module where Triplicated CRAM with auto-correction are used to Control a Tri-state Buffer

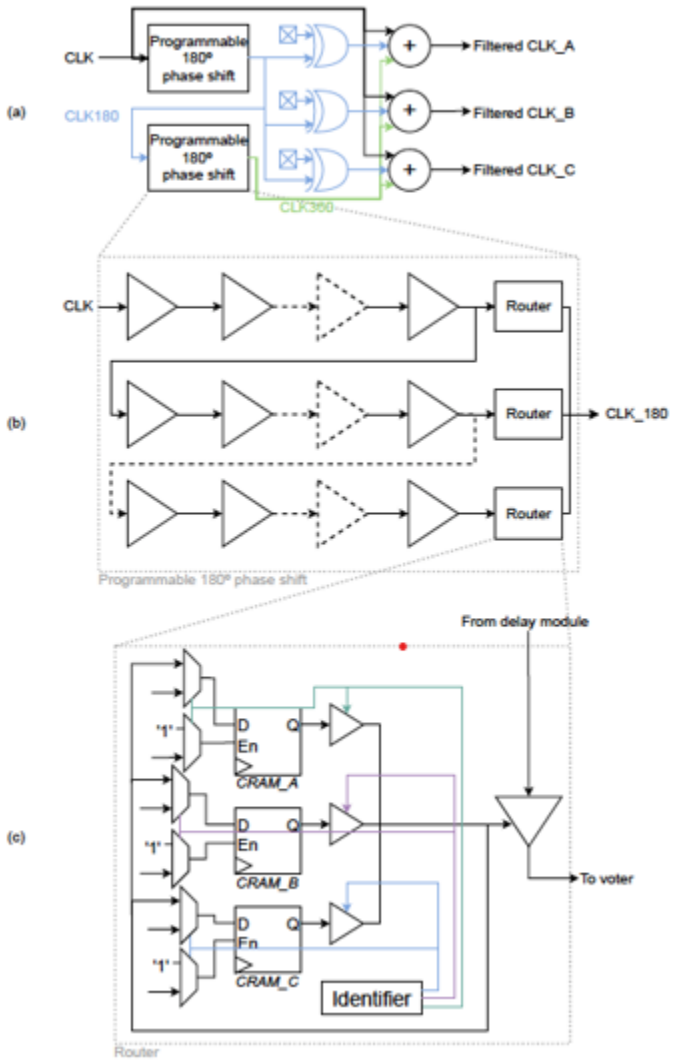


Figure 99: (a) is a Block Diagram that shows the Filtering Through Triplication from Locally Delayed Clock Signal(b) Illustrates how the Delayed signal are Generated using Buffers and Selectors (the Router module) (c) is the Schematic of the Router Module where Triplicated CRAM with Auto-correction are used to Control a Tri-state Buffer

3.5.5 Programmable Local Clock Filtering for Radiation-hardened FPGAs

Finally, we improved the local clock filter presented above. As illustrated in blue in Fig. 98(a), we replaced the initial inverters with CRAM-XOR pairs. This modification enables the re-utilization of fundamental frequencies for harmonics and near-harmonics frequencies. Therefore, the Programmable 180° phase shift modules, illustrated in Fig. 97(b), contain as many Router modules as required fundamental frequencies and reduce the total area by diminution of the number of self-protected CRAM, illustrated in Fig. 97(c). As proof of concept, we designed a *Programmable Local Clock Filter* (PLCF) with fundamental frequencies of 100 Mhz, 125 Mhz, 150 MHz, 175 MHz, and 200 MHz. We ran simulations for operating frequencies between 100 MHz and 800 MHz with 50 MHz steps with 30,000 SET injections. The PLCF demonstrated a

complete immunity to SETs for all the frequencies.

3.5.6 Custom Radiation-Hardened Architectures Characterization

Our work on radiation-hardened FPGAs led us to the design of three *Configurable Logic Block* (CLB) architectures. Those architectures, illustrated in Figures 99, 100 and 101, are based on our Smart-Redundancy [61] and IMECCC works. All three architectures intend to replace *Triple Modular Redundancy* (TMR) with locally monitored dual-redundancy. The first architecture handles *Single Event Transients* (SETs) in combinational logic with local *Temporal TMR* TTMR, while the two others are inspired by *Dual Modular Redundancy* (DMR) and called *Dual TMR* (DTMR) and *Double DMR* (DDMR).

We characterized the area overhead compared to a standard CLB as well as the CRAM self-repair time after tech-mapping, as illustrated in Tables 23 and 24. The results of this characterization showed self-repair times under 2 ns, which is an order of magnitude below read-back-scrubbing (usually several µs). They also showed an area overhead of only 38% in the best case.

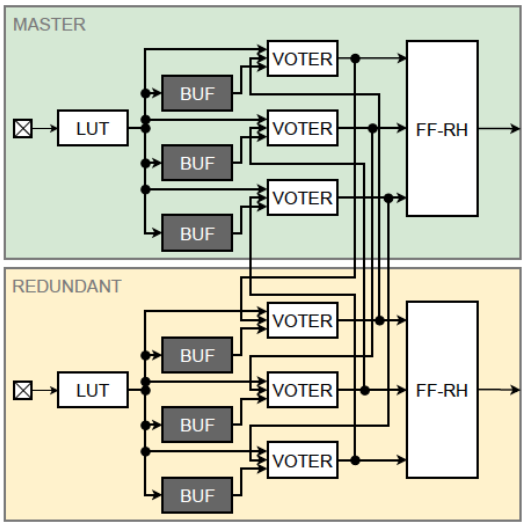


Figure 100: (topleft) TTMR Architecture Schematic with Delaying Buffer to Filter SETs

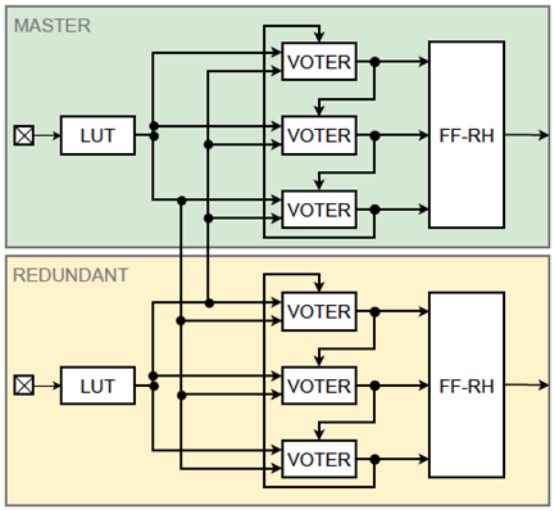


Figure 101: DTMR Architecture Schematic Replaces the Delay Module with Voter Looping Back

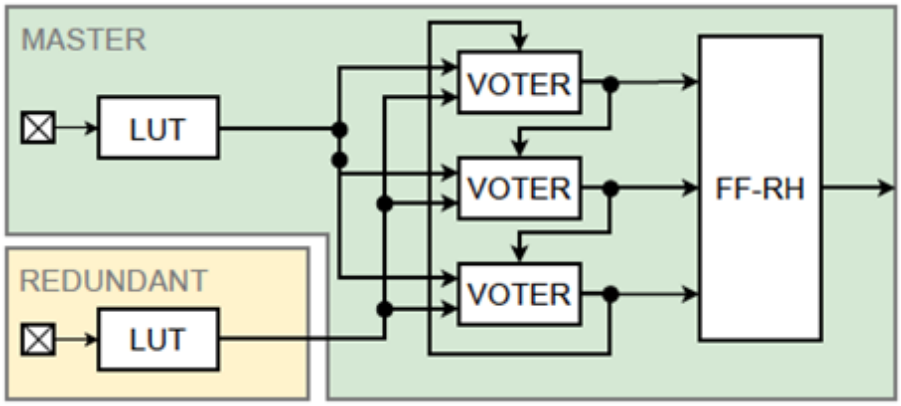


Figure 102: DDMR Architecture Schematic as an Optimization of DTMR

Table 23: Area Analysis of Custom Radiation-hardened k6N8 CLB Architecture in 40nm Technology

Architectures		Area (μm^2)	ratio
CRAM16	standard	10,666	1.00
	DDMR	16,111	1.51
	DTMR	16,904	1.58
	TTMR	17,138	1.61
CRAM32	standard	10,719	1.00
	DDMR	14,763	1.38
	DTMR	15,430	1.44
	TTMR	15,659	1.46
CRAM64	standard	10,653	1.00
	DDMR	16,887	1.59
	DTMR	17,656	1.66
	TTMR	17,851	1.68

Table 24: Time to Self-repair a Memory Word in a k6N8 CLB Architecture

	Delay (ps)
CRAM16	590
CRAM32	910
CRAM64	1,250

3.5.7 BRAM-like Configuration Protocol

We expanded the frame-based configuration protocol to support multi-bit writing as illustrated in Fig. 102, which approximately reduces programming time by number of bits ×. These packets, or words, are also fundamental for rapid *Error Correcting Code* (ECC) and Smart-Redundancy integration.

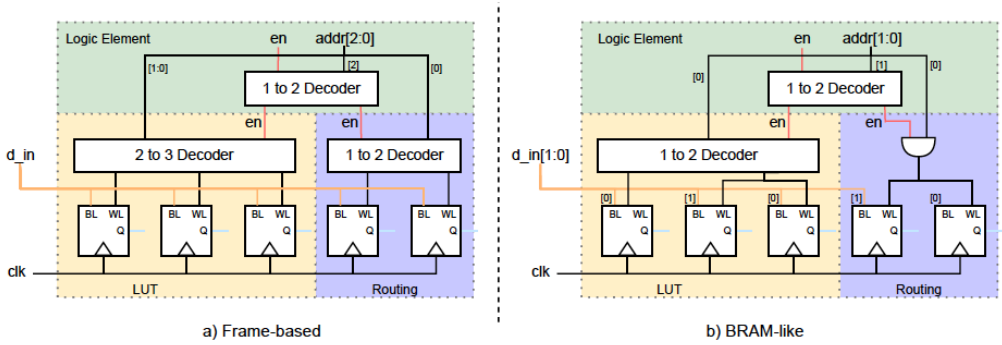


Figure 103: Comparison between Frame-based and BRAM-like Configuration Protocols through FPGA Hierarchy

Custom Radiation-Hardened Architectures Characterization

We characterized the *Smart-Redundancy with In-Memory Error Correction Code* (SRIMECCC) architectures presented in the previous report. Those architectures replace bitstream scrubbing and *Triple odular Redundancy* (TMR) with monitored dual redundant configuration memories and built-in filters to mitigate *Single-Event Effects* (SEEs). The post-physical design characterization has demonstrated the following performance:

- A reduction of 22% on average of effective area, despite a *Configurable Logic Block* (CLB) area increase by 40%, as shown in Fig. 103. Standard CLBs contain twice as many LE as SRIMECCC CLB because the last includes master and redundant elements.
- A power consumption reduced by 79% on average coming from 31% reduction on design implementation and 82% by scrubbing removal, as illustrated in Fig 104. Number of CLBs and LEs is reduced by the utilization of monitored dual redundancy compared to TMR. The amount of channel is also reduced because wires no longer need triplication. Then, the power reduction mainly comes from scrubbing removal but also from the required CLB amount reduction.
- A configuration memory bit-flip detection of 1.62 ns, which is thousands of times faster than scrubbing.

- An operating frequency reduced by 20% on average.

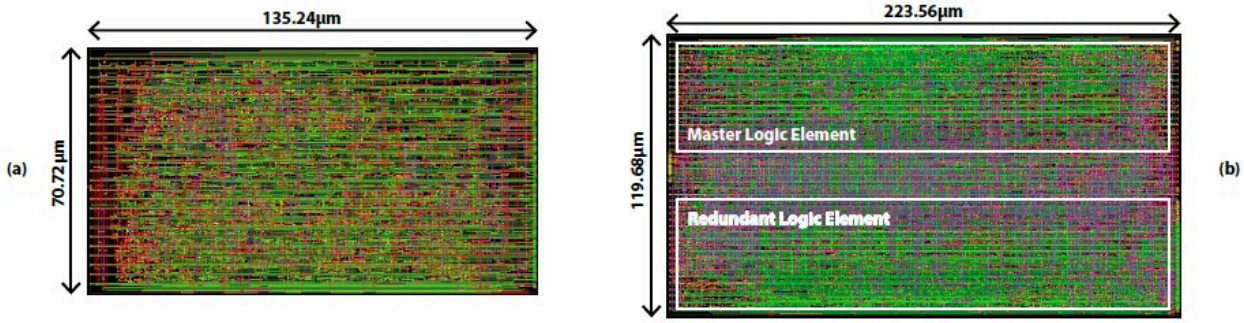


Figure 104: Layout View of (a) Standard FPGA Logic Element (LE) at 87% Utilization, and (b) an SRIMECCC LE at 88% Utilization

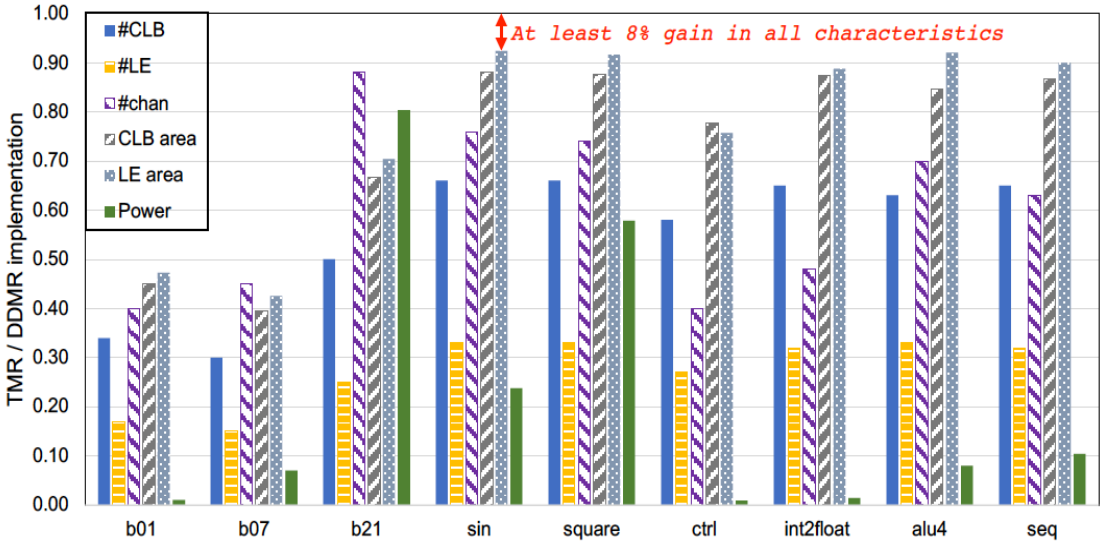


Figure 105: Global Performance Improvement from the SRIMECCC Architecture Compared to the State-of-the-Art Single-event Effects Mitigation Methods for a Subset of Benchmarks

3.6 Modernization on EDA Flow

To support a wider range of benchmarks, we have modernized the OpenFPGA flow by integrating Yosys [62] as front-end, as shown in Fig. 105. The flow is now capable of converting a Verilog benchmark to bitstream and outputting the Verilog testbench/netlists of a FPGA fabric. Note that our Verilog netlists for FPGA fabric are independent from the Verilog benchmark. The design flow has allowed us to test more modern Verilog benchmarks (compared to the traditional benchmarks used by VTR), such as the PicoRV32 [5] that can successfully be implemented on our generated fabrics.

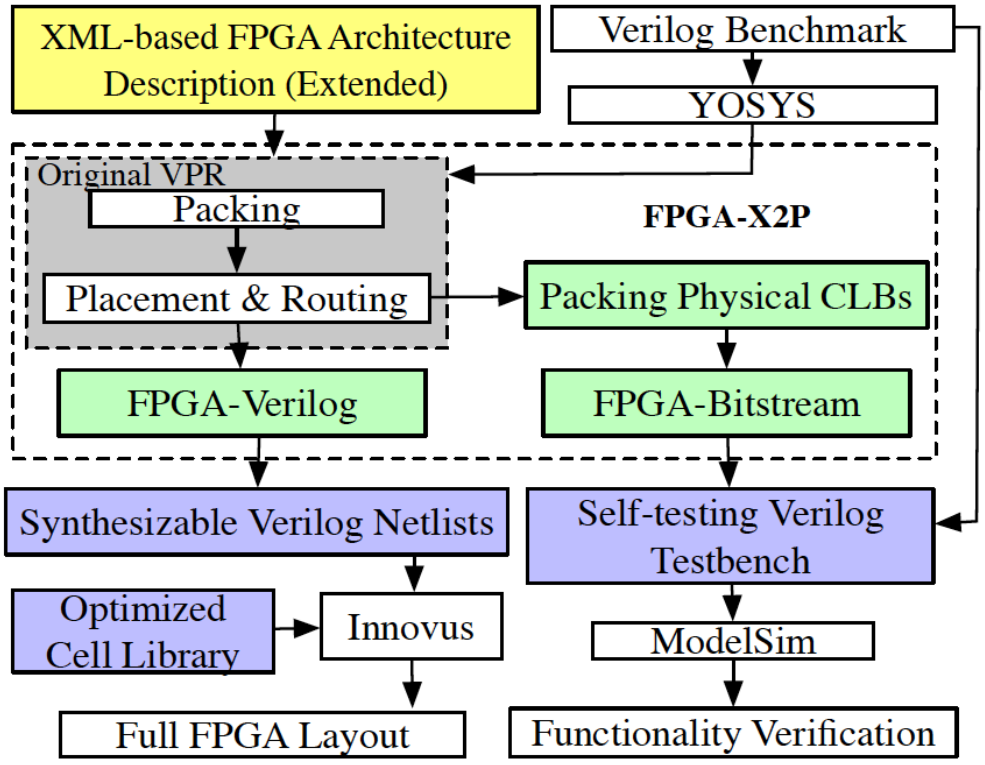


Figure 106: Refined EDA Flow based on Yosys, VPR and Verilog Generator

In addition, we have created a command-line user interface (see Fig. 106) to enable/disable our Verilog Generator functionalities. Table 25 provides a detailed explanation on each option.

```
FPGA Verilog Generator Options:
    --fpga verilog
    --fpga verilog dir <directory path of dumped Verilog files>
    --fpga verilog print top auto testbench <path to the Verilog benchmark>
    --fpga verilog print modelsim autodeck
    --fpga verilog modelsim ini path <path to modelsim ini file>
```

Figure 107: Command-line user Interface of FPGA Verilog Generator

Batch execution improvements of OpenFPGA shell: To enable fluent execution, we added a batch execution mode for the OpenFPGA shell. When enabled, OpenFPGA will abort immediately when fatal errors occurred, allowing better support for regression tests, as discussed in Section 3.9.4. Otherwise, OpenFPGA will enter an interactive mode when fatal errors occur. See details in [63].

