TACO: A Tiled Architecture of (re)Configurable Operators

Jason Yan (zexuany2), Nitish Bhupathi Raju (nbhup2),
Vinit Gupta (vinitg2), Feiyang Liu (fl22), Rudra Thakkar (rudrapt2)

ECE 427 Advanced VLSI System Design
Fall 2025 Final Report

***Abstract*—In this project, our group designed an FPGA with a custom architecture in the TSMC 65nm process. Our FPGA fabric features Configurable Logic Blocks (CLB) that form the core of the reconfigurable logic array. The routing network consists of Switch Boxes (SB) with 8-wide routing tracks, enabling programmable interconnections between CLBs, and Connection Boxes (CB) that interface each CLB to the surrounding routing fabric. At the center of the FPGA fabric, we integrate dedicated Multiply and Accumulate (MAC) units to support arithmetic-intensive applications. Surrounding the core fabric, the FPGA includes 1kb on-chip BRAM for data storage, GPIO pins for external interfacing, a clock divider that dynamically adjusts the operating frequency based on the supported frequency (F_{max}) of the mapped design, and a JTAG interface used for programming and debugging by shifting in bitstreams. To support the full design flow, we utilize the open-source Verilog-to-Routing (VTR) framework to compile Verilog designs into bitstreams compatible with our FPGA architecture.**

I. Project Overview

The general goal of this project is to design, verify, and tape out a custom-designed FPGA in the TSMC 65nm process with custom compiler support. By the end of the project, we implemented:

1) 288 configurable logic blocks (CLB) with a single 4-input lookup table, a flip-flop, and a mux
2) 8-wide Interconnection between CLBs through connection blocks and switch blocks
3) JTAG hardware for programming and debugging
4) 39 GPIO pins for I/O
5) A Verilog-to-bitstream compiler based on open-source projects
6) 64 selectable clock frequencies through clock dividers
7) 1 KB BRAM (Memory)
8) 9 Multiply and Accumulate Units (MACs)

II. Main Features and Functionality

Our FPGA can run RTL designs (that fit within our resource budget) at a programmable frequency (up to the F_{max} of the design). To aid in versatility, the I/O pins are GPIOs that can be programmed through an independent bitstream, which allows reconfigurability based on the RTL design's needs. Furthermore, on-chip BRAM and on-fabric MAC units are available to accelerate data storage and arithmetic computations, respectively. We also created an end-to-end Verilog-to-bitstream compiler for our FPGA architecture to facilitate ease of use.

III. High Level Architecture

The High-level block diagram of our FPGA is shown in Figure 1. The figure includes a 2×3 fabric for simplicity. Note: Within the FPGA fabric, black lines indicate the interconnections, while the red lines indicate the scan-chain for JTAG.

IV. Physical Design

Figure 2 shows the physical view of our TACO. We also decided to add some easter eggs to our chip (on the AP layer). Figure 3 shows this view. Our chip has the following properties:

1) FPGA Clock Frequency: 20 MHz
2) JTAG Frequency: 10 MHz
3) Area Utilization: 85.756% for 708x708 μm^2 core site
4) Power Estimation: 7.214 mW
5) Input/Output Pins:
   a) 2 VDD Pins
   b) 2 VSS pins
   c) 8 pins that are always input
   d) 1 pin that is always output
   e) 39 GPIO pins that can be programmed as input or output

A. Design Choices and Justification

One major design choice was the size of the CLB array. We created a parameterizable design and explored various values until we found the maximum size that fit within the area constraints, which turned out to be 18×17. Additionally, we wanted to heavily prioritize reconfigurability in every part of the chip. This motivated the choice to add GPIO pins and a programmable clock divider. Furthermore, we decided to add support for JTAG to facilitate ease of use in real-world scenarios.

