PRGA: An Open-Source FPGA Research and Prototyping Framework

Ang Li
angl(at)princeton(dot)edu
Princeton University
Princeton, New Jersey

David Wentzlaff
wentzlaf(at)princeton(dot)edu
Princeton University
Princeton, New Jersey

ABSTRACT

Field Programmable Gate Arrays (FPGA) are being used in a fast-growing range of scenarios, and heterogeneous CPU-FPGA systems are being tapped as a possible way to mitigate the challenges posed by the end of Moore's Law. This growth in diverse use cases has fueled the need to customize FPGA architectures for particular applications or application domains. While high-level FPGA models can help explore the FPGA architecture space, as FPGAs move to more advanced design nodes, there is an increased need for low-level FPGA research and prototyping platforms that can be brought all the way to fabrication.

This paper presents **Princeton Reconfigurable Gate Array** (PRGA), a highly customizable, scalable, and complete open-source framework for building custom FPGAs. The framework's core functions include generating synthesizable Verilog from user-specified FPGA architectures, and providing a complete, auto-generated, open-source CAD toolchain for the custom FPGAs. Developed in Python, PRGA provides a user-friendly API and supports use both as a standalone FPGA as well as an embedded FPGA. PRGA is a great platform for FPGA architecture research, FPGA configuration memory research, FPGA CAD tool research, and heterogeneous systems research. It is also a completely open-source framework for designers who need a free and customizable FPGA IP core. An FPGA designed with PRGA is placed and routed using standard cell libraries. The design is evaluated and compared to prior works, providing comparable performance and increased configurability.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**.

KEYWORDS

FPGA; FPGA architecture; open-source hardware

ACM Reference Format:
Ang Li and David Wentzlaff. 2021. PRGA: An Open-Source FPGA Research and Prototyping Framework. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431920.3439294

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have become an increasingly important tool to enable application performance in a post Moore's Law [19] world. Whether they are being used as a standalone compute fabric or a supplement to processors at the chip-level [8, 10, 29], board-level [20], system-level, or datacenter-level [1, 5], the diversity of use cases and importance of FPGAs have been increasing. Ideally, an FPGA architecture should be optimized for each unique use case. In practice, though, it is very challenging to evaluate different FPGA designs in detail and even more challenging and time-consuming to prototype and bring those FPGAs to fabrication. This is because FPGA chip design flow has diverged from the design flows of other digital ASICs like processors. Commercial FPGAs are often designed with custom cells and specialized EDA tools that are publicly unavailable. Likewise, each unique FPGA requires the creation of customized CAD tools. Due to this high design cost, commercial FPGA vendors typically offer a limited set of designs optimized across common, but potentially non-characteristic, use cases. Due to similar reasons, FPGA architecture studies often use and stop at high-level models [4, 21].

To facilitate FPGA architecture research and enable designs optimized for custom applications, tools are needed to evaluate, optimize, and prototype FPGA architectures all the way down to the fabrication level. An ideal framework would be easy-to-use, extensible, scalable, and open-source. A framework that provides synthesizable RTL enables gate-level or transistor-level implementation using commercial ASIC design flows and standard cell libraries. By enabling such physical prototyping, a framework can be used to evaluate timing, power, and area with the utmost fidelity. Likewise, RTL-level prototyping incorporates the details of the configuration memory, enabling research on bitstream format and partial or dynamic reconfiguration. High-level modeling tools are an important first step, but there exists a need for low-level (RTL and below) frameworks that can be used to study low-level issues such as floorplanning, design regularity, signal integrity, and other physical design issues all while providing the path to then take the optimized design through prototyping and fabrication.