Version display: this feature is useful for users when they report bugs. Previously, we had no version display when running the OpenFPGA executable. Version information can be called through:

- An option --version has been added to openfpga executable, with which users can check versions, for example: ./openfpga --version;
- A command version has been added to the OpenFPGA shell, with which users can check versions easily in the script. See details in [64].

Table 25: Description of Verilog Generator Options

Option Name	Description
fpga verilog	Turn on the Verilog generation for full FPGA fabric.
fpga verilog dir	Specify the directory that all the Verilog files will be outputted to. <directory path of dumped Verilog files>is the destination directory. Default val if not specified by user: <folder of user verilog benchmark>/syn verilog.
fpga verilog print top auto testbench	Turn on the self-testing Verilog testbench generation. The testbench will be placed in the folder *fpga verilog dir*. Users are required to specify the path to their Verilog benchmark, which will be included in the Verilog testbench.
fpga verilog print modelsim autodeck	Turn on auto-generation of Modelsim scripts to compile and launch HDL simulations. When this option is enable, two TCL scripts will be generated in the folder *fpga verilog dir*. Sourcing the TCL scripts in Modelsim will invoke a push-button HDL simulation for the self-testing Verilog testbenches.
fpga verilog modelsim ini path	Specify the path of the Modelsim configuration file, which is called in the auto-generated TCL scripts. This option is only valid when option *fpga verilog print modelsim autodeck* is enabled.

3.6.1 Continuous Integration and Development (CI/CD)

We moved our *Continuous Integration*/*Continuous Development* (CI/CD) platform to GitHub actions. It provides many useful features to enhance run-time. We created a conditional execution flow which detects the change in the different parts of the project and executes the respective integration or regression tests. Current run-time could be reduced to 5 minutes depending upon which part of the code repository is changed. This helps us reduce the wait time to merge pull requests.

Docker image auto-release: Moreover, this interface also generates docker images from the master branch with pre-compiled binaries. These released docker images can be directly used by the user, reducing the time required for tool adoption. The documentation of the respective features are also updated [65].

Pull request template: As OpenFPGA has more contributors than before, we formalized pull request [66] by introducing a template file for contributors. Any pull request must be submitted with a checklist and a brief design document, which help code reviews and merge them in time. A template pull request can be found here [67].

3.6.2 Extended Compiler Support

Compatibility to different compilers are crucial to OpenFPGA as users may have various environment. Previously, we recommended GCC-8 to compile the code base, which may conflicts with other software packages which require older GCC versions. We have extended our compiler support to GCC- 5, GCC-6, GCC-7, GCC-8, GCC-9, Clang-6 and Clang-8, covering most of frequently used compilers in Linux environment. The compatibility is continuously tested in our CI. Thus, OpenFPGA has upgraded its compilation system with CMAKE scripts. The upgrade brings the following advantages:

1. Seamless and easy integration with VPR8 [18], which uses the CMAKE system,
2. CMAKE can easily solve library dependencies, in particular considering that the VPR framework is under modularization,
3. CMAKE will ease the compilation of a *Graphic User Interface* (GUI) based on QT,

which uses CMAKE as a native compilation method,
4. CMAKE is more portable on cross-platform compilation, which should allow us to adapt OpenF- PGA for Win/Mac/Linux systems.

3.6.3 Regression Tests

We support regression tests to the current public GitHub repository of OpenFPGA [2]. The regression tests are based on GitHub CI/CD and supports two operating systems Linux and MacOS. Our test scripts support multitasking and a push-button Yosys + VPR + Verification flow, using the verification tool iVerilog Icarus [68]. Using these scripts, we have integrated a basic regression test, to cover 4 different architectures, as detailed in Table 26.

The benchmark is designed to test all the operating modes available in a Stratix-IV-like CLB includes adders, fracturable LUTs, shift registers, *etc*. For each benchmark and each architecture, we perform two tests: one is the fabric using minimum routable channel width and, the other is the fabric using a fixed routing channel width of 300. This aims to verify over a wide diversity of routing channel widths, which leads to different multiplexer sizes and structures. Note that the tests have covered the FPGA architectures and circuit-level designs to be used in our test chip in Section 3.10.

Table 26: Detailed Tests Integrated in GitHub CI/CD

Architecture	Testing goal
k6 N10 scff	Verify a fundamental FPGA with fracturable 6-input LUTs, that is configured by scan-chain flip-flops.
k6 N10 scff local encoder	Verify a fundamental FPGA, that is configured by scan-chain flip-flops and local encoders.
k6 N10 scff behavioral	Verify a fundamental FPGA where primitive blocks are generated in behavioral synthesizable Verilog.
k8 N10 scff	Verify an advanced FPGA with a fracturable 6-input LUT and a fracturable 4-input LUT, where multiplexers are made by standard cell MUX2.

To enrich the testing suites for OpenFPGA, the IWLS'2005 benchmark suite has been added to the project. All the benchmarks are deployed in the regression tests, where a complete Verilog-to-Bitstream design is applied to each benchmark, and then followed by a *Quality of Results* (QoR) check. Since the IWLS'2005 benchmarks require both DSP blocks and BRAMs, we use them when sizing the number of heterogeneous blocks for the Intel 22nm FFL tape-out. See details in [69].

3.6.4 Code Reconstruction

As described in Section 3.5.1, we have standardized OpenFPGA data structures, being independent from the VPR data structures, when refactoring OpenFPGA to support VPR8 release and 8.x versions. Therefore, a considerable amount of efforts have been made on refactoring our C code to the C++14 standard, which is used in the latest VPR versions. In particular, through the code reconstruction, we aim to provide easy-to-extend data structures for Verilog and bitstream generation, as well as documentation on how-to-use. All the new codes contain comments to help developers to understand internal data structures of OpenFPGA. We have accomplished all the action items in our code reconstruction plan for OpenFPGA, as shown in Table 27. In addition, we have performed intensive tests on the reforged OpenFPGA code base.

We have passed two benchmark suites, MCNC big20 and EPFL benchmarks, and add them to the regression tests.

Table 27: Detailed Plan on Code Reconstruction

Action Items	Status
Develop CircuitModel data structure (C++ object)	Completed
Develop a graph-based multiplexer representation MuxGraph(**New features to ease Verilog and Bitstream generation**)	Completed
Develop Verilog ModuleManager (**New features to ease hierachical Verilog generation w/o explicit port mapping**)	Completed
Reconstruct Verilog generation for primitive gates, such as multiplexers	Completed
Reconstruct Verilog generation for logic blocks using CircuitModel, MuxGraph and ModuleManager	Completed
Reconstruct Verilog generation for routing architectures using CircuitModel, MuxGraph and ModuleManager	Completed
Reconstruct Verilog generation for top-level netlists using CircuitModel and ModuleManager	Completed
Reconstruct Verilog generation for testbenches using CircuitModel and ModuleManager	Completed
Reconstruct bitstream generation using CircuitModel, MuxGraph and ModuleManager	Completed
Reconstruct Verilog generation for logic blocks using VPR8 data structure Netlist	Completed
Reconstruct Verilog generation for routing architecture using VPR8 data structure RRGraph	Completed
Legacy code removal and integration to VPR8	Completed

Finally, we have cleaned our code base by removing obsoleted codes related to VPR7 and updated documentation accordingly. We have enabled the best optimization level -O3 in compilation which provides the best run-time for users. Also we have paralleled our CI to reduce the run-time from 40 minutes to 15 minutes.

3.6.5 Yosys Integration

A critical bug has been fixed on the Yosys-VPR openfpga flow when using OpenFPGA shell, which is reported by the National Taiwan Ocean University [70]. After the bug fix, we now deploy 10+ test cases using the Yosys + OpenFPGA flow to CI, in order to avoid future bugs in the same category. These test cases cover full testbenches, pre-configured testbenches, and different configuration protocols.

We merged a pull request from QuickLogic which introduces their custom version of Yosys as a submodule [71]. QuickLogic also contributed makefile and compilation script so that the whole project can be compiled by a single command. This enables us to easily upgrade to future versions of Yosys without reworking on compilation scripts. Yosys is using a typical Makefile to compile while OpenFPGA and VTR are using CMake. To follow up the commercialization interest, QuickLogic keeps contributing their synthesis scripts and associated test suites to OpenFPGA and ensure that their device can be continuously supported in future [72].

3.6.6 Documentation and Communication

We have provided the OpenFPGA online documentation [1], short videos available on our YouTube channel [73] and, tutorials to provide more intuitive guidance for the users. Furthermore, to help users learn OpenFPGA's capabilities, we provide a technical highlights section to our documentation [74]. This is a summary page about the circuit designs, FPGA

architectures and tools that are supported by OpenFPGA, with links to complete examples.

Tutorials:

- An overview video about OpenFPGA. See details in [75].
- Tutorial about how to compile OpenFPGA. See details in [76].
- Tutorial about how to generate Verilog netlists. See details in [77].
- Tutorial about how to use user defined template.v. See details in [78].
- Tutorial about how to build FPGA with standard cells. See details in [79].
- *Frequently Asked Question* (FAQ) page, see details in [80]
  - Questions created from Github issues and pull requests,
  - Questions were approved after recommended changes were done,
  - FAQ page submitted to Github and was incorporated into documentation.

Videos:

- Video tutorial about how to define a new user defined template.v and published to Youtube [81], see details in [78]

3.7 T-9: Soft-core Architecture Exploration

In that task, we wanted to co-optimize FPGA architectures for better soft-core processors mapping, using OpenFPGA as our *Design Space Exploration* (DSE) tool. In order to adjust the different constraints and parameters of the architecture, we have set up an OpenFPGA compatible environment to generate the VPR architectures and the OpenFPGA tasks, and run various soft-core benchmarks. We have extended the benchmarking of the SOFA architecture by implementing the VexRISCV processor [82]. We investigated the different *Block RAM* (BRAM) memory compilers to use, as well as the effect of BRAM on the performance of SOFA [83].

3.7.1 Memory Compiler Evaluations

We decided to evaluate OpenRAM [84] and DFFRAM [85] memory compilers to understand which option was best suited for the design. OpenRAM creates more dense, optimized block RAM, while DFFRAM uses standard cells to make up the memory, which results in a simpler design that is easier to implement. From Table 28 and Table 29, we can see that DFFRAM has much less density than OpenRAM. After looking at how much area we have to fit the BRAM into the FPGA, we decided to use the DFFRAM because we have enough area available, and it is much simpler than OpenRAM.

Table 28: OpenRAM Memories

Size (bits)	Area (um^2)
512	12,137
1024	13,741

Table 29: DFFRAM Memories

Size (bits)	Area (um^2)
512	19,663
1024	37,617

3.7.2 BRAM Usage Evaluation

As presented in Table 30, we ran benchmarks to see how the BRAM integration might affect the utilization/performance of the VexRISCV processor [82] on the SOFA+ FPGA [86]. Thus, we can see that the BRAM helps to free up LUTs and DFFs that previously were being used for the register file in the processor. We traced the critical path in order to see what part of the processor was limiting the maximum frequency of the processor. In our breakdown, we found that in both the VexRISCV [82] and PicoRV32 [5] processors, the critical path was in the execute stage of the processor. After looking through the LUT mappings, we found that because there is no hard adder feature in the FPGA fabric, the tool uses a long chain of LUTs to perform the 32-bit address offset calculations.

Table 30: BRAM Improvements

	No BRAM	With BRAM	% Improvement
LUTs	2807	1067	+61.98 %
FFs	1652	509	+69.20 %
Critical Path (ns)	8.8	9.0	-2.30 %

3.7.3 Yosys and VPR Mapping

We have investigated how different architectural changes affected the placement and routing of the VexRISCV processor in the FPGA. For the baseline architecture, we used a k6 N10 FLE architecture with a hard adder carry chain, 18-bit multiplier, and 1 kB dual ported RAM. The performance of this architecture was then compared with other architectures that were the same, but with one of the components mentioned removed. This was done in order to test if that component was causing the placement and routing to change.

Yosys Adder Inference – In order to implement adder circuitry in the FPGA architecture, we first needed to properly synthesize the VexRISCV [82] processor benchmark. We found that in the current synthesis scripts, the addition performed by the processor was being mapped to soft adders (LUT logic), not the carry chain. In order for Yosys to use the carry chain in its synthesis, we had to create verilog files that outlined how to map the addition operations to the carry chain. We were successfully able to synthesize the processor, using the carry chain instead of soft addition.

VPR Carry Chain – We made sure that VPR took the BLIF file output by Yosys and properly mapped those synthesized carry chain cells to the carry chain described in the VPR architecture. We modified the VPR architecture to incorporate the carry chain as we described in the Yosys mapping files, then did the same for the OpenFPGA architecture files. We were then able to verify that the carry chain was properly implemented in the entire flow, from Yosys to VPR. As shown in Tab. 31, for smaller benchmarks, it seems that the carry chain causes performance degradation, like the critical path in the `8bit adder` benchmark or the CLB usage in the

counter 16bit. However, on a more complex and larger design like the i2c master, we can see almost a 20% improvement in speed with minimal CLB usage increase.

Table 31: With Carry Chain

	CLBs	LUTs	Max Frequency (MHz)
8bit adder	-4	-11	-8.38%
counter 16bit	+1	-8	+0.56%
i2c master	+1	-16	+19.78%

3.7.4 Soft-core Exploration Platform

As our previous work focused on the PicoRV32 [5] and VexRiscv [82] processors, we intended to extend to other open-source 32-bit RISC-V cores for better support of FPGA architectures, such as SERV [87], IBEX [88], or RI5CY [89]. After a first feasibility study, we found that VPR is limited when adjusting the design mapping on the FPGA target, and cannot modify the post-synthesis design generated by Yosys, which leads to a lack of architectural flexibility. In order to adjust the different constraints and parameters of the architecture, we have set up an OpenFPGA compatible environment to generate the VPR architecture and the OpenFPGA tasks, and run various soft-core benchmarks.

Design Space Exploration Platform

We continued our effort to co-optimize FPGA architectures for better soft-core processors mapping, using OpenFPGA as a *Design Space Exploration* (DSE) tool. We propose an environment platform as a Python API, wrapping the OpenFPGA framework tool, for fast and accurate design evaluation, as presented in Fig. 107. The main objectives of the platform are to provide: (1) a methodology to improve soft-core mapping and debugging, (2) placement and router analysis tools, to locate critical FPGA paths related to the soft-core description, and (3) a better domain-specific FPGA architecture based on the application requirements. In addition, we benefit from our previous SOFA [90] and SOFA+ [86] FPGA tape-outs to back-annotate the target FPGA architecture to increase the accuracy and reliability of routing analysis.

We then implemented a placement analyzer to extrapolate each path, which passes through physical blocks (CLBs, DSPs, BRAMs), and routing blocks (SBs, CBs), to analyze routing congestion in the soft-core design. Thanks to this path restructuring, we have improved the placement analysis performed by the OpenFPGA framework and extract path distances, constituting a first routing metric. Then, we developed a path builder tool that extends the description of each point composing the path in the implemented design. In order to provide a correct point naming and to group these paths into a common bus style, we developed a BLIF (*Berkeley Logic Interchange Format*) parser to link the physical mapping of FPGA blocks to the actual RTL instance names of the soft-core design. Yosys [**yosys˙github**] is configured with additional options to preserve the Verilog hierarchy naming after synthesis optimizations, even when using dedicated blocks, such as DSPs or BRAMs. Thus, the exploration platform was able to generate enough data and metrics to evaluate the impact of the VPR [**vtr8**] tool steps (packer, placer and router) and the impact of the soft-core architecture settings. Finally, we have restructured the tools and scripts for users to explore, analyze and improve the generation of custom FPGA architecture for soft-core mapping, as shown in Fig. 108. First, we proposed user-friendly launchers to run synthesis, packing, placing and routing steps for multiple set of

parameters to be explored, as referring the `run-dse` command. Second, results are parsed and extracted from the tools to provide an easier analysis of the design constraints and architecture requirements, as referring to `report-*` and `analyze-*` commands.

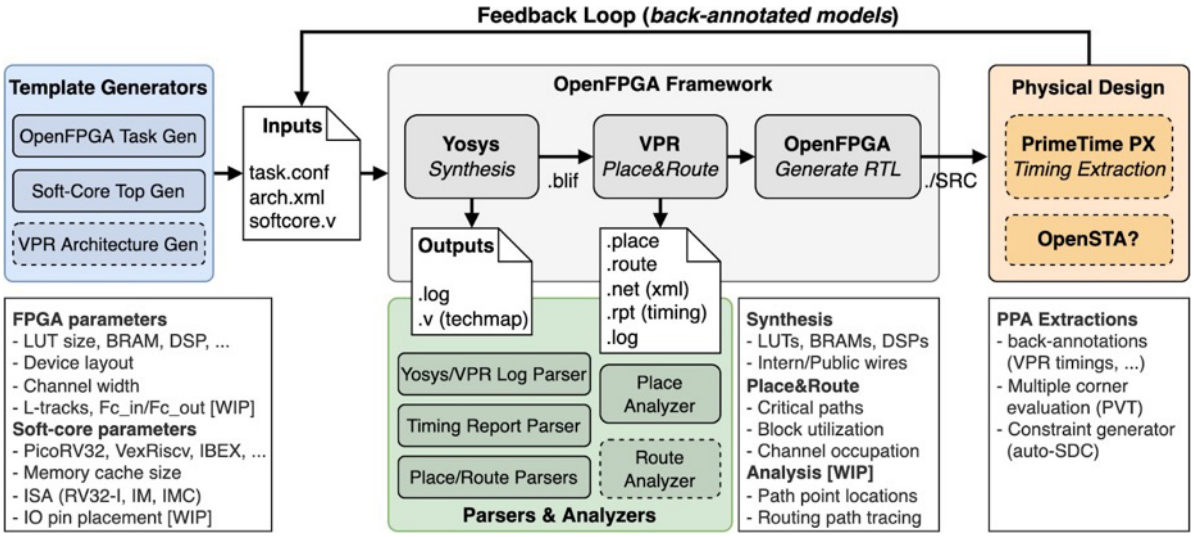


Figure 108: OpenFPGA Soft-cores Exploration Platform

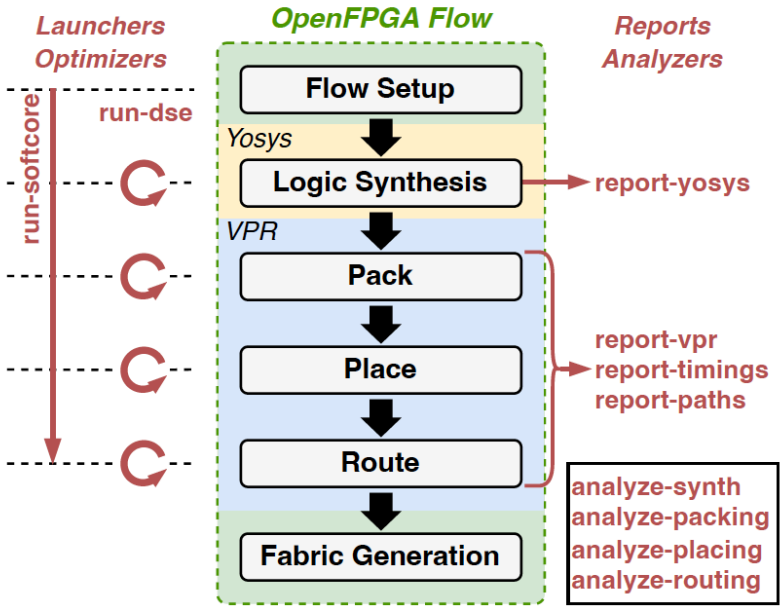


Figure 109: OpenFPGA Soft-core Exploration Platform

Soft-core Exploration Results and Analysis

In that section, we have evaluated the PicoRV32 [5] soft-core design on a 40 ×40 heterogeneous FPGA architecture with a total 100 kB of BRAM blocks and 36-bit DSP blocks. Thus, adding a dedicated DSP to accelerate the ALU improves the performance of the soft-core, but is now limited by the instruction decoder that uses BRAMs to store the address before fetching the instructions from memory. The critical path is between two distant BRAMs located at the edge of

the FPGA, demonstrating that the VPR placer has already achieved the best memory placement for the other paths, as detailed in the following paragraphs.