We also prioritized adding two advanced features to our chip: BRAM and MAC units. Most modern FPGAs contain on-chip memory; therefore, we decided to add some memory support to facilitate practical RTL designs that require data

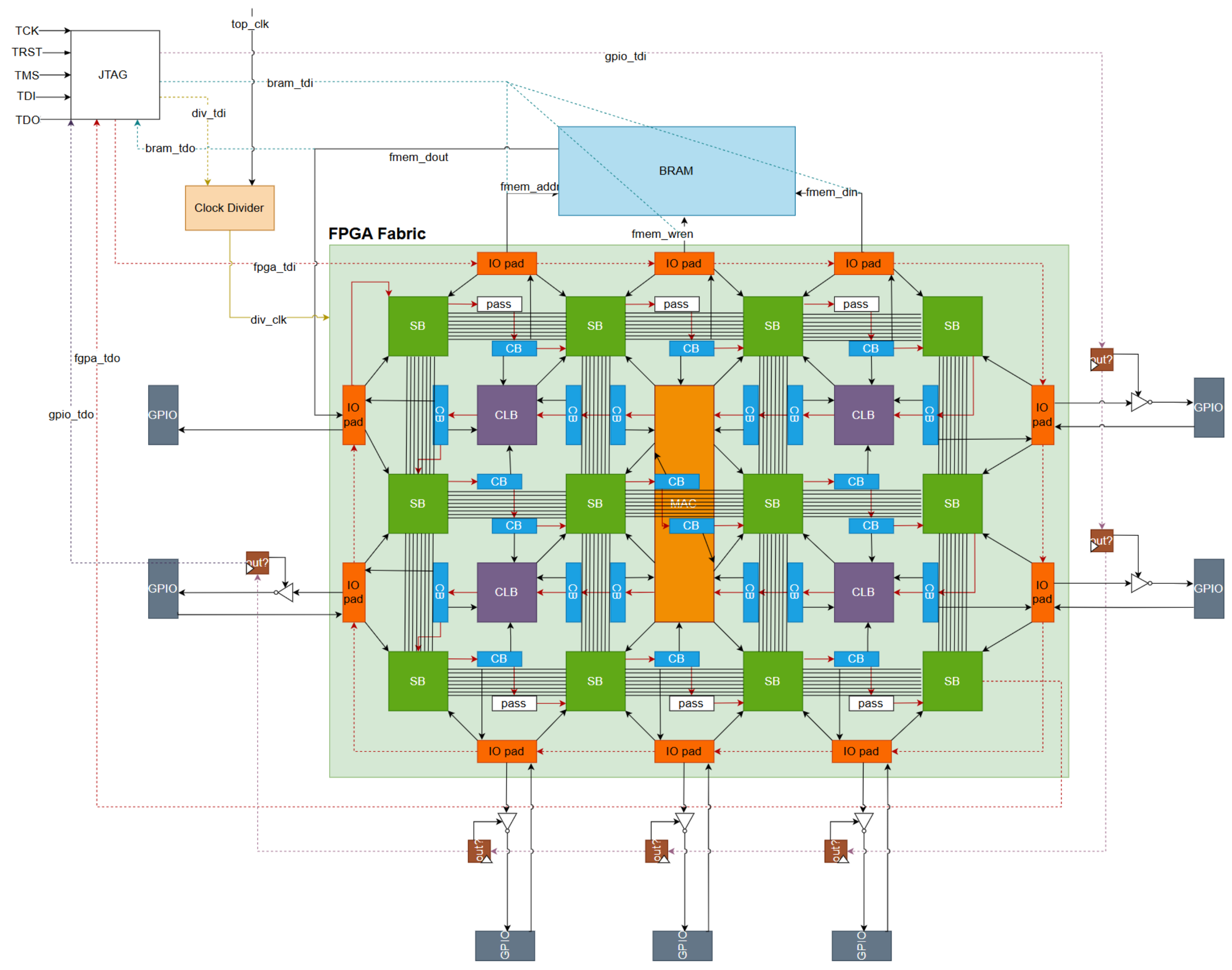

Fig. 1. High Level Design Diagram

storage. Additionally, we integrated MAC units into the design to create a novel FPGA architecture capable of accelerating multiply–add computations. These units also serve as a proof of concept for incorporating arbitrary hard IPs into the fabric with compiler support.

B. Comparison with Proposal

We added most features that we were planning to implement from the project proposal (These features are described in the Project Overview section). However, the following advanced features were proposed, but they were not implemented due to prioritization of other advanced features: Carry-chain for the CLBs, Clustered CLBs, and On-chip RISC-V Core.