In this paper, we present Princeton Reconfigurable Gate Array (PRGA), a highly customizable, scalable, and complete open-source framework for building custom FPGAs. PRGA is available at `https://parallel.princeton.edu/prga`. Fig. 1 shows the workflow used to design a custom FPGA and then develop an application that uses it. The PRGA FPGA architecture is highly customizable, and it supports user-provided modules such as SRAM macros, hard arithmetic units, and routing switches, all of which can be easily added into the flow. PRGA is developed in Python and provides a well-defined Python API. Extensions are encouraged and

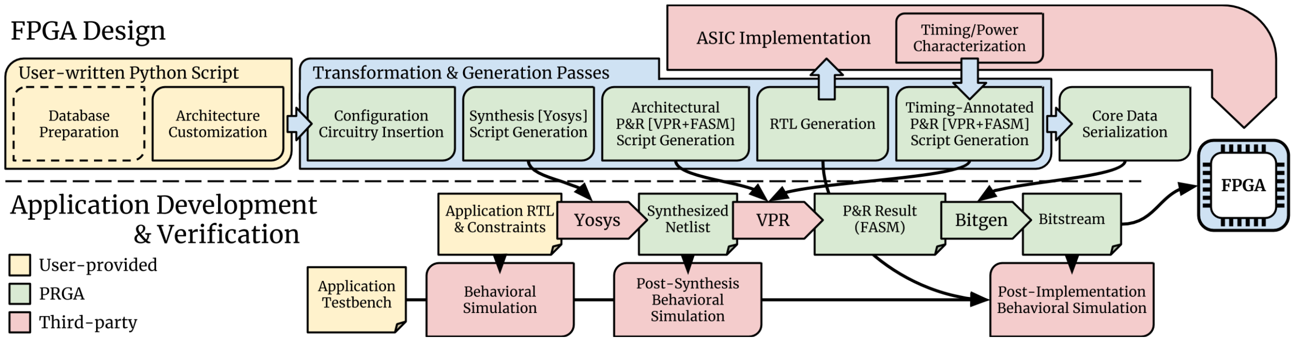

FPGA Design
User-written Python Script
Database Preparation
Architecture Customization
Transformation & Generation Passes
Configuration Circuitry Insertion
Synthesis [Yosys] Script Generation
Architectural P&R [VPR+FASM] Script Generation
RTL Generation
Timing-Annotated P&R [VPR+FASM] Script Generation
Core Data Serialization
ASIC Implementation
Timing/Power Characterization
FPGA
Application Development & Verification
Application RTL & Constraints
Yosys
Synthesized Netlist
VPR
P&R Result (FASM)
Bitgen
Bitstream
Application Testbench
Behavioral Simulation
Post-Synthesis Behavioral Simulation
Post-Implementation Behavioral Simulation
User-provided
PRGA
Third-party

Figure 1: Overview of a typical PRGA workflow

supported through modularization and low-level APIs. To further lower the barrier of extending the framework, most output files are generated from human-readable Jinja [25] templates that are customizable without changing the Python codebase. At the end of the FPGA design flow, PRGA produces human-readable, industry-standard, Verilog files that are synthesizable and physically implementable using commercial EDA Tools. PRGA is ASIC-friendly and can be used to generate standalone FPGAs as well as embedded FPGAs, where the customization that PRGA provides is critical.

PRGA is not derived from prior FPGA modeling/exploration tools. Therefore, it is not restricted by the internal representations of legacy tools. This enables PRGA to support flexible hierarchies that match physical implementation needs. Likewise, the generated configuration circuitry is highly flexible and is decoupled from the design hierarchy, opening up the ability for researchers to explore novel configuration strategies (order, storage, and topology) which is a key component to building efficient FPGAs. Such physical-aware customizability is critical to the design of large-scale, high-performance, fabrication-ready FPGAs.

In addition to designing the FPGA itself, PRGA offers a complete HDL-to-bitstream solution using open-source CAD tools, configuring and parameterizing those FPGA implementation CAD tools for the created custom FPGA. Specifically, this flow uses Yosys [28] for technology mapping and synthesis, VPR [21] for place & route, FASM [23] for raw bitstream generation, and a custom bitstream generator to convert the raw bitstream into binary format. The target application can be verified by simulation with various levels of abstraction throughout the flow, making it easy to debug both the FPGA itself and the application. Scripts and data files for the same FPGA are reusable across application development runs.