Packer and placer evaluations: We analyzed the variance of path arrival time between paths to evaluate design constraints. At first, we evaluate the packer stage by comparing the number of points (the logic equation required in the soft-core behavior) and its mapping to the physical blocks (PBs). As presented in Fig. 109, the total arrival time takes into account the memory read latency, which becomes the main bottleneck of the design mapped on the FPGA. Thus, additional constraints must be considered to overcome this limitation, using a design constraint file providing fixed pin Input/Outputs (IOs) and BRAM placements.

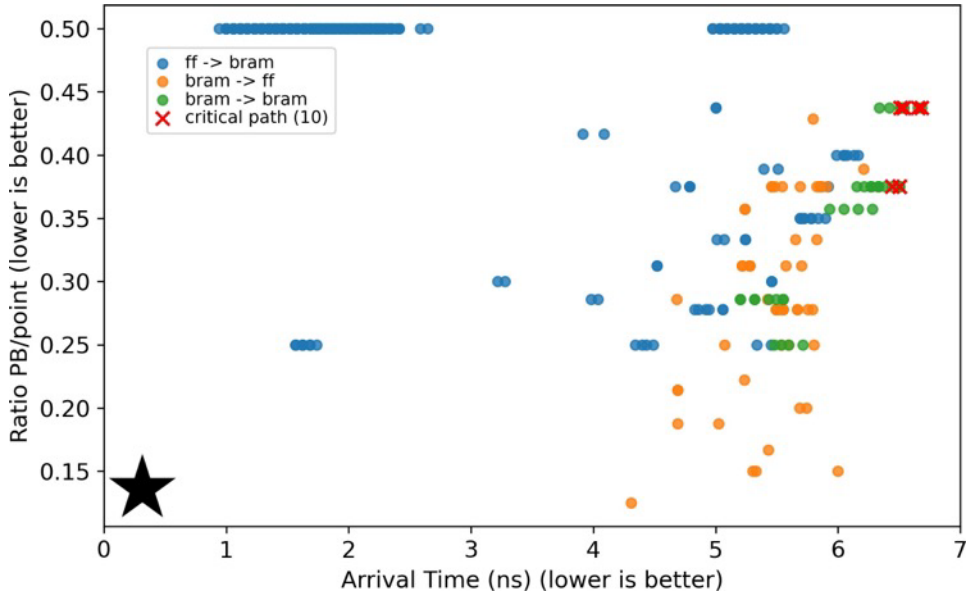


Figure 110: Packing Analysis of a PicoRV32 [5] Soft-core Map on a 40x40 FPGA using 64kB of BRAM Memory and one 32-bit DSP Multiplier
Best results are lower in both axes.

In a second analysis, we evaluate the placer and router stages, comparing the best Manhattan distance to the PB-to-PB distance. As presented in Fig. 110, the critical distances between paths are always BRAM-to-BRAM paths, mainly due to memory read latency and BRAM placement. That limitation is related to the FPGA architecture, whose BRAM placement is restricted, implying a significant data transfer between the most distant memories located at the edge of the FPGA.

Soft-core design parameter evaluations: Fig. 111 shows all paths routed on the FPGA according to their arrival times. Related to the PicoRV32 hardware description, the current critical bus is located between memory.mem and `cpu.cpuregs.regs` which are physically mapped on BRAM blocks. Thanks to this analysis, the user is able to improve its design, either by changing the FPGA architecture by adding more dense memory or by changing the PicoRV32 hardware description.

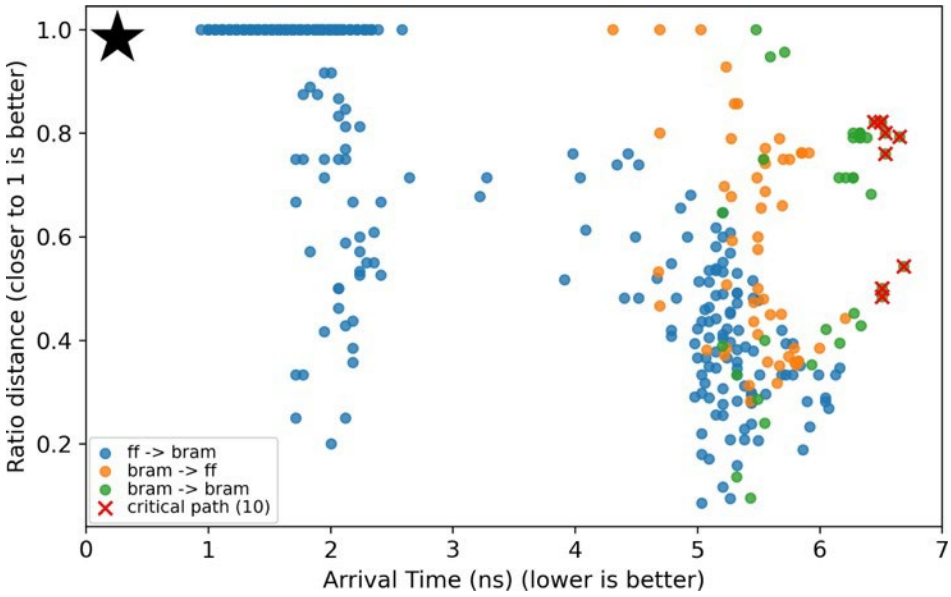


Figure 111: Physical Block Placement and Routing Analysis of a PicoRV32 [5] Soft-core Map on a 40x40 FPGA using 64kB of BRAM Memory and One 32-bit DSP Multiplier
The evaluation is performed considering the Manhattan distance as the best path mapping, referring to 1 in Y-axis.

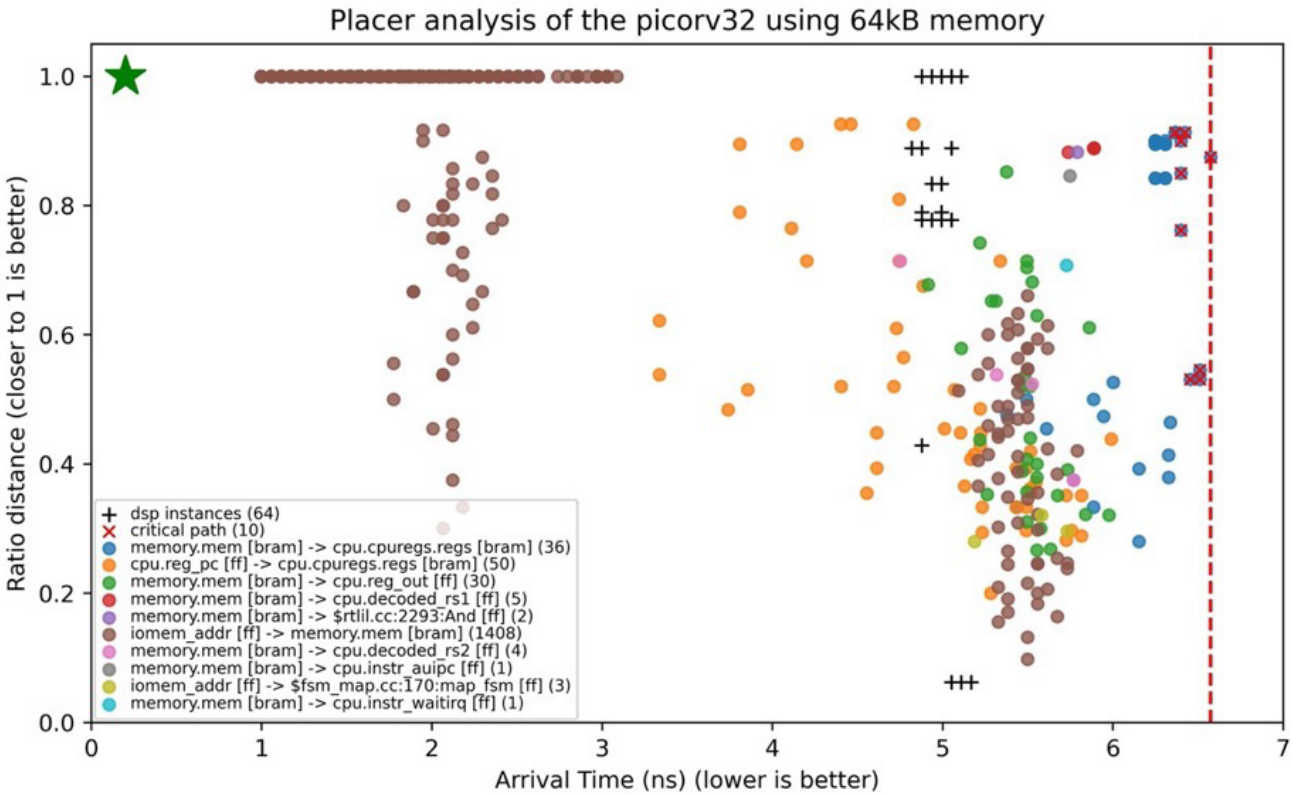


Figure 112: Post-routing Path Analysis of the PicoRV32 Core on the 40x40 Heterogenous FPGA

Futhermore, we can focus on this critical bus analysis by changing the soft-core memory size, from 16 kB up to 96 kB, as shown in Fig. 112. Thus, we confirmed that the critical path is still

located on the same bus (memory.mem ⟶ cpu.cpuregs.regs) with no regards on the memory size of the softcore. In this example, the FPGA architecture need to be adapted to improve the soft-core performance (related to the operating frequency). The proposed platform helps the user to better understand how the OpenFPGA platform implement a given benchmark on a select FPGA architecture.

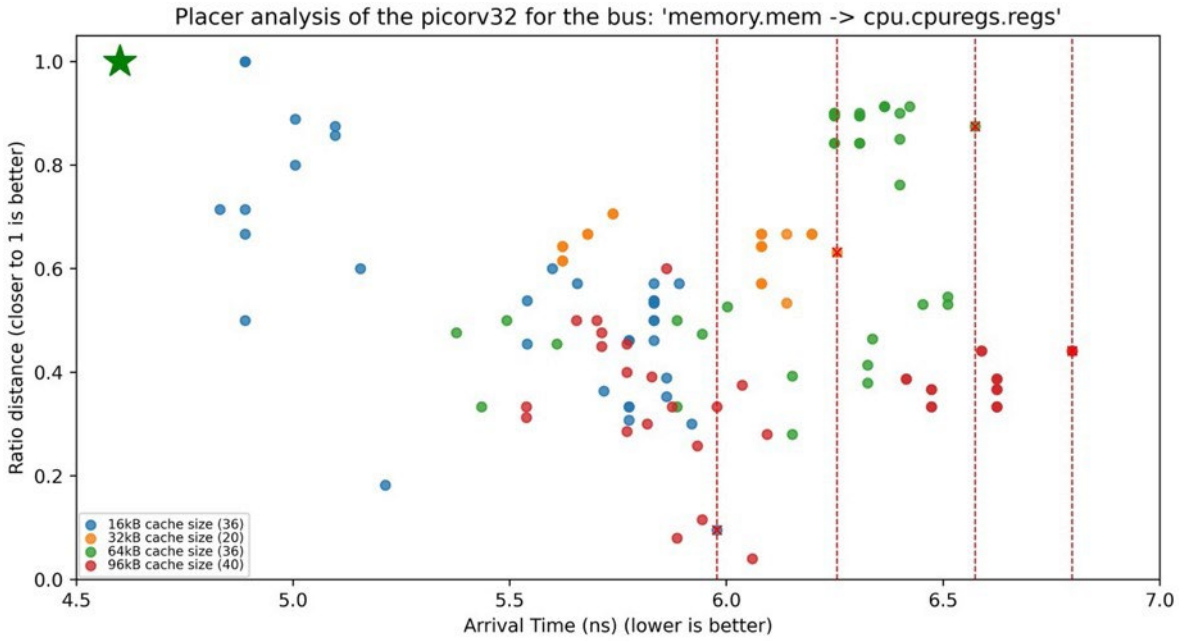


Figure 113: Post-routing Critical Path Analysis for Various PicoRV32 Data/Instruction Memory Sizes

3.8 TA-1: Design and Tape-out of Heterogeneous FPGA Testchip

To study the FPGA architecture, in particular the *Basic Logic Element* (BLE) architecture, used in the test chip, we have compared the performance of two types of BLEs, which are k8 frac and k6 frac. The k8 frac, depicted in Fig. 113, is a BLE similar to Stratix-IV, which can implement 8 different operating modes. The k6 frac, depicted in Fig. 114, is a simplified version of k8 frac, which exclude the fracturable 4-input LUT. For the test chip, the k6 frac is easier to debug and test than the k8 frac, and is therefore an overall safer choice. In addition, we have performed some performance evaluations with twenty MCNC benchmarks and three practical benchmarks (such as PicoRV32 [5]) on both architectures. Our results show that considering MCNC benchmarks, the k8 frac is 2% better in critical path delay while 2% larger in area than the k6 frac. When considering practical benchmarks, the k6 frac is 20% better in critical path delay while 14% better in area than the k8 frac. As a result, we select the k6 frac in the test chip.

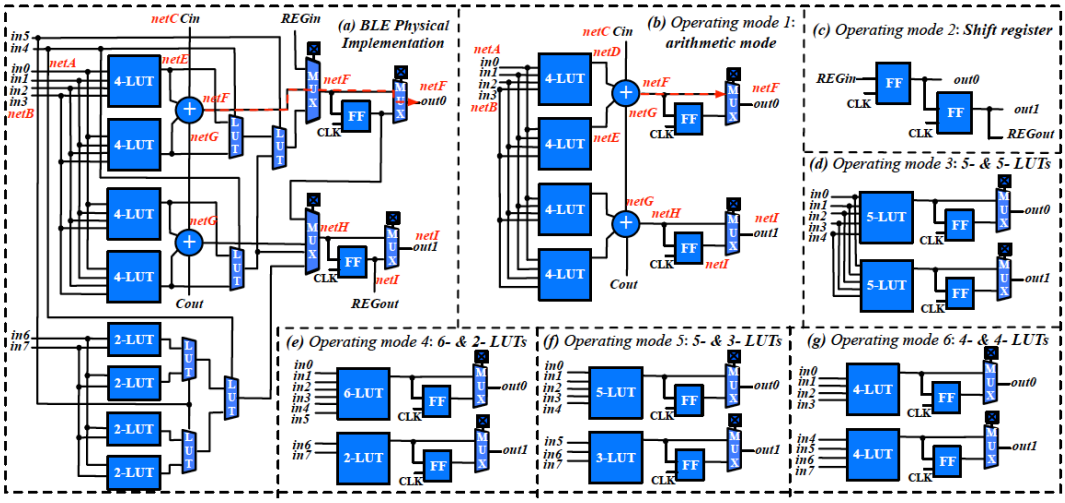


Figure 114: k8_frac BLE Architecture Consisting of a Fracturable 6-input LUT and a Fracturable 4-input LUT

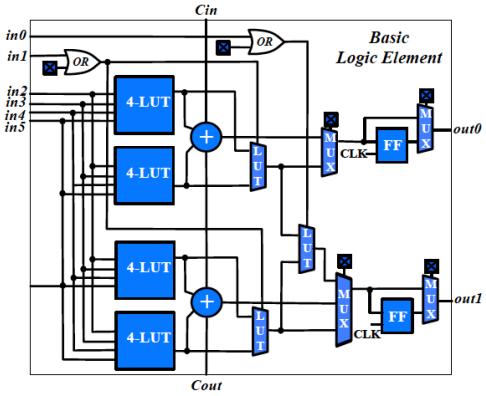
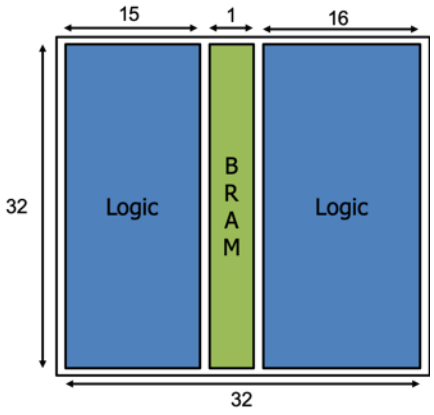


Figure 115: k6_frac BLE Architecture Consisting of a Fracturable 6-input LUT

3.9 Fabric Definition

We have resized the FPGA fabric subjected to the updated area budget (= $3mm \times 3mm$) for our test chip. As shown in Fig. 115(a), we implemented an array of 32×32, where a column of BRAM sits in the center of the fabric. The array size is determined by considering the area budget, fabric tileability (to guarantee minimum number of unique tiles) and BRAM density (being competitive to Xilinx Kintex-7).



(a) FPGA fabric organization.