C. Verification and Test Strategy

*1) **Verification Flow**:* The following plan outlines the features that were verified, testing methodologies, and the prioritization of verification phases. This plan was adopted for RTL and Post-PnR Verification.

*a) **Tile-Level Verification**:* Since the FPGA architecture consists of repeated tiles, verification begins at the tile level. Each tile includes a Configurable Logic Block (CLB) and routing units (connection and switch blocks). Simple synthesized standard cells (up to four inputs and one output) are tested on a small 2×2 FPGA to validate CLB functionality and routing correctness.

*b) **Functional Block-Level Verification**:* After tile-level testing, the FPGA is scaled to a 10×10 configuration for larger designs. Connectivity blocks are verified using constraint-random testing, with the intended RTL serving as the golden model. Additional components, such as the JTAG TAP controller, clock divider, and GPIOs, are verified individually before integrated into the FPGA fabric.

*c) **Full-Chip Verification**:* Following block-level validation, the entire chip is tested as a single unit by toggling JTAG control pins, loading configuration bitstreams, and applying randomized GPIO inputs while monitoring outputs. Advanced features, such as the in-fabric MAC and memory units, are

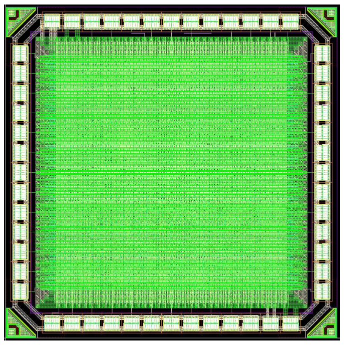
Fig. 2. Physical View

Fig. 3. AP Layer View

also verified.

*d) **Functional/Toggle Coverage**:* Each CLB is tested in both combinational and sequential modes. Larger designs, including ALUs and a simple processor, are synthesized and verified on the FPGA. Toggle coverage for RTL simulation is approximately 80% at 14x15 fabric, limited by dead code in the open-source JTAG TAP controller, dummy logic from the tiled design, and unused IP functions that do not impact overall behavior.

*e) **Automated Regression Tool**:* A Python-based regression framework automates test generation, execution, and error reporting. Inputs and timing variations are randomized to enhance coverage and verification robustness.

*f) **Post-PnR Verification**:* We modified the original testbench to drive the stimulus at the negative edge of the clock to ensure the set input delay is properly respected. We also utilized a similar testbench as the RTL simulation, but with path delays annotated by the SDF files generated by Innovus. The regression tool is applied to check the correctness across all test cases.

*2) **DFT Features**:* Numerous DFT features were added to the design, and they are described in-detail below.

*a) **Scan Chain**:* One benefit of our scan chain approach to programming is that we can easily use this feature for DFT. By shifting out the bitstream, we can retrieve the internal state of most flip-flops in the FPGA fabric, which allows us to verify the functionality of the design.

*b) **BRAM**:* One important feature of our design that is not included in the scan chain is BRAM. To enable testing, we decided to add a DFT feature where the contents of the BRAM can be written to and read from using JTAG.

*c) **Backup Shift Pins**:* To program any part of the chip, our initial plan was to use JTAG. However, this creates a single point of failure; if JTAG does not work as intended, we cannot control any aspect of the chip. To circumvent this, we dedicated two I/O pins to serve as a backup (Master Shift En and Master Shift In). We created a simple protocol in the chip where, once certain instructions are sent through these pins, all components of the chip can be programmed through them instead of JTAG.

V. Post-Silicon Validation Plan

We plan to design a PCB to bring up and validate our chip. The chip will be programmed using a standard FPGA development board (such as Digilent Genesys 2). For initial testing, we plan to drive JTAG pins directly from the FPGA by following the TAP controller state machine developed for the testbench framework. For more robust testing, we plan to program the chip using OpenOCD as a driver for JTAG through a JTAG HS3 cable.

FPGA dev-board programming: We plan to use the existing testbench framework (developed in SystemVerilog) to be synthesized on the FPGA as TACO's control unit.