In summary, the key features of PRGA include:

- *Architecture Customizability*
  1. Fully-customizable, heterogeneous logic blocks: LUT count, LUT size, local interconnect, hard adder chains, multi-modal primitives, logic elements, and more.
  2. Bring-Your-Own-IP: block RAM, hardened multiplier/accumulator, and even big IP cores like CPUs, memory/network controllers, etc.
  3. Fully-customizable routing structure: switch box pattern, connection box pattern, non-uniform channel, long wires, and global wires.
  4. Extensible configuration circuitry: simple scanchain-based configuration or complex, NoC-based, packetized bitstreams with support for partial reconfiguration. Custom configuration circuitry can also be designed using low-level API.
- *CAD Support*
  1. Auto-generated Yosys script for synthesis: BRAM inference, hard logic techmap, and post-synthesis simulation.
  2. Auto-generated, FASM-annotated VPR inputs for placement, routing and raw bitstream generation.
- *ASIC Compatibility*
  1. Bring-Your-Own-Circuits: replace generated modules with custom Verilog modules or hard macros.
  2. ASIC-friendly module hierarchy: fracturable switch box to maximize regularity; arbitrary levels of sub-arrays to balance ASIC QoR and ease-of-backend.
- *Framework Extensibility*
  1. Modularized, pass-based workflow. Passes may be added or modified without affecting the rest of the flow.
  2. Core data structure can be serialized to disk. Tools don't need to rerun the entire building process every time.

In this paper, we evaluate PRGA by characterizing its scalability in terms of memory usage and runtime and find that it enables the creation of very large designs with reasonable computational resources. In addition, we take a design through place and route to tape-in quality to show that PRGA is production-ready. Finally, we compare designs created with PRGA with prior and commercial designs in terms of area and delay, and show that we are competitive with other standard-cell-based FPGA generators.

PRGA enables many exciting applications. It is a great platform for FPGA architecture research, in particular bridging the gap from high-level FPGA architecture exploration tools down to low-level implementation details, as well as enabling RTL-in-the-loop FPGA architecture optimization studies. It can also be used to build targets for FPGA CAD tool research, for example, security-aware place-and-route tools. PRGA is a framework that allows the creation and exploration of many different FPGA designs, which makes it more than an FPGA generator that can only generate a certain type of FPGA. The FPGAs built with PRGA can be used either as standalone FPGAs or integrated into SoCs. It can even be an excellent platform for CPU-FPGA heterogeneous system research.

2 PRGA WORKFLOW

Fig. 1 shows an overview of a typical PRGA workflow. The FPGA design flow is driven by a user-written Python script, while the HDL-to-bitstream flow integrates open-source CAD tools to generate valid bitstreams for the created custom FPGA.

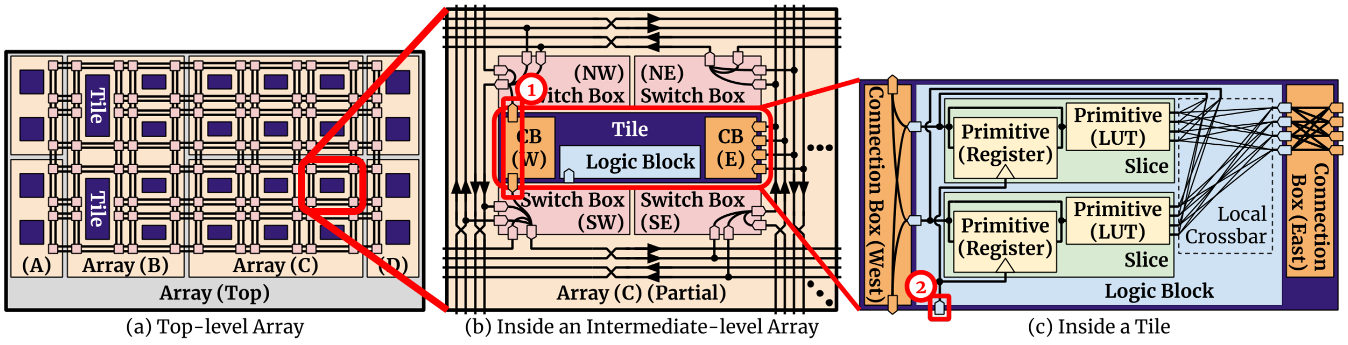


Figure 2: FPGA architecture modeled by PRGA. The position, shape and size of the modules do not reflect the physical properties in an ASIC implementation. Programmable connections are shown as many-to-one connections in routing boxes and blocks. ① are bridging nets discussed in Sec. 3.3.3; ② is an unroutable clock pin directly connected to the global clock tree.