Resource	Capacity
LUTs [1]	9.92 k
Arithmetic units [2]	19.84 k
Flip-flops	19.84 k
Block RAM [3]	512 kbit
User I/O pins	124
I/O voltage	1.8 V
Core voltage	0.8 V

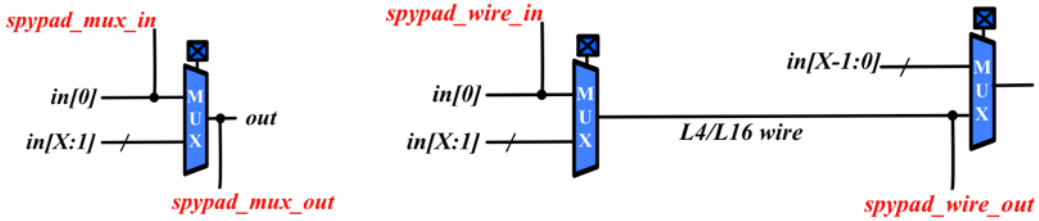
[1] Counted by 6-input fracturable LUTs
[2] Counted by 1-bit full adders
[3] Include sixteen 32kb memory blocks

(b) Logic capacity summary

Figure 116: Detailed FPGA Fabric for Test Chip

3.9.1.1 Spy-pads Addition

We add three types of spy pads, in the purpose of testing the propagation time of multiplexers, long routing wires and LUTs. The spypads in Fig. 116(a) are designed to measure the delay of a multiplexer in the local routing architecture of CLBs, while the spypads in Fig. 116(b) are designed to measure the delay of a multiplexer in the global routing architecture. The spypads in Fig. 117 are designed to capture the delay of different LUT outputs. To capture the performance difference due to process variation, the spypads were added to a center tile of the fabric and a fringe tile of the fabric. The exact number and location of spypads will be defined once the number of GPIOs are finalized, with respect to the area budget.



(a) Spypads for multiplexers. **(b)** Spypads for routing wires.

Figure 117: Spypads for Multiplexers in Fabric

Configuration chain: We add three spy pads to the configuration chain, mainly for functional tests, as illustrated in Fig. 118. Spypad[0] aims to test if the connection between *Configuration Chain Flip- Flops* (CCFFs) are working. Spypad[1] aims to test if the CCFFs of global routing architecture are working. There could be a long metal wires connecting the tail of CCFFs of routing architecture to the head of CCFFs of CLBs. It is meant to check if the connection is fine. Spypad[2] aims to test if the CCFFs of CLBs are working. There could be a long metal wires connecting the tail of CCFFs of CLBs to the head of CCFFs of IOs. It is designed to check if the connection is fine.

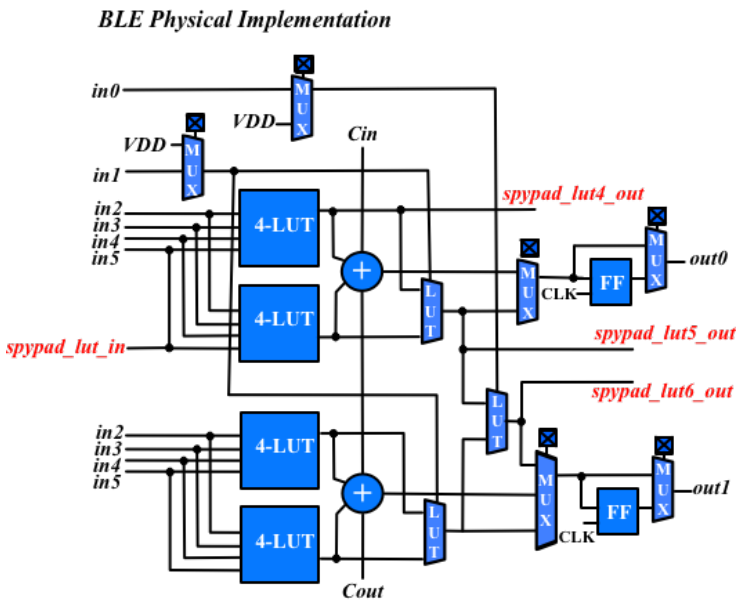


Figure 118: Spypads for Fracturable LUTs

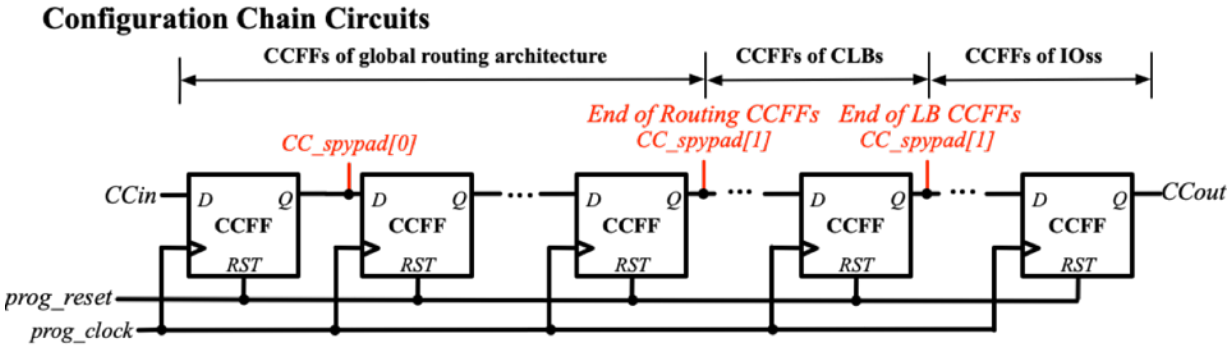


Figure 119: Spy-pads for Configuration Chain

CLB: We add spy pads to two CLBs in the test chip: (a) CLB[0][0] sitting in the bottom-left corner of the chip; (b) CLB[0][31] sitting in the the top-left corner of the chip. This choice is based on the following consideration: (a) CLB[0][0] is the head of our configuration chain. In case the configuration chain is partially working, we can still perform testing on a CLB. (b) CLB[0][31] is the head of our scan chain. In case the scan chain is partially working, we can still perform testing on a CLB. For each CLB, we add spy pads to all the outputs of a fracturable LUT in the first BLE. In addition, we add spy pads to the carry chain, shift register and scan-chain outputs of a CLB, which is used to read back results of small arithmetic benchmarks.

All the spy pads have been added to the top-level Verilog module of our test chip. We keep intensively testing our RTL design, which include spy pads and scan-chains. To accelerate the verification on configuration chain, we have adapted Synopsys VCS which is friendly to multi-threading (Modelsim cannot be multi-threaded). The HDL simulation (post-layout) takes 2.5 days to complete. This appears to be the longest verification we have to finish in each iteration of our test chip. To accelerate this process, we implemented SystemC netlists and started using Verilator.

3.9.1.2 Design for Test (DFT)

To guarantee testability of our test-chip, we have inserted a scan-chain to force/read back internal *Flip-Flop* (FF) bits, by exploiting our enhanced generator (see Section 3.5.5). To enable this, all the FFs in each BLE have been replaced by a scan-chain FF from the GF 12 nm standard cell library.

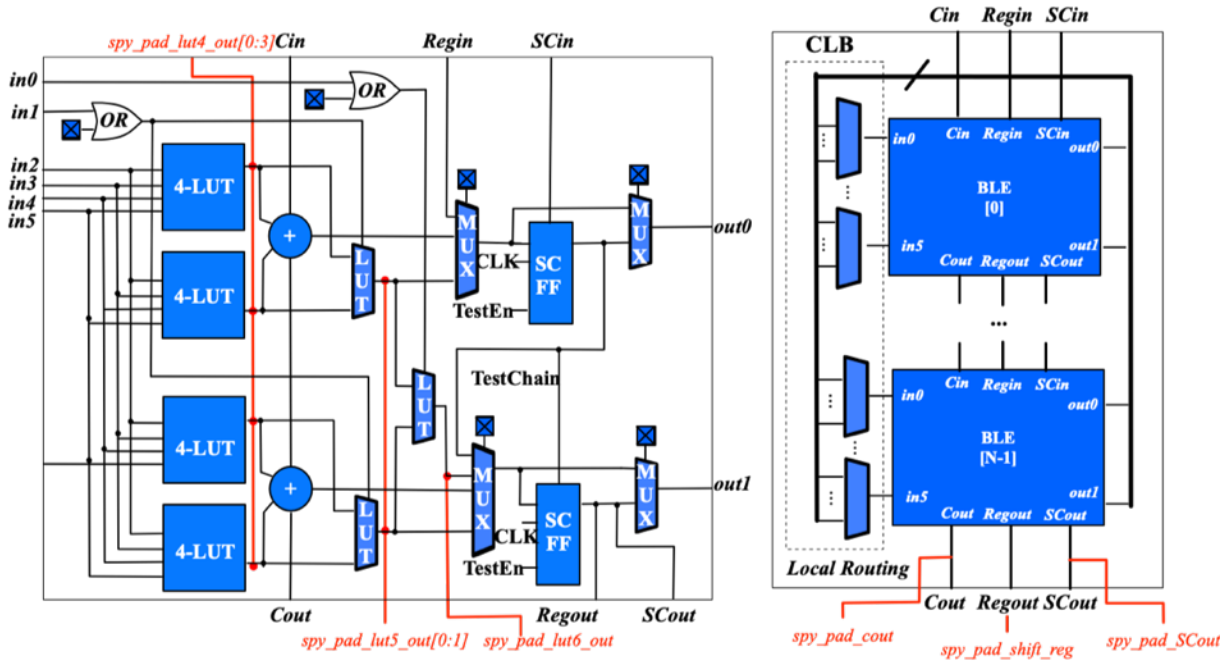


Figure 120: Spy-pads for CLB

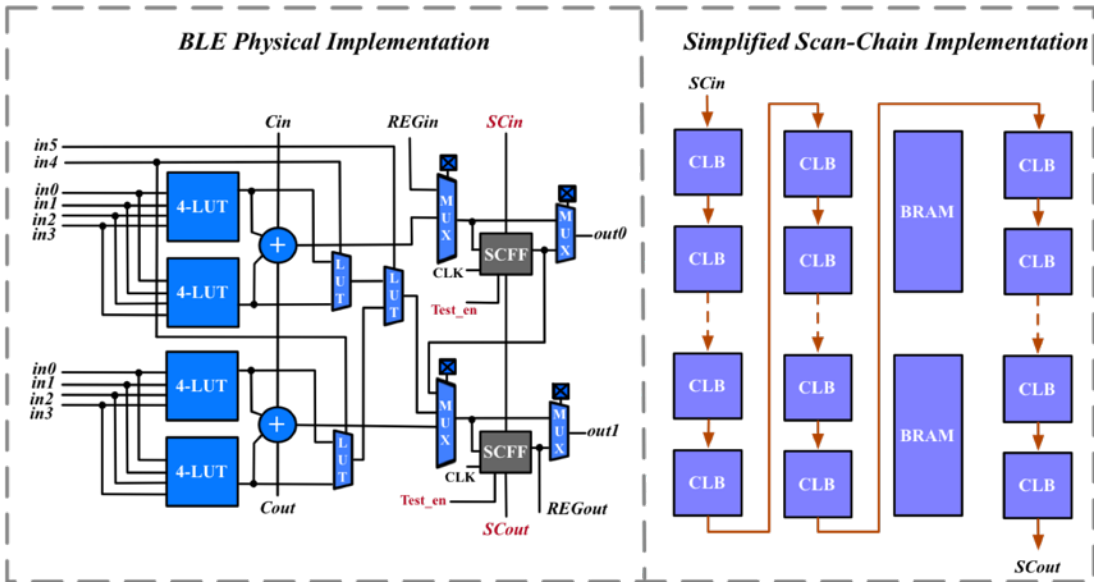


Figure 121: Design for Test CLB and Scan-chain Implementation

3.9.2 RTL Finalization for Version 2

While we were performing golden sign-off for version 1, we started preparing for the RTL for version 2. As version 2 uses the Verilog netlists generated by reforged engine, we have changed the spy-pad adding strategy to configuration chain. As shown in Fig. 121, we add five spy-pads to the configuration chain of FPGA, each of which have different testing purposes:

1. *cc spy0* aims to check the output of first CCFF,
2. *cc spy1* and *cc spy2* aim to check the completeness of I/O part of configuration chain,
3. *cc spy3* and *cc spy4* aim to check the completeness of programmable fabric part of configuration chain.

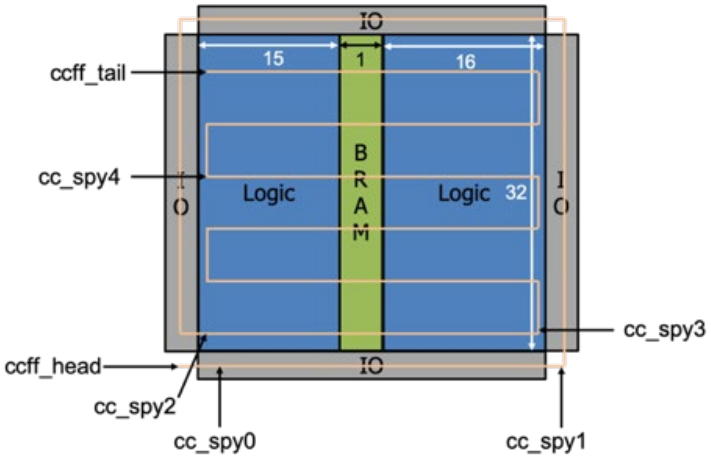
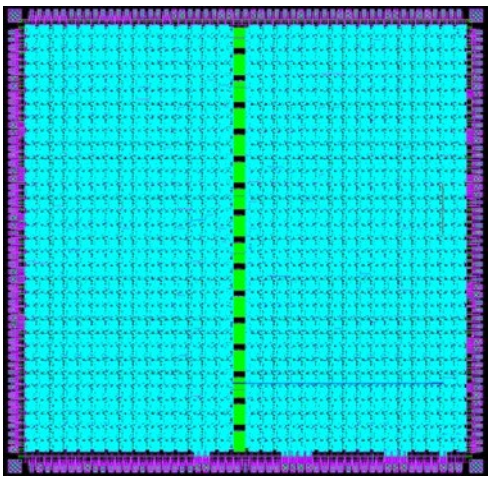


Figure 122: Spy-pad Addition to Configuration Chain

3.9.3 PnR Finalization

The hierarchical flow is separated into two sub-flows, the Design Planning and the Place and Route flow. As shown in Fig. 122, it is a 32 $\times$32 FPGA fabric, where the BRAMs are located in the middle column of the chip. Note that the height of a BRAM block is 2 $\times$of a CLB, the width of a CLB tile is very similar to a BRAM tile, as shown in Fig. 123. This area tuning guarantees a high-granularity FPGA implementation in the PnR flow. To ease the PnR, we have implemented each repeatable CLB, CB and SB as hard blocks, and then instantiated them in the top-level design. To relax routing congestion in the top-level design, we forced a utilization rate of 80%. In low-level primitives, such as CLBs, CBs and SBs, an average of 85% utilization rate has been used. The total run-time of PnR flow for this 32 $\times$32 fabric is $\sim$12 hours, which improves by at least 50% as compared to our previous best record on a 20 $\times$20 fabric. We worked on the golden sign-off on version 1 of our test chip (see Fig. 122(a) for details), using Calibre.



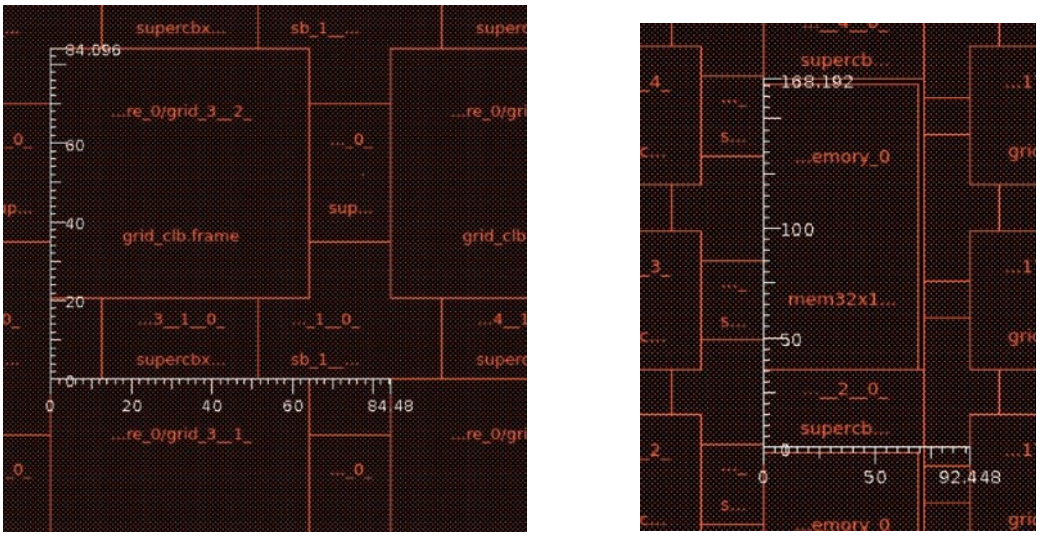
(a) Test-chip version 1.

Resource	Capacity
Look-Up Tables (LUTs)	9.92 k
1-bit full adders	19.84 k
Flip-flops	19.84 k
Memory size	512 kbits
User I/O pins	128
Total I/O pins	230
Max. configure speed	TBD
Max. operating speed	150 MHz
Max. I/O speed	300 MHz
I/O voltage	1.8 V
Core voltage	0.8 V

(b) Chip statistics.

Figure 123: Detailed FPGA Fabric for Test Chip

Following the tape-out delay, we started iterations to refine our PnR scripts to further guarantee stronger area and timing constraints. Note that the version 1 is based on our old Verilog and Bitstream generators. In the following versions, we switch to the reforged Verilog and Bitstream generators (See details in Section 3.5.1), because they are more generic, modularized and optimized for the needs of PnR tools. The primitive modules, i.e., CLBs, CBs and SBs are all DRC and LVS clean. We noticed issues at the top-level design, i.e., the FPGA fabric, because of the current high density. Therefore, we decided to reduce the size of FPGA fabric to 20×20, leaving enough margin between primitive modules, so that problems can be avoided. In addition, we have developed scripts to automate the I/O assignment for the test chip. Previously, the I/O assignment, including the power I/Os, was done by manually changing the Verilog netlists generated by OpenFPGA. We have automated this process significantly by using python scripts to generate the Verilog codes for the I/Os placement, such as: (1) defining the I/O locations in a spreadsheet with good visualization on the results, and (2) executing the script and append the Verilog codes to top-level netlist. This has significantly accelerated the development and avoid mistakes due to manual modification on netlists.



(a) Area of CLB (height = 84 μm, width = 84 μm). **(b)** Area of BRAM (height = 168 μm, width = 92 μm).