Timeline:

- 01/15 - 02/15: PCB design, FPGA dev board setup + testing
- 02/15 - 02/28 (chip arrival): Power + reset testing, scan chain shift-in/out
- 03/01 - 03/21: Standard cell bitstream test
- 03/21 - 04/15: Advanced bitstream test (FSM + CPU)
- 04/15 - 05/15: OpenOCD bring-up, Debug + tests

VI. Individual Contributions

A. Jason

- Full Testbench Verification Setup (RTL, Post-Synth, Post-PnR) + Python Regression Setup
- PnR Scripts for Innovus
- Virtuoso Integration (DRC + LVS)

B. Rudra

- Compiler Flow + Bitstream Generation
- Synth + PnR Scripts for Innovus
- Virtuoso Integration (DRC + LVS)

C. Nitish

- RTL Design + Preliminary Verification
- Core Architecture + Advanced Features (MAC Integration, Clock Divider, GPIO)
- DRC Fixes (I/O Pad)

D. Vinit

- RTL Design + Preliminary Verification
- Core Architecture + Advanced Features (BRAM Integration, CSRs, GPIO)
- Debugging

E. Adam

- Compiler Tool Chain
- Synth Scripts + Post Synth Verif + Pnr Scripts
- Virtuoso Integration (DRC + LVS)

VII. Major Challenges and Lessons Learned

One major challenge we encountered was that one of the larger benchmark designs mapped to our FPGA was failing when compared to the golden model. We spent considerable time debugging this issue before identifying that the VTR toolchain was not correctly processing OR gate logic. This required us to implement additional processing steps to handle OR gates properly within the VTR flow. Resolving this bug reinforced the importance of thorough, bottom-up verification and motivated our decision to develop a validation flow in which all standard cells are first verified on a 2×2 FPGA.

Another major challenge we encountered was a comb-loop bug that appeared during Post-PnR testbench simulations. Our initial assumption was that once the bitstream was generated, there should be no active combinational loops within the FPGA fabric, as the VTR toolchain guarantees this. However, we discovered that an unused CLB input was being driven by its own output. This occurred because VTR assumes unused CLB inputs are undriven, whereas in our fabric, we default our CBs to track 0, which happened to be the same track driven by the CLB output. The bug only surfaced with SDF annotations, since under normal conditions, an unused input should not affect the CLB output. To address this issue, we modified our RTL design and compiler toolchain so that all unused CLB inputs are set to 0. This experience reinforced the need to be cautious when making simplifying design assumptions and highlighted the importance of writing stable logic.

VIII. Design Documentation

A. CLB Architecture

Our baseline CLB implementation contains one LUT4, flip-flop (FF), and a mux to select between the combinational output from the LUT4 and the sequential output from the FF to determine if the CLB is modeling combinational or sequential logic. The CLB Architecture is shown in Fig. 4.

B. Connection Box (CB)

Each connection box is responsible for connecting the input of the CLB to the global routing network. Since each CLB has 4 inputs, each CLB has 4 CBs that surround it. Each CB chooses one of the eight tracks as the input to the CLB. When a CLB input is left unused, it is set to 0.

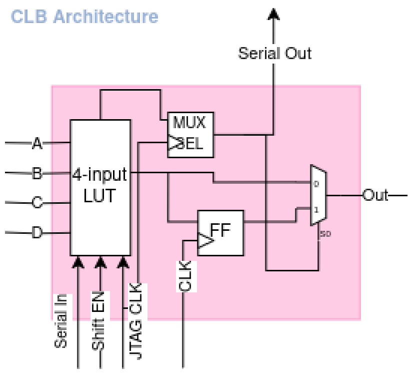

Fig. 4. CLB Architecture

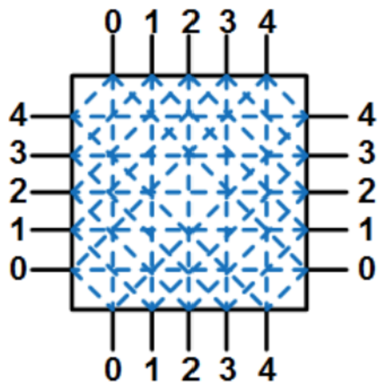

Fig. 5. Universal Switch Box

C. Switch Box (SB)

Each switch box is responsible for connecting tracks to create the desired routing. We used a unidirectional switch box structure that enables connections to three different tracks (up, down, and continue in the same direction). We implemented the universal switch box architecture as shown in 5.