2.1 Database Preparation

PRGA offers various basic components for building an FPGA, including look-up tables (LUT) and registers. However, it is common that FPGA designers want to add custom components into the design, for example, SRAM macros, hard arithmetic units, or even complex IP cores such as memory controllers. These components are classified as **`Logic Primitive`**s in PRGA and elaborated in Sec. 3.3.1. During the database preparation step, users can add custom **`Logic Primitives`** by creating proper models for them, whether they are hard macros or synthesizable HDL designs.

2.2 Architecture Customization

After database preparation, the users can build the desired custom FPGA architecture by creating programmable blocks and assembling them in a hierarchical manner, as shown in Fig. 2. In summary, the top level of the hierarchy is a 2-dimensional **`Array`** composed of **`Tile`**s, **`Switch Box`**es, and nested **`Array`**s. Each **`Tile`** contains one **`Logic Block`** or multiple **`IO Block`**s, in addition to various numbers of **`Connection Box`**es. **`Logic Block`**s and **`IO Block`**s consist of zero to many **`Slice`**s and **`Logic Primitive`**s. Last but not least, **`Slice`**s are composed of nested **`Slice`**s and **`Logic Primitive`**s. Each of these modules will be discussed in detail in Sec. 3.

One key feature of PRGA is the decoupling of the functional abstraction of the FPGA, the underlying configuration circuitry, and the physical implementation of the circuits. During architecture customization, FPGA designers can focus on describing the logical function and connectivity, and leave the implementation details to later steps that are independently customizable. For example, to create a configurable connection in a **`Logic Block`**, users do not need to explicitly specify how the MUX tree should be constructed, or the order of the configuration bits for the MUXes.

The high customizability of PRGA opens up a huge design space for exploration, which is critical to architecture researchers and experienced FPGA designers with specific needs. On the other hand, to make this highly flexible framework easy to use for new users, PRGA provides abundant built-in algorithms to ease the generation of decent FPGAs from meta parameters. For example, while routing boxes are customizable on a per-wire, per-connection basis, various default algorithms are provided for populating them, including Fc-based **`Connection Box`** patterns [4], Universal [6], Wilton [27], and Cycle-Free [15] **`Switch Box`** patterns, etc.

2.3 Transformation and Generation Passes

PRGA does all the heavy-lifting work through transformation and generation passes. The passes shown in Fig. 1 are the most commonly used ones:

Configuration Circuitry Insertion expands the functional, abstract description of the FPGA by elaborating the configuration circuitry and implementing the configurable connections with programmable switches. The configuration circuitry does not need to match the logical design hierarchy. Typically, each type of configuration circuitry has its own implementation of this pass.

Synthesis Script Generation discovers all the logical resources available in the FPGA, then generates technology mapping and synthesis scripts for mapping applications onto the custom FPGA.

Place and Route Script Generation generates the XML files needed by VPR [21] to place and route for the FPGA. To accurately model the highly customizable routing resources, PRGA generates the *Routing Resource Graph* XML in addition to the *Architecture Description* XML. PRGA uses FASM [23] for raw bitstream generation, so additional annotations are added to the XML files accordingly.

RTL Generation generates industry-standard, human-readable, synthesizable Verilog files for the FPGA. As mentioned in Sec. 1, PRGA uses Jinja [25] for text file generation, and the templates can be changed without affecting the rest of PRGA.

Passes may be bound to specific execution order constraints, depend on other passes, or conflict with each other. For example, *RTL Generation* depends on *Configuration Circuitry Insertion* to fill in the physical implementations of the abstract architecture specifications, while *Configuration Circuitry Insertion* passes for different types of configuration circuitry conflict with each other. Graph analysis algorithms are applied to determine the correct execution order of passes, making it easier to add or modify passes. In addition to these built-in passes, users are encouraged and well-supported to add their own passes to optimize, analyze and create custom reports for their custom FPGAs.