Figure 124: Detailed FPGA Tile Surface Area

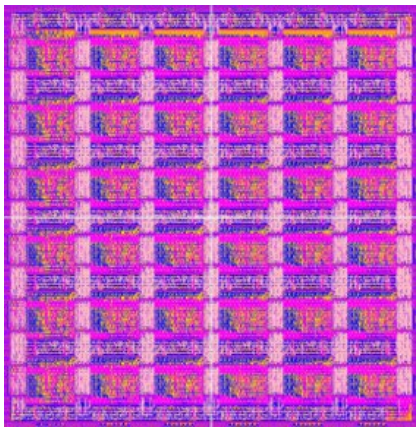
3.9.4 GF 12nm Tape-out and Sign-off Analysis

We have successfully taped-out our first test chip. We have submitted a GDSII of a fully instrumented 6 ×6 FPGA fabric, as shown in Fig. 124(a). It completed our entire sign-off strategy: all the Global Foundries' requirements on DRC, DRC+ and MAS, LVS, formal verification, timing sign-offs and IR- drop analysis, as listed in Fig. 124(b). We have developed all the scripts to run timing analysis and extract results at block-level (CBs, SBs and CLBs) and top-level (full-fabric). Table 32 reports the path delays we have achieved and compare to a previous work [46]. Both FPGAs are implemented with standard cell libraries but the previous work was using a TSMC 40 nm technology.

Table 32: Path Delay Comparison between Previous Works [46] (using standard cells implemented at 40nm) and OpenFPGA (using standard cells implemented at 12nm)

Path Type (Delay unit: *ns*)	**Previous Work (TT)** [1] [46]	**OpenFPGA (TT)**
6-LUT	0.50	0.23
20-bit Adder[3]	1.63	1.13
Local Routing[4]	0.27	0.15
L-4 track[5]	2.53	0.82
BRAM T_{setup}	-	0.32
BRAM T_{hold}	-	0.13

[1] TT: Typical-Typical corner.
[2] Adder paths start from a CLB input and end at a CLB output with local routing included.
[3] Local routing path starts from a BLE output and ends at a BLE input.
[4] L-4: FF →length-4 wire →Local Routing →LUT →FF.



(a) DRC-clean GDSII view on the 6 ×6 FPGA.

Action Items	Status
DRC	Clean. Waivers approved
LVS	Clean
MAS	Clean
DRC++	Clean
QRC for HDL simulations	SDF extracted and used in post PnR verification
IR-drop	Passed with 114mV drop when an activity of 65% is applied
Timing analyses	See details in Table 33, Fig. 125 and Fig. 126

(b) Detailed status on 6 ×6 FPGA layout.

Figure 125: Detailed FPGA Fabric for Test Chip

Thanks to the hierarchical physical design flow, the *Quality-of-Results* (QoR) of modules, i.e., CLBs, SBs and CBs, has been well constrained across the whole fabric. In particular, we see less than 10% variations on the programmable resources across the fabric, when SDCs are properly applied. Table 33 shows that the delay differences between LUTs are within 1% while the delay variation of local routing multiplexers are controlled under 30%. Fig. 125 and Fig. 126 show the delay variation of connection block routing multiplexers are within 5% and the delay of routing wires are controlled under 10%. Overall, we see less than 10% variation on the programmable resources across the fabric, with full respect on our SDC constraints. We repeat the same sign-off process done on the 6 ×6 FPGA, to ensure no performance degradation.

Table 33: Detailed Timing Results on CLBs

	Delay (*ns*)		
Path type	**Minimum**	**Average**	**Maximum**
4-input LUT	249.20	249.40	251.20
5-input LUT	277.27	277.48	279.19
6-input LUT	302.78	302.95	304.53
Local routing	146.40	190.55	251.57
20-bit adder	2186.17	2186.17	2186.17

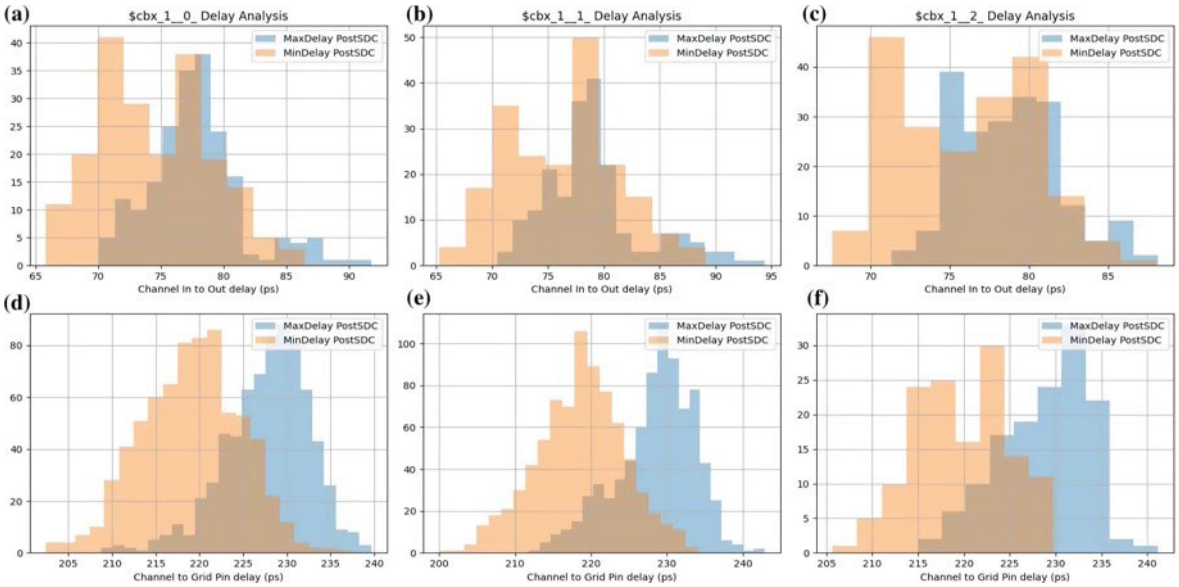


Figure 126: (a)(b)(c) Delay Distribution of X-direction Routing Wires and (d)(e)(f) Delay Distribution of X-direction Connection Blocks

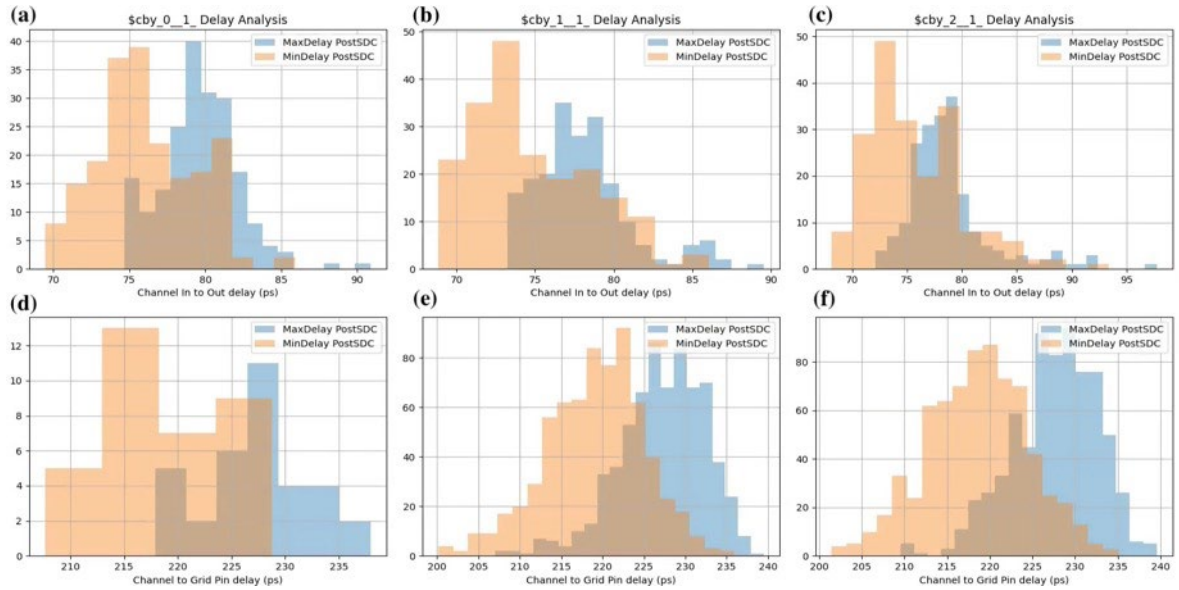
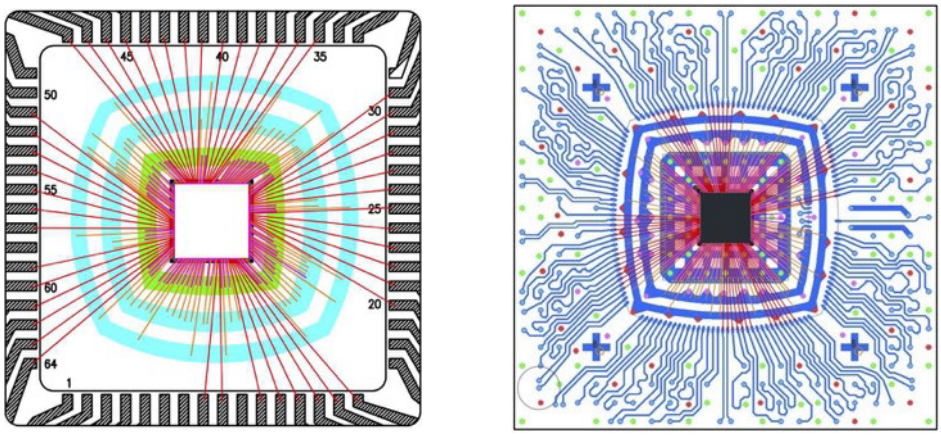


Figure 127: (a)(b)(c) Delay Distribution of Y-direction Routing Wires and (d)(e)(f) Delay Distribution of Y-direction Connection Blocks

In addition to the GDSII finalization, we have designed two types of packages, one is PGA and the other is BGA, and have sent to vendors for review. Technical details are summarized in the Table 34 according to their wire-bounding as illustrated in Fig. 127.

Table 34: Technical Details of Package Designs

Package Type	PGA	BGA
Vendor	SpectrumSemi	FabSierra Proto Express
Assemble service	QuickPak	QuickPak
Pin count	65	352
Quantity	20	20
Usage	Testboard	Demonstration Board
Wirebonding diagram	see Fig. 127(a)	see Fig. 127(b)



(a) PGA wire-bonding diagram. **(b)** BGA wire-bonding diagram.

Figure 128: Wire-bonding Diagrams

3.9.5 Skywater 130nm SOFA Tape-out

Physical design of Skywater Open-source FPGAs (SOFA) in three versions, such as: SOFA-HD (see layout in Fig. 128), QLSOFA-HD and SOFA-CHD is finished using the Synopsys IC Compiler II. We validated all the physical design techniques developed during previous (Clock tree pre-routing, Global signal pre-routing and abstract flow). Current run-time for 12 × 12 end-to-end automated design flow is <1 hour. All design cleared post-PnR functional verification. We prepared Skywater MPW submission repository with pre-checker passed [91–93]. Technical details are released at the online documentation of SOFA GitHub repository [94].

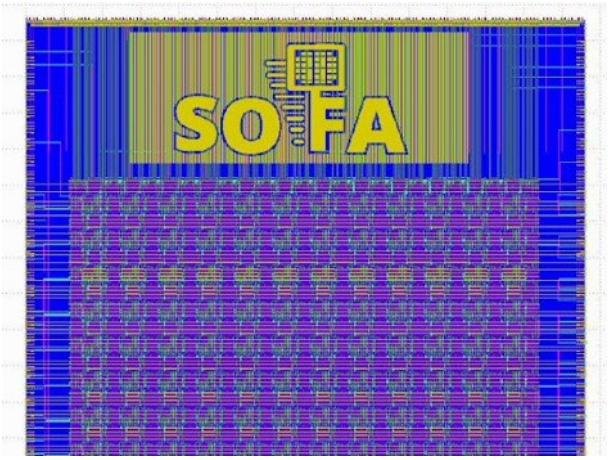


Figure 129: Layout View of SOFA HD FPGA IP

We have developed scripts for post PnR timing extraction, to evaluate the performance of the fabric. Fig 129 shows the L1, L2, and L4 delays of the QLSOFA architecture. The closeness of the delay numbers in different directions demonstrates the advantage of using the hierarchical design flow.

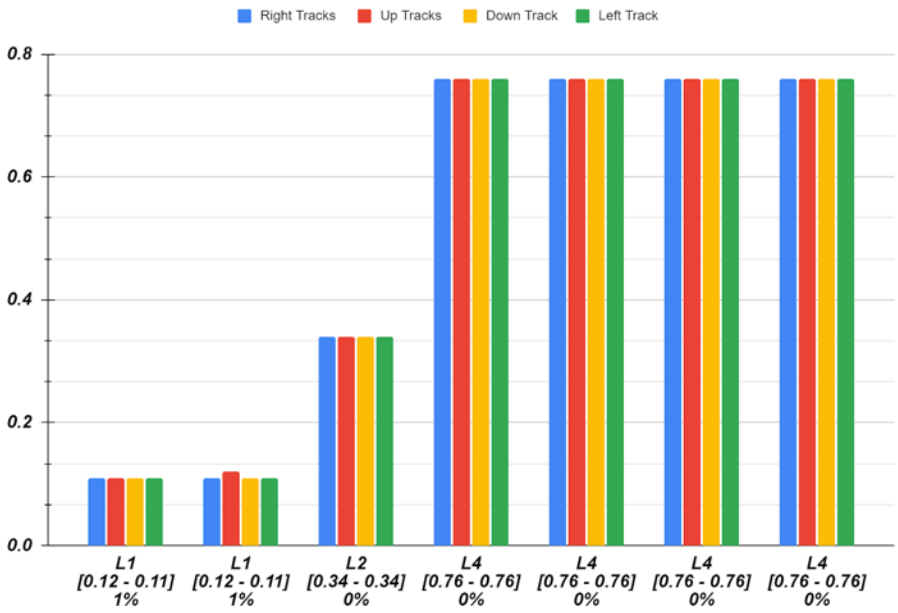


Figure 130: L1, L2 and L4 Delays of the QLSOFA Architecture Measured in all Different Direction

Chip-art design

We have developed a script to add a track based chip-art design. The script accepts the SVG, allows pattern fill from the given set of options, and creates metal track information. These tracks can be laid out using any PnR tool. Fig. 130 shows the Chip-art designed for the QL SOFA-HD and SOFA-CHD.

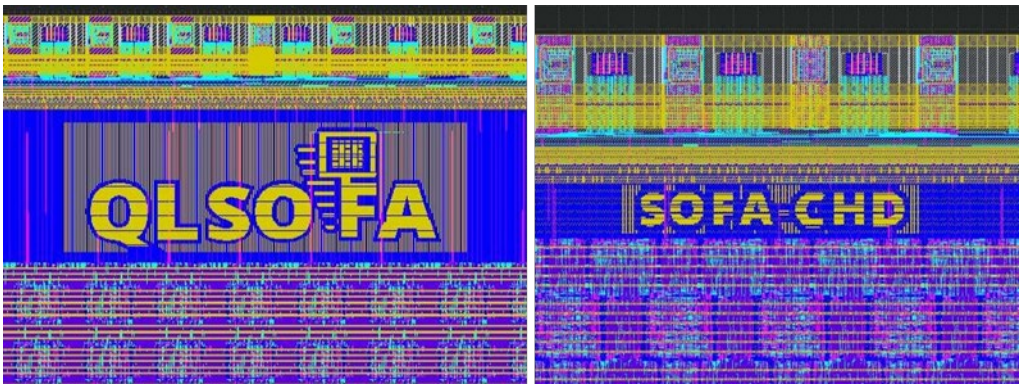


Figure 131: Track Based Chipart for QLSOFA-HD and SOFA-CHD

3.9.6 Skywater 130nm SOFA-Plus Tape-out

We have finalized the architecture for the SOFA-Plus eFPGA IP, which was submitted to the eFabless Skywater MPW-2 shuttle [95]. SOFA-Plus is the first open-source heterogeneous FPGA, which contains enriched DSP blocks, targeting arithmetic intensive applications. Fig. 131 illustrates the architecture of the SOFA-Plus eFPGA. Similar to other SOFA eFPGAs, the array size remains 12 $\times$12 due to the tight area budget of the MPW (3.5mm $\times$2.9mm). However, we have landed the following architecture features:

1. Two rows of *Configurable Logic Blocks* (CLBs) are replaced with DSP blocks. Each DSP block includes a 18 $\times$18 fracturable multiplier, which can operate as a 18 $\times$18 multiplier or two independent 9 $\times$9 multipliers;
2. Each CLB now has a fully connected crossbar to enhance the routability between logic elements;
3. Each D-type flip-flop can operate in different modes, where the reset signal can be either active- high or active-low.

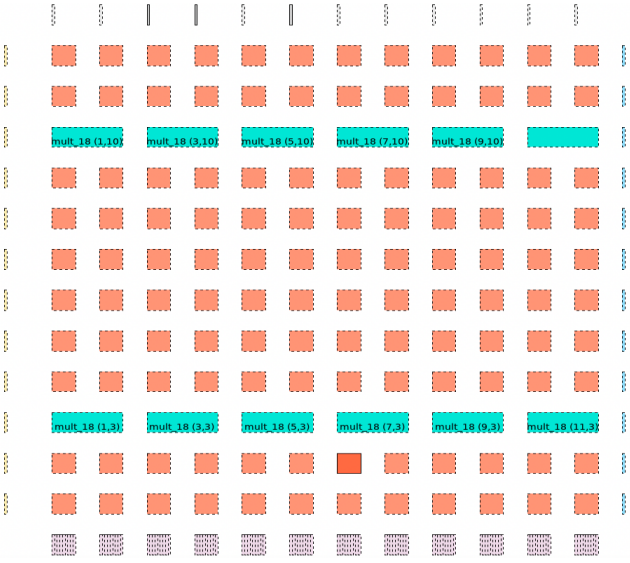


Figure 132: SOFA-Plus Architecture Overview
orange blocks are CLBs, while green blocks are DSPs

To enable the tape-out, our physical design scripts have been upgraded. In particular, the pre-routed clock/global signal network has been extended to support the heterogeneous blocks which may span multiple columns or rows, as depicted in Fig. 132. We have submitted the GDS of the SOFA plus eFPGA IP which is integrated to the efabless Caravel SoC. The layout in Fig. 133 has passed the DRC check, LVS check as well as the post-layout functional verification.