Certain SBs have ”extra inputs” coming into the Switch Box to connect IO pad inputs, CLB outputs, and MAC outputs to the routing network. These extra inputs are fed into the fourth input of a 4:1 mux (which would otherwise be left unconnected).

D. Track Width

The number of tracks required per channel depends on the level of complexity. According to multiple studies, with around 200 single-LUT4 CLBs, the most complex design will require at least 6-7 tracks per channel. We received some suggestions to use 8 as our width to ensure that it will not be the limiting factor when mapping designs. Therefore, we elected to use 8 as our width.

E. BRAM

We implemented an off-fabric 1kb BRAM that the FPGA Fabric can communicate with using designated IO pads. The BRAM has the following ports: 8-bit address, 4-bit din, 1-bit wen, and 4-bit dout. Currently, we are able to switch between

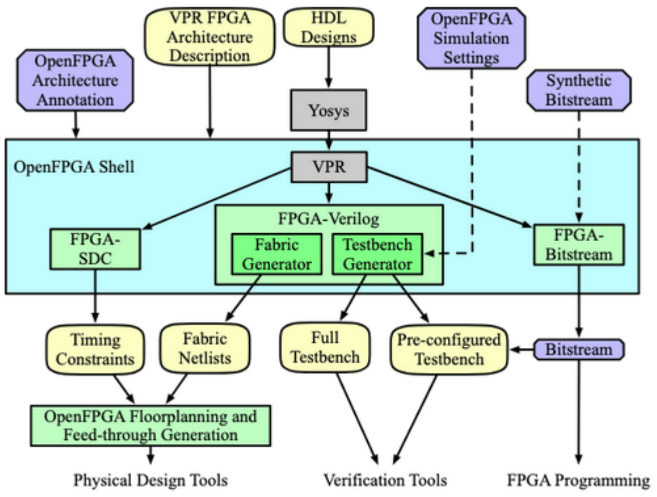

Fig. 6. Example Bitstream Generation Workflow

using SRAM and a register file for the BRAM. We will use a register file if SRAM proves to be too complicated.

F. MAC

There is a column of MACs at the center of the FPGA fabric. Each MAC has the following inputs: 3-bit a, 3-bit b, 1-bit mac_enable, 1-bit mac_clear. Each MAC has an 8-bit output. If the RTL logic mapped onto our fabric needs to use MAC, these signals need to be specifically named and used in the RTL.

G. Clock

For the clock, we will use an external oscillator. The clock divider is a standard divide-by-even divider. The output of this module is the clock provided to the FPGA fabric.

H. Verilog-to-bitstream Generation

Our bitstream generation flow involves open-source tools, including Yosys and Versatile Place and Route (VPR). Yosys is a synthesis tool that takes Verilog designs and generates a netlist. VPR takes in the netlist, our FPGA architectural description file in XML, and some constraints and generates packed netlists and the corresponding placement and routing on the FPGA. The generated files from VTR include several intermediate files, such as .net, .place, .route, .blif, and .fasm. These files are subsequently processed through custom Python scripts to generate the final bitstream for our FPGA. An example workflow from the OpenFPGA project is shown in Fig. 6.

I. JTAG

The primary interface for programming the FPGA's configuration bitstream and for low-level debugging is the industry-standard IEEE 1149.1 JTAG. This serial interface minimizes pin count requirements for our design.

We made trivial modifications to an open-source JTAG implementation to create shift_enable signals for different instructions. We added four new instructions: FPGA_SHIFT_IR, CLOCK_DIV_IR, GPIO_IR, and MEM_JTAG_IR (which allows the JTAG controller to read from/write into BRAM). The first three instructions enable the shifting logic for the corresponding module. This way, each module of the chip is independently programmable.

J. User Data Interface

For general-purpose user interaction, the FPGA utilizes parallel General Purpose Input/Output (GPIO) pins. These pins can be configured by the loaded bitstream as either inputs or outputs, allowing the FPGA logic to directly interact with external components. The tri-state buffer logic to support GPIO is shown in 1.