2.4 ASIC Implementation

PRGA is designed and optimized with a strong emphasis on enabling gate-/transistor-level prototyping or even fabrication, especially as a target of modern ASIC design flow using commercial EDA tools and standard cell libraries. Common ASIC implementation techniques are taken into consideration during RTL generation.

For example, the configuration enable signal is registered at different levels across the hierarchy to reduce potential skew of this high-fanout net. In addition, the flexible hierarchy and customizable RTL generation grant FPGA designers the freedom to explore and optimize the layout strategy specific to their process and design.

2.5 Application Development and Verification

All the files required by the open-source HDL-to-bitstream toolchain are generated by the generation passes in the FPGA design flow, and they can be reused across different runs of the application development and verification flow. Furthermore, PRGA offers various executable Python scripts for generating the Verilog testbenches, constraints, and Makefiles to automate the flow. A valid and verified bitstream is thus only a few commands away from the HDL inputs.

With the RTL-level model of the FPGA, we can simulate the FPGA from power-on reset, bitstream loading, all the way to application emulation. This *Emulation-over-Simulation* approach offers great fidelity of the architecture, and simplifies debugging both the FPGA and the application. Moreover, after pushing the RTL through an ASIC implementation flow, we can run gate-/transistor-level simulation with more accurate timing and power characteristics.

3 ARCHITECTURE

In this section, we describe PRGA's highly flexible and customizable architecture.

3.1 Routing Resources

PRGA currently supports straight, uni-directional routing tracks, global nets, and direct inter-block wires. Routing tracks, also called wire segments, are grouped by length in the unit of logical tiles. By default, PRGA fills every routing channel in the custom FPGA with all groups of tracks. However, if certain tracks are not driven by any `Switch Box` or `Connection Box`, they are depopulated from the specific routing channel, enabling per-wire, per-channel customization. We intentionally dropped support for bi-directional routing tracks because they are not compatible with common ASIC implementation flows, and they are no longer used in newer generations of commercial FPGAs either. Global nets and direct inter-block wires (`Tunnel`s in PRGA's terminology) are non-programmable routing resources. Global nets are typically used for clock or reset trees, while `Tunnel`s are usually used to implement fast carry-chains.

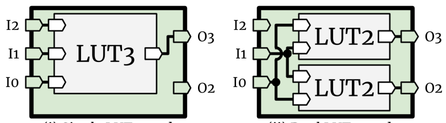
3.2 Views

As outlined in Sec. 2.2, PRGA decouples the logical abstraction, configuration circuitry, and physical implementation of the FPGA to maximize modularization. This is achieved by using different `View`s, a concept borrowed from the EDA world, at different steps throughout the FPGA design flow:

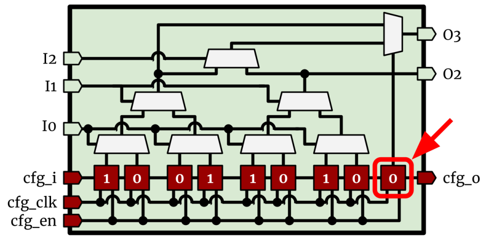
The `Abstract View` offers a high-level abstraction of the architecture. It is primarily used during the architecture customization and CAD script generation steps.

The `Design View` is usually the output of the *Configuration Circuitry Insertion* pass, implementing the `Abstract View` specified by users. Synthesizable Verilog can be generated from this view.

The `Physical View` offers more flexibility over the `Design View` and is primarily reserved for extensions. One typical use of this



(a) Two modes in `Abstract View`



(b) `Design View` implemented with the `Scanchain` configuration circuitry. Mode selection bit highlighted with red box and arrow.

Figure 3: Different views of a fracturable LUT3 which can be used as one single LUT3, or two LUT2s with shared inputs.

view is to generate RTL files for ASIC implementation, which are different from those used in simulation.