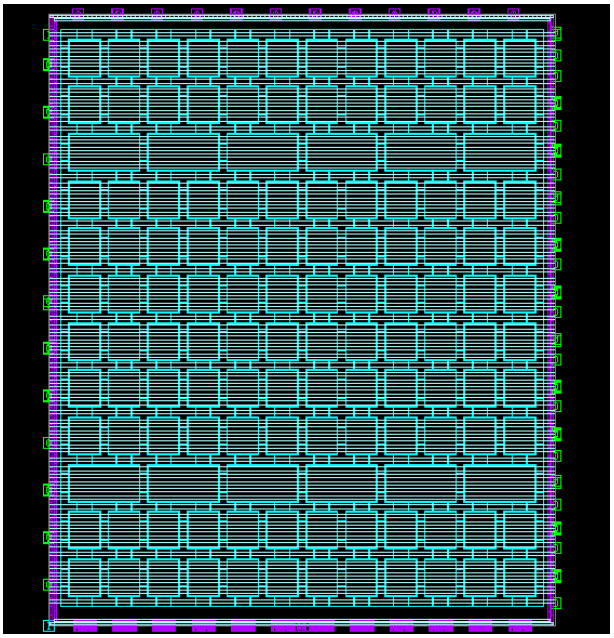


Figure 133: Floor-planning of the SOFA-Plus Layout

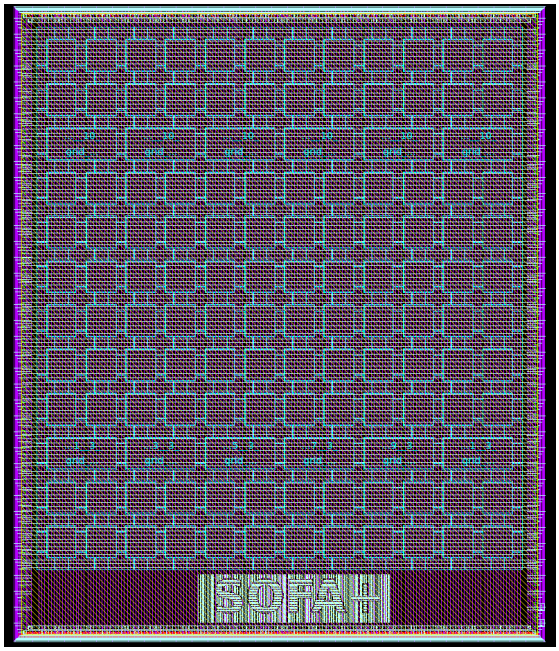
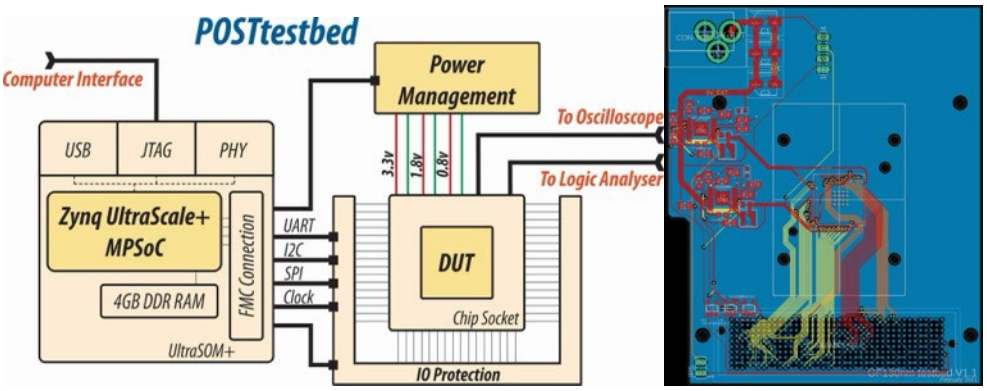


Figure 134: GDSII of the SOFA-Plus tape-out

In this tape-out project, we introduced the *Continuous Integration* (CI) system to perform pre- and post- layout simulations. Each time there is an update on the FPGA architecture definition or on netlists generation, the verification is automatically triggered on Github. It allows us to manage a tape-out project using cutting-edge CI techniques from software projects, enhancing the correctness and quality control on eFPGAs.

3.9.7 Daughter Board for FPGA Test-chips

We finished the test-bed board design for the GF 130 nm FPGA testchip, which can be easily adapted to the GF 12 nm testchip. The test-bed consists of a Trenz Electronics base-board controlled by a Xilinx MPSoC and a custom daughter board to interface with the socket as shown in Fig. 134(a). The power for the FPGA is provided by two synchronous buck converters with power sequencing, one for the core and one for IO. We interface with the daughter board through an *FPGA Mezzanine Card* (FMC) connector and have the option for separate BNC connectors for external probes. The layout for the PCB without BNC connectors is shown in Fig. 134(b).



(a) Block diagram of test-bed setup using UltraSOM+ base board. **(b)** Daughter board PCB layout.

Figure 135: Daughter Board PCB to Test FPGA Test-chips

3.9.8 Testing the GF 12nm Test-chip

Fig. 135 shows the testing environment setup, composed of the GF 12 nm FPGA socket, the daughter board, and the base-board. The Xilinx FPGA is used as a stimulus generator and the test chip outputs are read-back through a logic analyzer. We have developed a HDL design of the bitstream loader, which can be synthesized by Xilinx Vivado. In our chip testing, a Xilinx Zynq FPGA is configured as the bitstream loader for the FPGA under testing. We successfully powered up the GF 12 nm chip. Fig. 136 shows a graph of the power draw of the I/O and core of the FPGA. The VDD_{io} is set to 1.8V, while the VDD_{core} is ramped to 0.8V (blue line in Fig. 136). The I/O current reaches about 4.5 μA and the core current reaches about 800 μA. That is 1.5 $\times$what we expected, but still is very reasonable.

Configuration Chain Test: We clocked a single 1 into the head of the chain, then watched the 1 propagate across the debug pads on the FPGA. In Fig. 137, the yellow signal is the 1 on the head of the chain, and the blue signal is the first spy-pad on the configuration chain. This shows that the signal is propagating along the chain. During this process, we see a few hold violations in the configuration chain for the current chip under testing. The hold violations caused a few 1 in the

bitstream to be flipped when loading to the configuration chain. We have spotted the position of the hold violations in the configuration chain and developed a strategy in patching bitstreams.

LED Blinker Test: We use OpenFPGA to generate a bitstream for a LED blinker design. The bitstream is patched to resist hold violations in the configuration chain. The bitstream has been successfully loaded to the FPGA test chip. We see in Fig. 138 and Fig. 139 the expected blinking wave-forms at the desired I/O of FPGA. We confirmed the output waveform using an oscilloscope.

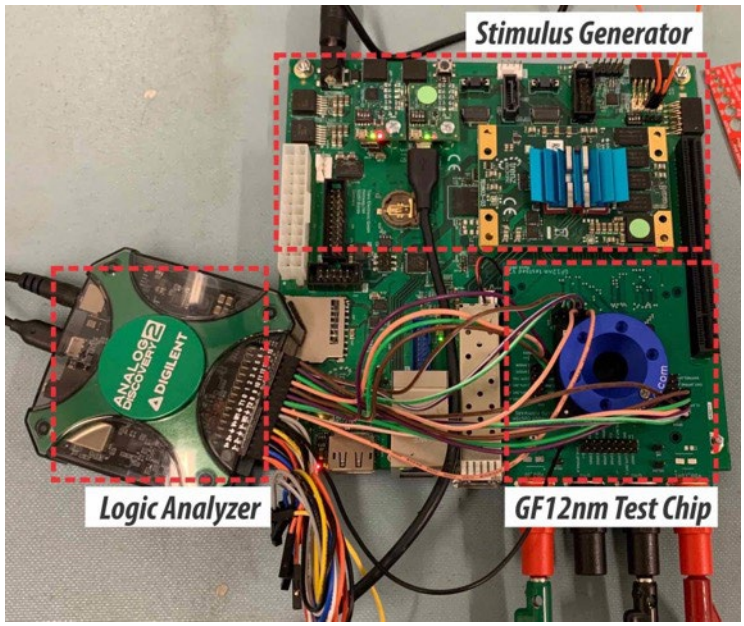


Figure 136: GF 12nm FPGA test-chip setup.

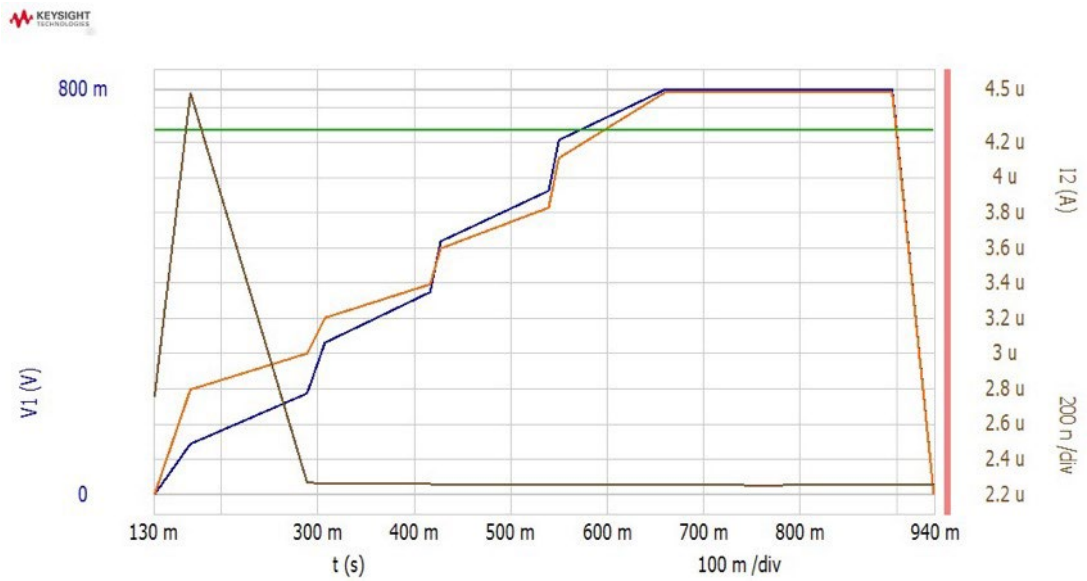


Figure 137: GF 12 nm FPGA Test-chip Power-up Current

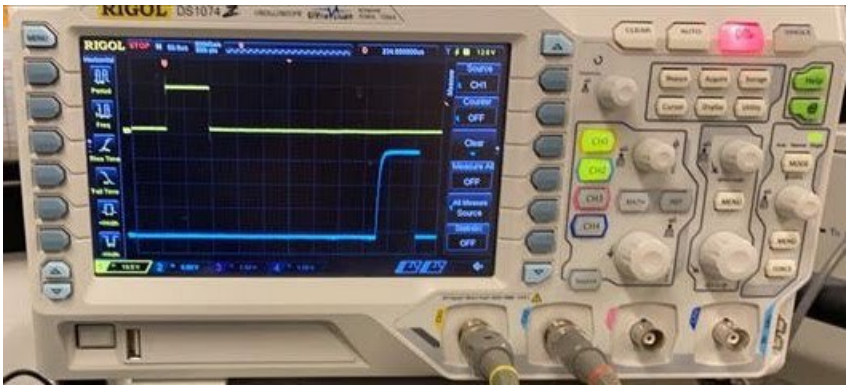


Figure 138: Configuration Chain Input and Output Pulses on an Oscilloscope

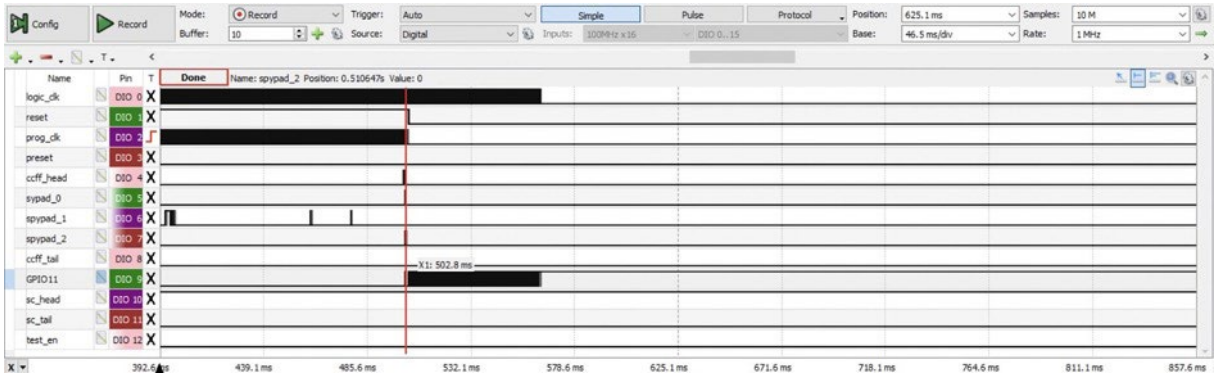


Figure 139: Complete Waveform Readback Through a Logic Analyzer on the FPGA which is Programmed to Implement a LED Blinker

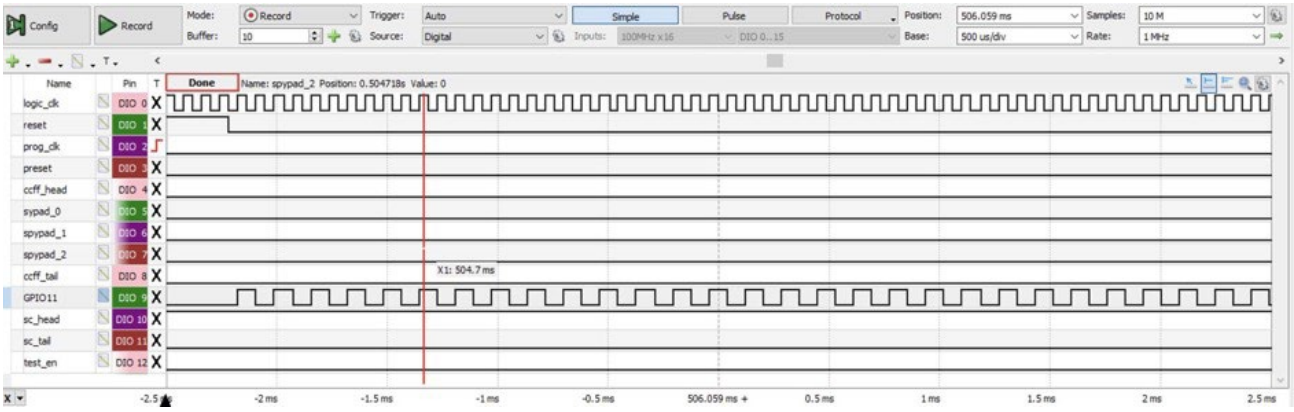


Figure 140: Blinking Waveform see at the GPIO11 of the FPGA After Programming

3.9.9 Testing Progress on GF 12nm Test-chip

We turned our efforts on patching the test bitstream to address the problem we have been having with hold violations on the configuration chain. To better understand the behavior of the hold violations, we ran several different experiments.

3.9.9.1 Voltage Tests

The first test was a voltage sweep to see how the VDD CORE voltage affected the number of hold violations on the configuration chain we saw. From this test, we found that the number of hold violations varies with voltage, and that a voltage near nominal would give the least amount of hold violations.

Table 35: Voltage Tests

VDD IO (V)	VDD CORE (V)	# Hold Violations
1.8	0.56	11
1.8	0.70	5
1.8	0.75	5
1.8	0.80	6
1.8	0.90	4
1.8	0.95	4
1.8	0.98	5

3.9.9.2 Chip Tests

We also tested all of the chips available to us. We had 21 packaged chips (other 20 dies are not yet packaged, which were reserved for demo board) including the chip we have been running tests on. The purpose of this test was to see if there was a different chip that has less hold violations on its configuration chain, due to process variation in the silicon. In table 36, we can see the five chips with the least amount of hold violations.

Table 36: Chip Tests

Chip ID	# Hold Violations
box0 0-2	2
box1 1-3	3
box1 0-1	3
box1 2-3	3
box1 0-3	4
box1 2-1	4

From these tests, we found that chip box0 0-2 would give us the least amount of hold violations and would be best to work with when trying to create a working bitstream. As a result, we managed to run a blinking test on the chip and create a demo video posted on LinkedIn. See [96] for details.

3.10 TA-3: Development of Practical Test Applications and open-source Demonstration Boards, and TA-4: Trouble Shooting and Community Support for the Demonstration Boards

Alongside the 12nm First Reconfigurable Open-source Gate-array (FROG) testchip tapeout, we have developed an open-source testbed which can be applied with minimal modifications for the testing of future tapeouts. This consists of a commercial FPGA board and a carrier card holding the FROG chip, along with a series of test designs which, when loaded on the commercial FPGA, evaluate the correctness and performance of the FROG testchip. All packaging information, PCB

designs, commercial hardware requirements, and testbenches are available on github in the FROG test infrastructure repository. As those are open source, and no request came from users, troubleshooting nor technical support have not been needed.

4 MANAGEMENT SUMMARY

4.1 Modularization Efforts

We have done the first VPR modularization meeting on March 12th, 2019. Following the agreed roadmap, our first attempt was to standardize an existing VPR data structure DeviceContext. As shown in Fig. 140, the structure containing a full set of FPGA architecture parameters and resource description is a standard input of VPR's packing, placement, routing and STA engines. We reorganize the content of the device database, with a particular focus on the *Routing Resource Graph* rr graph. A new class wascreated to encapsulate the routing data strūctures that exists in DeviceContext. Then, we started encapsulating the routing engine, i.e., the high-level interface RouteNetlist. The routing algorithms to be modularized include breadth-first and timing-driven. We expected the two examples provide the templates to modularize the other high-level interfaces.

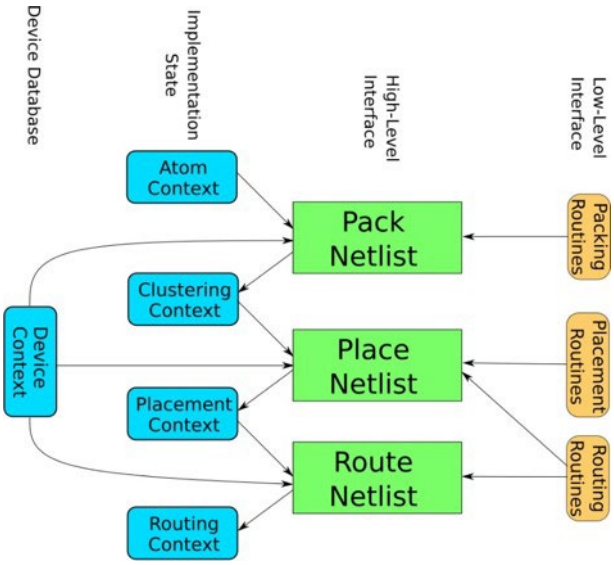


Figure 141: Modularization of Core Engines: Creation of Standard Database and Interface

Our pull request on modularized *Routing Resource Graph* (RRGraph) has been accepted by VTR project. We have done a follow-up pull request, which includes functions to convert a legacy RRGraph to an RRGraph object [97]. Collaborating with the leading VTR developers, we have improved by $\sim 2\times$ memory footprint for our RRGraph object, being similar to the classical data structure while containing more information. We have passed regression tests over both VTR and Titan benchmarks suites, which are the must-run tests for VPR code release[3]. We summarize the memory footprint benchmark by benchmark in a spreadsheet. Full technical details can be found on the pull request.