IX. Testing Instruction

Ensure that Synopsys 2024 is being used and unload module calibre (it causes Python path conflicts for the script). To run a specific test case for RTL simulation, navigate to "/groups/ece427-group7/pnr_output/the_final_taco" and use the "run_rand_test.py" script and specify its file path as an argument. For example, to execute the testcase "Processor.v", you can run "./run_rand_test.py ../vtr/hdl/10x10_testcases/Processor.v" in the terminal. To execute all test cases for the larger design, run "./run_rand_test.py ../vtr/hdl/10x10_testcases/". This process may take over 30 minutes.

To run post-PnR simulation, add " –tb_path hvl/top_post_pnr_tb.sv" at the end of the command line. The process can take up to hours for all testcases.

The log files will be automatically saved under "/groups/ece427-group7/taco/rtl/log", and a summary of the test results will be printed to the terminal upon completion.

References

[1] Xilinx Inc., "7 Series Architecture Overview," ARM SoC Series. [Online]. Available: https://www.southampton.ac.uk/~bim/notes/cad/reference/ARMSoC/P4/7_Series_Architecture_Overview.pdf

[2] Xilinx Inc., "7 Series Configurable Logic Block User Guide," UG474 (v1.11). [Online]. Available: https://www.eng.auburn.edu/~nelson/courses/elec4200/FPGA/ug474_7Series_CLB.pdf

[3] V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs". Springer, 1999. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4615-5145-4

[4] K. Compton and S. Hauck, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Springer, 2010. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4615-5145-4

[5] University of Illinois Urbana-Champaign, "ECE 498HK: GRAPE." [Online]. Available: https://courses.grainger.illinois.edu/ECE498HK/fa2024/projects.html

[6] TinyFPGA, "TinyFPGA." [Online]. Available: https://tinyfpga.com/

[7] L. Braman *et al.*, “OpenFPGA,” GitHub repository. [Online]. Available: https://github.com/lnis-uofu/OpenFPGA?tab=readme-ov-file

[8] The FABULOUS Project, “FABULOUS FPGA Toolflow,” Read the Docs. [Online]. Available: https://fabulous.readthedocs.io/en/latest/

[9] AMD, “DS099: 7 Series FPGAs Data Sheet: Overview,” Rev. 1.0. [Online]. Available: https://docs.amd.com/v/u/en-US/ds099

[10] Xilinx Inc., “XC4036XL-1HQ160I Data Sheet.” [Online]. Available: https://www.xilinxsemi.com/datasheet/xilinxsemi/XC4036XL-1HQ160I.pdf

[11] YosysHQ, “Yosys Open Synthesis Suite,” GitHub repository. [Online]. Available: https://github.com/YosysHQ/yosys

[12] YosysHQ, “nextpnr: Next-Generation Place and Route,” GitHub repository. [Online]. Available: https://github.com/YosysHQ/nextpnr

[13] C. Wolf, “Project IceStorm: Open source flow for Lattice iCE40 FPGAs,” Read the Docs. [Online]. Available: https://prjicestorm.readthedocs.io/en/latest/overview.html#what-is-project-icestorm

[14] OpenFPGA Project, “Architecture Description Language: Configuration Protocol,” Read the Docs. [Online]. Available: https://openfpga.readthedocs.io/en/master/manual/arch_lang/config_protocol/#configuration-protocol

[15] R. Sivaswamy, G. Konjevod, and S. Hauck, “On the Feasibility of FPGA-Based Networks,” Proc. FPGA Netw. Workshop, Univ. Washington, 2007. [Online]. Available: https://courses.cs.washington.edu/courses/cse591n/07au/papers/p21-sivaswamy.pdf

[16] OpenOCD Project, “OpenOCD: On-Chip Debug, In-Circuit Emulator, Flash Programmer.” [Online]. Available: https://openocd.org/

[17] G. Yu, T. Cheng, B. Kettlewell, H. Liew, M. Seok, & P. Kinget (2017). ”FPGA with Improved Routability and Robustness in 130nm CMOS with Open-Source CAD Targetability.” https://arxiv.org/abs/1712.03411