We have attended the VTR developers' meeting at the International Symposium on FPGA'2020.

[3] https://docs.google.com/spreadsheets/d/1aDcFZ3eo5SwTaVwxuGgf_fo0QTU8XieI0Uw0W_bvTm8/edit?usp=sharing

We have agreed with Vaughn Betz (University of Toronto) on multiple new features, which was contributed to VPR. Note that these features have already been implemented in OpenFPGA.

- Modularized RRGraph: This is one of the major efforts in refactoring VPR code base. As suggested by Tim Ansell, we reworked our pull request to push incremental changes.
- LUT truth table fix-up at post-packing: This feature is to fix the consistency between the truth tables of LUTs and packing optimization. VPR packer swaps the pins of LUTs to achieve better timing, but this causes that the truth table of LUTs are no long the same as original designs. Without this feature, the bitstream generation is wrong. We tested the correctness of the fix-up and pushed to VPR once it is ready.
- Pack results fix-up at post-routing: This feature is to fix the consistency between the pin placements and routing optimization. VPR router swaps the pins of CLBs/heterogeneous blocks to achieve better timing but causes bitstream generation to be wrong. We tested the correctness of the fix-up and pushed to VPR once it is ready.
- Invisible modes for packer: This feature aims to help users debug more efficiently when using multi- mode CLBs. For example, one or more modes of a CLB causes problems in VPR packing stage. Users can modify the XML file to tell VPR packer not to consider these modes. As such, users can quickly spot which mode is the source of problems. We pushed this feature as soon as OpenFPGA
  + VPR8 integration is tested.

4.1.1 Integration Initiative to VTR8

We have forked the VTR Github repository [19], where we implemented a new class `rr graph` and tested it in the framework. A new class `rr graph` has been created with useful functions to access the data structure, such as `rr graph builder` and `rr graph area`. Due the large number of functions, the workload is higher than expected. Guided by the VTR team and collaborators, we define several milestones and do a pull request to the VTR repository when a milestone is achieved. We had a conference call with Vaughn Betz about the detailed technical plan on moving to VTR8. We have agreed on a loose integration, which keep OpenFPGA and VTR8 as separate as possible. As such, both teams can benefit from new features while keep engineering effort low. In addition, Vaughn has agreed to bring two features of OpenFPGA into VTR8 with tight integration, which are the post-routing truth table adaption and post-routing packing result corrected[4]. All updates have been pushed to the dev branch of the OpenFPGA github project, where we perform massive regression tests.

4.1.2 Code Contribution to VTR Upstream

Invited by VTR developers, we have contributed to the VTR project, and have the following pull request accepted and merged to the main branch of the project:

[4] https://docs.google.com/document/d/1V10ONAcDm-jEWjb7FnXEFrwnXeE6FPzqvOhvwQpJAC4/edit?ts=5cde097e#

- **VTR pull request 1309** [98]: OpenFPGA relies on this feature to accelerate packing algorithms, which also eases the debugging for VPR users on multi-mode configurable logic block architectures. In addition, we also contribute tutorials about how to use this features for debugging.
- **VTR pull request 1355** [99]: Post-routing netlist synchronization. This feature carries a high value to VPR upstream, without which bitstream generation could be wrong.
- **VTR pull request 1448** [100]: Bug fix on the RRGraph generator. We have detected a critical bug in VPR router, where two nets could be mapped to the same routing resource node in their Stratix IV-like architecture. The pull request contained the following updates:
  1. A critical patch on the RRGraph generator,
  2. A patch on the Stratix IV-like architecture and another flagship architecture.
- A patch on the golden results.**VTR pull request 1621** [101]: This pull request is a patch for the VPR upstream code, which aims to fix the bug in RRGraph representation for the Stratix-IV architecture.
- **VTR pull request 1661** [102]: This pull request is a follow-up patch where the new APIs is deployed to the graph user interface and router functions. As a result, the Stratix-IV architecture can be displayed consistently with internal representation, and its QoR can be improved.
- **VTR pull request 1742, 1747, 1667, 1693 and 1694** [103–107]: These pull requests are refactoring a new node `side()` APIs for the RRGraph builder and router, without which a 2-10$\times$ overhead in critical path delay may be seen in Stratix-IV architectures [108].
- **VTR pull request 1800 and 1801** [109, 110]: The refactoring RRGraph data structure. With these pull requests, we have reached the first milestone in the refactoring effort, where we successfully reconstruct the node look-up data structure, an essential component of the RRGraph.

4.1.3 LeWiz Integration

LeWiz indicated their interest for integrating the OpenFPGA fabric in their products and we discussed a possible very easy path forward, which would require an option to (1) generate an "eFPGA" by removing the I/O blocks and creating a top module that provide direct channel access; and (2) access the configuration chain FFs through a word instead of a single-bit to maximize programming speed. Lewiz has reached out to us with an intention on implement their MAC core using OpenFPGA. We have provided a project template based on our latest code base and send them detailed guidelines.

4.2 Skywater 130nm Tape-outs

Google funded us to participate a coming tape-out using the open-source Skywater 130nm PDK. We designed the FPGA architectures, which includes innovation that we developed in past few years, such as MCluster [111] and pattern-based local routing [112] *etc*. We finished the architecture exploration for a coming tape-out using the open-source Skywater 130nm PDK, and

we worked on the back-end implementation. Later, we completed an alpha version of layout for the tape-out using the open-source Skywater 130nm PDK. It is an embedded 1.2k-LUT4 FPGA fabric interfacing a RISC-V processor in an SoC system. Our tape-out effort, including the scripts without any confidential Tcl commands, architecture description, GDS2, LEF and other results, are open-source in a public github repository [13, 83]. We believe that this helps the marketing of OpenFPGA and gain more public exposure. Finally we completed three versions of FPGA layouts which are ready for fabrication (DRC clean, LVS clean and passed sign-offs) using the open-source Skywater 130nm PDK. The three FPGA IPs are optimized for different capacity and performance requirements:

- **SOFA HD**: A low-cost high-density standard-cell FPGA IP with 1.2k-LUT4 and 2.3k registers,
- **QLSOFA HD**: An improved version of SOFA HD which adapts the soft-adder logic from Quick- Logic's AP3 device architecture,
- **SOFA CHD**: An improvised version of QLSOFA HD which is built with custom cells for high- performance routing multiplexers.

All the SOFA IPs have been merged to the layout of Caravel SoC [23]. The three versions of SoCs have passed the efabless pre-checks required by the Skywater 130nm MPW tape-out run. Reaching this stage, we finished the MPW effort using Skywater 130nm PDK. All the SOFA IPs including datasheets are now released on the project's repository [83]. We made a release on the *Skywater Opensource eFpgAs* (SOFA) project [83]. The release includes the production-ready GDSII files of three eFPGA IPs, which are all compatible to the Caravel SoC [113]. Datasheets are available online [13].

SOFA-Plus tape-out: As eFabless launched the second *Multiple Project Wafer* (MPW) shuttle (see details in [95]), we have participated in the project with the SOFA+ FPGA. This is an upgraded version of the released SOFA IPs. We have adopted heterogeneous FPGA architecture where DSP and BRAMs can be integrated. We have submitted the GDS of *Skywater Open-source eFpgAs* (SOFA) IP, called SOFA-Plus for the second MPW shuttle hosted by efabless (see details in [86, 95]).

RRAM contribution on the SkyWater 130-nm PDK: We have submitted RRAM test structures (GDS, SPICE, Schematic) to the Google repository [114], to extend the available structures, such as 1T1R, 4T1R, and 4T1R MUX2 designs. Other cells on this repository are not complete, see details in [115].

4.2.1 Intel 22nm FFL Tape-out

We kicked off our tape-out project using the Intel 22nm FFL technology. Since we submitted the SOFA-Plus GDS, we summarized our experience on previous tape-outs, including the *Global Foundries* (GF) 12nm chip. Meanwhile, we started the architecture for the Intel 22nm FFL tape-out, to upgrade the existing SOFA plus architecture by adding block RAMs. We mostly put on hold the physical design of the tape-out because we had to focus on the GF 12nm chip testing and the Skywater tape-out.

4.3 QuickLogic Collaboration

We have collaborated with QuickLogic to use OpenFPGA to create embedded FPGA IPs. Quick- Logic is interested in using OpenFPGA to accelerate their development in embedded FPGA IPs at 40nm and more advanced technology node. A technical plan has been drafted on the tape-out practice for a Delaware FPGA architecture. We have implemented a first version of VPR architecture XML that models the Delaware architecture, including the logic cells, I/O interfaces, DSP and BRAM. We have tested the VPR architecture file with QuickLogic's benchmarks through the SymbiFlow. We have verified all of the logic resources that are defined in the architecture, including Look-Up Table, Flip-flop, I/O, DSP and RAM, using QuickLogic's micro-benchmarks. QuickLogic showed interests in commercializing the FPGA IPs that we built for the Skywater 130nm MPW program, based on the fact that their logic element architecture is widely implemented in the SOFA FPGA IP family. QuickLogic was investigating how to optimize the FPGA IPs by introducing hard IPs, e.g., BRAM and multipliers, as well as configuration protocol, in order to satisfy their customers. In particular, QuickLogic is coordinating the integration of FASM [116] to OpenFPGA as their bitstream assembler for the FPGA Ips.

We supported QuickLogic to deliver an embedded FPGA IP for their customers. As the production of eFPGA requires high testability and multi-clock support, we upgraded OpenFPGA in this direction. We have implemented a 32 $\times$32 homogeneous FPGA design. The device is intended to be used in a signal processing industry as an interface device. Using our proposed PnR technique, the netlist modification has been performed, and performance numbers such as routing delay, logic delay, clock and global signal latency have been extracted from Skywater 130nm PDK. The number has been carefully extrapolated to understand the performance at UMC 22nm tech node, which is a technology of choice for the final tape-out. Finally, to enable long-term support for Quicklogic's devices, Quicklogic contributed 20 micro-benchmarks to OpenFPGA [117]. The micro-benchmarks are continuously tested in OpenFPGA's CI, ensuring the correctness of Quicklogic synthesis scripts and device functionality.

We have worked with the Quicklogic teams to integrate the OpenFPGA development methodology in various design stages. The netlist post-processing script is amended to perform the following:

- Auto-generate configuration chains in each row of the FPGA (max 32 chains in parallel),
- Add "design for testability" related features like scan and debug mode. The scan mode amends the existing configuration chain with data flip-flops and creates 4 global chains. Whereas debug mode loops back all the scan-chains, allowing probing the fabric status, but restoring the fabric's logical status at the end,
- Functional verification-related efforts are ramped up by integrating CocoTB-based testbench in the existing flow. It performs all the peripheral verification. The CI/CD workflow is implemented for the ModelSim-based verification to allow continuous development of the architectures and back-end methodologies.

The QuickLogic eFPGA tape-out project is in the final stage. The finalized grid array is 24 ×24. We managed to implement a dual voltage domain "Always on" and a Core power domain, and we created an automated script to handle the placement of the cells in the specific region on the chip. The power grid creation has been improved to generate supply ports on the top of the design. The .lib and .lef files are generated to be shared with the customer for further integration. In June 2021, the UMC 22nm eFPGA design has passed all the sign-offs, such as timing closure, IR-drop analysis, and power evaluation required by the customers.

4.3.1 Google Collaboration

We set up monthly calls with Google to track the latest progress on VPR and Symbiflow. We worked out a detailed plan on merging technical features especially RRGraph objects to VPR. In addition, Google would like to tape-out an iCE40-like FPGA on the Skywater 130nm technology using the OpenFPGA technology.

Moreover, to disseminate our work and platform to the open-source community of hardware developers lead by Google teams, we addressed the following tasks:

- **On-the-self FGPA macro tiles for SoC designers:** We have proposed GDS-ready macro tiles for faster System-on-Chip (SoC) developments. This task involves timing-closure automation scripts for various tile size in order to create a database of macros. Finally, we have pushed all *Skywater Open- source FpgA* (SOFA) files on Google git servers [118], thus providing mandatory files (GDS, LEF, SPEF, and post-pnr verilog files) to enable eFPGA hard-IPs using OpenFPGA prototyping tool.
- **FuseSoC integration for SoC designers:** We target user-friendly integration using an existing package manager, such as FuseSoC [119], which is probably the most likely method for getting people to embed macros into their designs. We have defined all SOFA hard-IPs as a ”core” of FuseSoC in order to seamlessly integrate them into any future SoC design. In the same time, we prepared a set of basic benchmarks so that users can deploy their own design on SOFA/SOFA+ architectures using the FuseSoC framework.
- **EDAlize integration for SoC designers:** EDAlize is a Python wrapper for interacting with various EDA tools. It can generate project files for supported tools and run them in batch or GUI mode (where supported). As of mid-february, we have integrated an OpenFPGA task generator back-end for EDAlize, in compliance with the developer standards. Users can evaluate various Verilog test-benches on our four different SOFA/SOFA+ architectures [86, 90] using EDAlize’s Python launchers. The pull request of this OpenFPGA backend has been accepted and merged into the master branch of the EDAlize project, see details in [120].

4.3.2 New York University (NYU) Collaboration

In parallel to our secured FPGA bitstream loader (PMU) proposal, we have started a collaboration with the FPGA security team of the *New York University* (NYU) to propose a secured FPGA’s auxiliary IP. This IP can be defined as a firmware, to continue upgrading the programming options and configurations post-manufacturing, thus, enhancing the current FPGA programming configurations. The end user can choose, a secure programming (or not) via a

JTAG link or directly with an on-chip memory close to the FPGA. We have adopted a red/blue team approach to evaluate the implemented countermeasures, in order to improve the reliability, safety, and security of the target FPGA configuration protocol. We validated and tested our proposed PMU between our universities on the following aspects: the passage of messages between the on-chip memory and the FPGA core and during interrupts of the JTAG protocol. This can improve the safety, security, and reliability of the target FPGA configuration protocol.

5 CHALLENGES AND ISSUES

We still have challenges for the OpenFPGA framework to provide additional features for the end user.

5.1 OpenROAD Integration

We noticed that the designs generated by OpenFPGA are generally compatible with the OpenROAD flow, but it also revealed a few major issues:

- Most static timing analysis tools, including OpenSTA, cannot analyze FPGA designs. This limitation is fatal to the OpenROAD flow, causing it to lock up during the global placement, ”replace”, global routing, and timing analysis steps. As a result, it is not possible to run the OpenROAD flow on an FPGA design without removing some OpenSTA checks. We found a more stable way to work around this issue.
- The place and route steps are completely unconstrained, so the resulting GDS file is disorganized and sub-optimal (Fig. 141).
- We need to understand how to mark tiled modules as macro blocks in a way that OpenROAD can understand. The macro placement step also relies on fixing OpenSTA to work with the FPGA designs.
- As Google shows a strong interest in tape-out with the Skywater 130nm PDKs, we also investigated the OpenLane scripts (which are well tuned for the PDK) as an alternative to the generic OpenROAD flow.

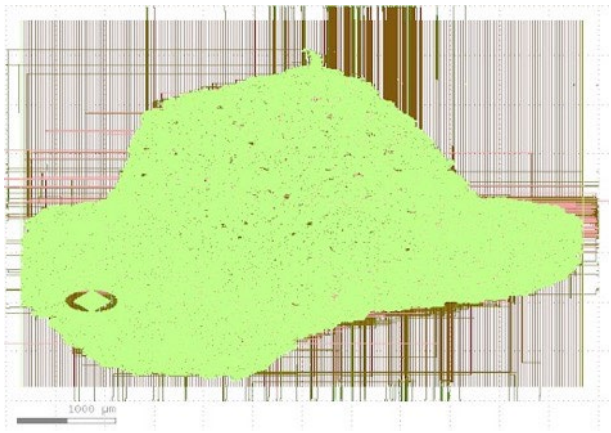


Figure 141: The Unconstrained PnR Tools Fail to Generate a Tiled Design, which Leads to a Poor Final Result

We also implemented a 2×2 homogeneous FPGA design using the OpenLane flow [121], with rectangular macro cells arranged in a tiled pattern. Once a reliable automated RTL-to-GDS flow is realized, we created vanilla OpenROAD TCL scripts to support the same behavior. Macro-level I/O placement was a challenge, because OpenROAD only supports pre-hardened rectangular macro cells. It is possible to create rectilinear macro cells with some small workarounds (Fig.

142), but we focus on finding a way to automatically place local I/O signals adjacent to their counterparts in neighboring cells.

5.2 Verification Support with Commercial EDA Tools

We have seen a large gap between open-source HDL simulator and commercial ones, especially in runtime and syntax support. Even though our *Continuous Integration* (CI) covers the Verilog-to- Verification flows using iVerilog [68], we cannot deploy large FPGA fabric in the tests due to their high run-time cost. We see limitations in our CI because we cannot integrate commercial tools due to license issues. However, in our tape-out projects, commercial tools, e.g., Mentor ModelSim is widely used. This caused a lot of recurring engineering effort in our tape-out projects.

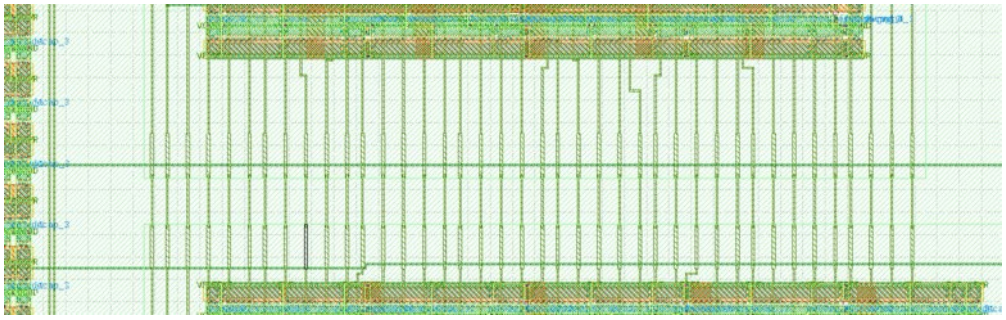


Figure 142: Aligned I/O Signals between Adjacent Macros

6 CONCLUSION

During this four-year project working on OpenFPGA, the program was successful in achieving its goals to provide a fully functional homogeneous FPGA core IP and bitstream generator, an automatically-generated heterogeneous FPGA core IP fabrics with secured bitstream loading protocols integrated in the FPGA IP framework, and compare performance to commercial FPGAs.

All open-source code was made available as initially planned (https://github.com/lnis-uofu/OpenFPGA), and functional chips were created and shared. To this date, the project received 570 forks and 126 stars on GitHub therefore demonstrating a wide and successful respond to the community engagement.

The POSH infrastructure is currently used by multiple companies, namely Quicklogic, RapidSilion, RapidFlex, Efablesss, as well as routinely used in academic tape out for domain specific and radiation tolerant FPGAs.

7 PUBLICATIONS

Publications accepted and published during the period of the funding are as follows:

1. X. Tang, E. Giacomin, G. Micheli, Giovanni and P.-E. Gaillardon. "FPGA-SPICE: A Simulation-Based Architecture Evaluation Framework for FPGAs". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 2018. 10.1109/TVLSI.2018.2883923
2. X. Tang, E. Giacomin, B. Chauviere, A. Alacchi and P. -E. Gaillardon. "OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs". *IEEE Micro*. 2020. 10.1109/MM.2020.2995854
3. X. Tang, E. Giacomin, A. Alacchi, B. Chauviere, P.-E. Gaillardon. "OpenFPGA: An Opensource Framework Enabling Rapid Prototyping of Customizable FPGAs". 2019. 10.1109/FPL.2019.00065
4. X. Tang, E. Giacomin, A. Alacchi and P. Gaillardon. "A Study on Switch Block Patterns for Tileable FPGA Routing Architectures". *International Conference on Field-Programmable Technology (ICFPT)*. 2019. 10.1109/ICFPT47387.2019.00039
5. X. Tang, G. Gore, E. Giacomin, A. Alacchi, B. Chauviere and P.-E. Gaillardon. "OpenFPGA: Towards Automated Prototyping for Versatile FPGAs". Workshop on Open-Source EDA Technology (WOSET). 2020. https://woset-workshop.github.io/
6. G. Gore, X. Tang and P.-E. Gaillardon. "A Scalable and Robust Hierarchical Floorplanning to Enable 24-hour Prototyping for 100k-LUT FPGAs". *ACM International Symposium on Physical Design (ISPD)*. 2021. 10.1145/3439706.3447047
7. A. Alacchi, E. Giacomin, X. Tang and P. -E. Gaillardon. "Smart-Redundancy: An Alternative SEU/SET Mitigation Method for FPGAs". *IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021. 10.1109/ISCAS51556.2021.9401092
8. X. Tang, G. Gore, G. Brown and P.-E. Gaillardon. "Taping out an FPGA in 24 hours with OpenFPGA: The SOFA Project". *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*. 2021. 10.1109/FPL53798.2021.00085
9. A. Alacchi and P.-E. Gaillardon. "Programmable Local Clock SET Filtering for SEE-Resistant FPGA". *IEEE Transactions on Circuits and Systems II: Express Briefs*. 2022. 10.1109/TCSII.2022.3172390

The following papers were submitted but not accepted during the period of the project:

- A. Alacchi, X. Tang, P.-E. Gaillardon. "LUT-RAM Study on FPGA architectures". *IEEE International Conferences on Field Programmable Logic and Applications (FPL)*. 2020.
- G. Gore, J. Sandrini, E. Giacomin, G. Molas, X. Tang, E. Nowak, P.-E. Gaillardon. "A High-Performance and Ultra-low-power Monolithically Integrated RRAM-based FPGA for IoT Edge Computing". *IEEE Symposium on VLSI Technology and Circuits (VLSI)*. 2020.
- A. Alacchi, E. Giacomin, X. Tang and P.E. Gaillardon. "SEU Sensing with In-Memory ECC Checking for Interuption-based FPGA CRAM Reconfiguration". *Radiation and its Effects on Components and Systems (RADECS)*. 2021.
- G. Gore, X. Tang, A. Pond, S. Charas, N. Page, A. Alacchi, E. Giacomin, B. Chauviere, T. Saxe and P.-E. Gaillardon. "A 12nm FinFET-based FPGA designed with Open-source Unified FPGA Compiler". *2022 IEEE Custom Integrated Circuits Conference (CICC)*. 2022.
- G. Gore, J. Sandrini, E. Giacomin, G. Molas, X. Tang, E. Nowak and P.-E. Gaillardon. "A High-Performance and Ultra-low Power RRAM-based FPGA for Edge Computing Applications". *IEEE Custom Integrated Circuits Conference (CICC)*. 2022.

8 REFERENCES

[1] "OpenFPGA online documentation". https://openfpga.readthedocs.io.

[2] "OpenFPGA GitHub Project". https://github.com/lnis-uofu/OpenFPGA.

[3] "SpyDrNet physical documentation". https://ganeshgore.github.io/spydrnet-physical/.

[4] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/latest/openfpga_shell/openfpga_commands.html#fpga-sdc.

[5] "PicoRV32 Github Repository". https://github.com/YosysHQ/picorv32.

[6] J. H. Kim and J. H. Anderson. "Synthesizable FPGA fabrics targetable by the Verilog-to-Routing (VTR) CAD flow". *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, 10.1109/FPL.2015.7293955.

[7] "OpenFPGA Project Issue". https://github.com/lnis-uofu/OpenFPGA/issues/319.

[8] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/328.

[9] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/348.

[10] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/339.

[11] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/349.

[12] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/master/manual/fpga_bitstream/generic_bitstream.html#file-format.

[13] "SOFA: Skywater Opensource FPGAs Online Documentation". https://skywater-openfpga.readthedocs.io/en/latest/.

[14] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/manual/file_formats/bitstream_setting/.

[15] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/243.

[16] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/311.

[17] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/294.

[18] "Versatile Place and Route (VPR) Documentation". https://docs.verilogtorouting.org/en/latest/vpr/.

[19] "Verilog-to-routing Github Project". https://github.com/verilog-to-routing/vtr-verilog-to-routing.

[20] "OpenFPGA Online Documentation". https://github.com/lnis-uofu/OpenFPGA/blob/dev/.travis/script.sh.

[21] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/148.

[22] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/149.

[23] "Caravel SoC". https://github.com/efabless/caravel.

[24] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/307.

[25] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/147.

[26] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/circuit_model_examples/#standard-fracturable-lut.

[27] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/circuit_model_examples/#native-fracturable-lut.

[28] "OpenFPGA Pull Request: Support on Using Scan-chain Flip-flop in a Configuration Chain". https://github.com/lnis-uofu/OpenFPGA/pull/167.

[29] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/config_protocol/.

[30] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/fabric_key.html.

[31] G. G. F. Lemieux, S. D. Brown, and D. Vranesic. "On Two-step Routing for FPGAS". *Proceedings of the 1997 International Symposium on Physical Design*. ISPD '97. ACM, 1997, 10.1145/267665.267682. http://doi.acm.org/10.1145/267665.267682.

[32] G.-M. Wu, M. Shyu, and Y.-W. Chang. "Universal Switch Blocks for Three-dimensional FPGA Design". *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*. FPGA '99. ACM, 1999, 10.1145/296399.296530. http://doi.acm.org/10.1145/296399.296530.

[33] S. J. E. Wilton. "Architectures and Algorithms for Field-programmable Gate Arrays with Embedded Memory". PhD thesis. University of Toronto, 1997.

[34] M. I. Masud and S. J. E. Wilton. "A New Switch Block for Segmented FPGAs". *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*. FPL '99. Springer-Verlag, 1999, http://dl.acm.org/citation.cfm?id=647926.739234.

[35] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/270.

[36] "OpenFPGA Online Documentation". https://github.com/lnis-uofu/OpenFPGA/blob/dev/openfpga_flow/arch/vpr_only_templates/k6_frac_N10_tileable_adder_chain_mem16K_multi_io_capacity_40nm.xml.

[37] "OpenFPGA Online Documentation". https://github.com/lnis-uofu/OpenFPGA/blob/dev/openfpga_flow/arch/vpr_only_templates/k6_frac_N10_tileable_adder_chain_mem16K_reduced_io_40nm.xml.

[38] "OpenFPGA Online Documentation". https://github.com/lnis-uofu/OpenFPGA/blob/dev/openfpga_flow/arch/vpr_only_templates/k6_frac_N10_tileable_adder_chain_mem16K_aib_40nm.xml.

[39] "OpenFPGA Online Documentation". https://github.com/lnis-uofu/OpenFPGA/blob/dev/openfpga_flow/arch/vpr_only_templates/k6_frac_N10_tileable_adder_chain_wide_mem16K_40nm.xml.

[40] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/277.

[41] "OpenFPGA Online Documentation". https://openfpga.readthedocs.io/en/latest/arch_lang/addon_vpr_syntax.html#models-complex-blocks-and-physical-tiles.

[42] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/291.

[43] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/annotate_vpr_arch/#physical-tile-annotation.

[44] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/manual/file_formats/pin_constraints_file.

[45] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/simulation_setting/#clock-setting.

[46] B. Grady and J. Anderson. "Synthesizable Heterogeneous FPGA Fabrics". *IEEE International Conference on Field-Programmable Technology (IEEE FPT)*. 2018.

[47] D. Lewis, E. Ahmed, D. Cashman, T. Vanderhoek, C. Lane, A. Lee, et al. "Architectural Enhancements in Stratix-III™and Stratix-IV™". *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '09. ACM, 2009, 10.1145/1508128.1508135. http://doi.acm.org/10.1145/1508128.1508135.

[48] C. Chiasson and V. Betz. "Should FPGAS abandon the pass-gate?" *2013 23rd International Conference*

on Field programmable Logic and Applications. 2013, 10.1109/FPL.2013.6645511.

[49] E. Hung. "Mind the (synthesis) gap: Examining where academic FPGA tools lag behind industry". *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, 10.1109/FPL.2015.7294007.

[50] Weiping Shi and Zhuo Li. "A fast algorithm for optimal buffer insertion". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.6 (2005),

[51] "Fork on the OpenLane Project". https://github.com/lnis-uofu/OpenLane.

[52] "Magic VLSI Layout Tool". http://opencircuitdesign.com/magic/.

[53] "SpyDrNet Github Repository". https://github.com/byuccl/spydrnet.

[54] "METIS Github Repository". https://github.com/KarypisLab/METIS.

[55] "FPGA Secured Bitream Github project". https://github.com/lnis-uofu/FPGA_Secured_Bitstream.

[56] "SiliconCompiler Github Repository". https://github.com/siliconcompiler/siliconcompiler.

[57] X. Tang, E. Giacomin, A. Alacchi, B. Chauviere, and P. Gaillardon. "OpenFPGA: An Opensource Framework Enabling Rapid Prototyping of Customizable FPGAs". *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, 10.1109/FPL.2019.00065.

[58] "SOFA Documentation". https://skywater-openfpga.readthedocs.io/en/latest/datasheet/sofa_chd/sofa_chd_circuit_design/#multiplexer.

[59] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/manual/arch_lang/circuit_model_examples/#circuit-model-mux-1level-example.

[60] "Custom cells in Skywater Opensource FPGAs Online Documentation". https://skywater-openfpga.readthedocs.io/en/latest/datasheet/sofa_chd/custom_cells/.

[61] A. Alacchi, E. Giacomin, X. Tang, and P.-E. Gaillardon. "Smart-Redundancy: An Alternative SEU/SET Mitigation Method for FPGAs". *IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, 10.1109/ISCAS51556.2021.9401092.

[62] C. Wolf. "Yosys Open SYnthesis Suite". *http://www.clifford.at/yosys/, https://github.com/YosysHQ/yosys*.

[63] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/manual/openfpga_shell/launch_openfpga_shell/.

[64] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/199.

[65] "OpenFPGA Documentation". https://openfpga.readthedocs.io/en/master/dev_manual/ci_cd_setup/.

[66] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/154.

[67] "OpenFPGA pull request user template". https://github.com/lnis-uofu/OpenFPGA/blob/master/.github/PULL_REQUEST_TEMPLATE.md.

[68] "Icarus Verilog Compiler". *https://github.com/steveicarus/iverilog, http://iverilog.icarus.com/*.

[69] "OpenFPGA Project Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/290.

[70] "OpenFPGA Project Issue". https://github.com/lnis-uofu/OpenFPGA/issues/52.

[71] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/152.

[72] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/156.

[73] "LNIS youtube Channel". https://youtu.be/YTggSZHsTjg.

[74] "OpenFPGA technical highlights". https://openfpga.readthedocs.io/en/master/overview/tech_highlights.

[75] "OpenFPGA Overview Video". https://openfpga.readthedocs.io/en/master/overview/motivation/.

[76] "OpenFPGA Tutorial about how to compile". https://openfpga.readthedocs.io/en/master/tutorials/getting_started/compile/.

[77] "OpenFPGA Tutorial about generating fabric netlists". https://openfpga.readthedocs.io/en/master/tutorials/design_flow/generate_fabric/.

[78] "OpenFPGA Tutorial about user defined template.v". https://openfpga.readthedocs.io/en/master/tutorials/arch_modeling/user_defined_temp_tutorial/.

[79] "OpenFPGA Tutorial about using standard cells". https://openfpga.readthedocs.io/en/master/tutorials/arch_modeling/open_cell_libraries_tutorial/.

[80] "OpenFPGA FAQ". https://openfpga.readthedocs.io/en/master/faq/.

[81] "Youtube video about user defined template.v". https://www.youtube.com/watch?v=YTggSZHsTjgl.

[82] "VexRISCV Github Repository". https://github.com/SpinalHDL/VexRiscv.

[83] "SOFA". https://github.com/lnis-uofu/SOFA.

[84] "OpenRAM Github Repository". https://github.com/VLSIDA/OpenRAM.

[85] "DFFRAM Github Repository". https://github.com/Cloud-V/DFFRAM.

[86] "SOFA Plus Github Repository". https://github.com/lnis-uofu/SOFA-Plus-eFPGA.

[87] "SERV Github Repository - a bit-serial RISC-V core". https://github.com/olofk/serv.

[88] "IBEX RISC-V Core Github Repository". https://github.com/lowRISC/ibex.

[89] "RI5CY Github Repository". https://github.com/openhwgroup/cv32e40p.

[90] "SOFA Github Repository". https://github.com/lnis-uofu/SOFA.

[91] "Caravel SOFA-HD". https://github.com/lnis-uofu/Caravel-SOFA-HD.

[92] "Caravel SOFA-CHD". https://github.com/lnis-uofu/Caravel-SOFA-CHD.

[93] "Caravel QLSOFA-HD". https://github.com/lnis-uofu/Caravel-QLSOFA-HD.

[94] "SOFA Online Documentation". https://skywater-openfpga.readthedocs.io/en/latest/.

[95] "THE EFABLESS OPEN MPW SHUTTLE PROGRAM". https://efabless.com/open_shuttle_program/2.

[96] "LinkedIn Post about Chip Testing". https://www.linkedin.com/posts/pegaillardon_fpgas-osfpga-vlsi-activity-6806815117074944000-d3IN.

[97] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1048.

[98] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1309.

[99] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1355.

[100] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1448.

[101] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1621.

[102] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1661.

[103] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-

to-routing/pull/1742.

[104] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1747.

[105] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1667.

[106] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1693.

[107] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1694.

[108] "VTR Project Issue". https://github.com/verilog-to-routing/vtr-verilog-to-routing/issues/1580.

[109] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1800.

[110] "VTR Project Pull Request". https://github.com/verilog-to-routing/vtr-verilog-to-routing/pull/1801.

[111] P. Gaillardon, X. Tang, G. Kim, and G. De Micheli. "A Novel FPGA Architecture Based on Ultrafine Grain Reconfigurable Logic Cells". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.10 (2015),

[112] X. Tang, P. Gaillardon, and G. De Micheli. "Pattern-based FPGA logic block and clustering algorithm". *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014,

[113] "Caravel Harness". https://github.com/efabless/caravel.

[114] "Skywater ReRAM Project Pull Request". https://github.com/google/skywater-pdk-libs-sky130_fd_pr_reram.

[115] "Skywater ReRAM Project Pull Request". https://github.com/google/skywater-pdk-libs-sky130_fd_pr_reram/pull/26.

[116] "FASM Online Documentation". https://fasm.readthedocs.io/en/latest/#example-canonicalization.

[117] "OpenFPGA Pull Request". https://github.com/lnis-uofu/OpenFPGA/pull/241.

[118] "GDS-ready SOFA IP macros". https://foss-eda-tools.googlesource.com/openfpga-macros.

[119] "FuseSoC Github Repository". https://github.com/olofk/fusesoc.

[120] "EDAlize Github Pull Request". https://github.com/olofk/edalize/pull/300.

[121] "OpenLane Project". https://github.com/The-OpenROAD-Project/OpenLane.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	DESCRIPTION
ASIC	Application Specific Integrated Circuits
ATPG	Automatic Test Pattern Generation
BL/WL	Bit Line/Word Line
BLE	Basic Logic Element
CB	Connection Block
CCFF	Configuration Chain Flip-Flop
CCFF	Configuration Chain Flip- Flops
CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Development
CLB	Configurable Logic Blocks
CRAM	Configuration Random Access Memory
CTS	Clock Tree Synthesis
DFS	Depth-First Search
DMR	Dual Modular Redundancy
DSE	Design Space Exploration
FF	Flip-Flops
FLE	Functional Logic Element
FPGA	Field Programmable Gate Array
FSM	Finite State Machines
HDL	Hardware Description Language
I/O	Input/Output
IMECCC	In Memory Error Correction Code Checking
INV	Inverter
IP	Intellectual Property
LEs	Logic Elements
LUT	Look-Up Table
MCNC	Microelectronics Center of North Carolina
MUX	multiplexers
PDK	Process Design Kit
PMU	Programming Management Unit
QoR	Quality-of-Results
RR	Routing Resource
SAN	Simulated Annealing
SB	Switch Block
SCFF	Scan Chain Flip-Flop
SDC	Synopsys Design Constraints

SETs	Single-Event Transients
SEUs	Single-Event Upsets
SPSA	Simultaneous Perturbation Stochastic Approximation
SRAM	Static Random-Access Memory
STA	Static Timing Analysis
TAP	Test Access Point
TCL	A string-based scripting language
TMR	Triple Modular Redundancy
TSMC	Taiwan Semiconductor Manufacturing Company
VDHL	VHSIC Hardware Description Language
VPR	Versatile Placement and Routing
XML	Extensible Markup Language