3.3 Hierarchy and Customizability

In this section, we dive into how PRGA organizes the hierarchy of the FPGA architectures and show the customizability of each level in the hierarchy.

3.3.1 Logic Primitive. Also known as logic elements or logic resources, `Logic Primitive`s are the building blocks of FPGAs. In PRGA, all hard logic that is targeted by technology mapping and synthesis are categorized as `Logic Primitive`s, including but not limited to LUTs, flip-flops, hard arithmetic units, SRAM macros, or even complex IP cores like memory controllers.

`Logic Primitive`s are further classified into three types:

- *Non-Programmable* primitives, e.g., simple flip-flops.
- *Programmable* primitives, e.g., LUTs. The functions provided by these primitives are altered by configuration.
- *Multi-Modal* primitives, e.g., fracturable LUTs. These primitives can be configured to one of its many modes, each offering a unique function. Multi-modal primitives are not directly targeted by synthesis. Instead, logical primitives emulated by its various modes are targeted by synthesis, then matched with the corresponding mode during packing.

Ideally, we would like to design primitives in a configuration-agnostic manner to maximize reusability and extensibility. Unfortunately, this is not always feasible. For example, LUTRAM, one mode of a LUT which can be used as a RAM, requires the capability of writing to the underlying configuration memory. Addressing this issue, PRGA exploits the views as discussed in Sec. 3.2. PRGA uses the `Abstract View` to specify the logical function of a primitive, and uses the `Design View` for the actual implementation. Fig. 3 shows the two `Abstract View`s and one `Design View` of a "fracrturable" LUT3, which can be used as one single LUT3, or two LUT2s with shared inputs.

All three types of primitives are customizable in PRGA. Non-programmable primitives can be easily added by creating new **Logic Primitive** objects with their interfaces and Verilog source files. Programmable primitives can be added in a similar way, only that an additional **Design View** matching the selected configuration circuitry is required. Multi-modal primitives require more hands-on development: Each mode of a multi-modal primitive is similar to a **Slice** in the **Abstract View**. A **Design View** implementing all the different modes and the mode-selection mechanism specific to the selected configuration circuitry is also required.

3.3.2 Slice, Logic Block, and IO Block. **Slice**s (also called ALMs or clusters), **Logic Block**s, and **IO Block**s are modules built upon **Logic Primitive**s, as shown in Fig. 2 (c). The purpose of these modules are three-fold: For the FPGA CAD tools, they serve as the targets for packing as well as routing terminals for place & route; For RTL-level simulation, they serve as hierarchical wrappers around **Logic Primitive**s, making the latter easier to monitor and debug; For ASIC implementation, they can be designed as the lowest-level physical block and reused across the chip, minimizing performance variation across block instances.

Logic Blocks may be logically wide, tall, or both, spanning across multiple tiles. It is also configurable if routing tracks are allowed to run through or over these large blocks. This logical size only affects the topology of the routing resources around it, and may mismatch the physical size or shape in the ASIC implementation.

3.3.3 Connection Box and Switch Box. Routing boxes are categorized into two classes in PRGA: **Connection Box**es that connect routing tracks to the pins of **Logic Block**s and **IO Block**s; and **Switch Box**es that connect routing tracks to other routing tracks.

By default, each side (north, east, south, and west) of a block needs a **Connection Box**, although it is very common to omit some **Connection Box**es when there are no routable pins on certain sides of a block. Fig. 2 (c) shows an example where **Connection Box**es are only needed on the east and west side of the **Logic Block**. Creating more **Connection Box**es for the same side of a block is also supported, enabling fine-grained customization around different instances of the same block. For tall and wide blocks, the number of **Connection Box**es needed on each side is equal to the width or height of the block.

Switch Boxes are modeled in a highly flexible way. As shown in Fig. 2 (a) & (b), each corner (northeast, northwest, southeast, and southwest) in a tile fits one **Switch Box**, allowing up to 4 **Switch Box**es wherever routing channels cross, although it's not necessary to fill all corners. No limit is set on the number of distinct **Switch Box**es used across the FPGA, though in practice, a small set of **Switch Box**es are often reused to reduce design complexity.

Since uni-directional routing tracks cannot have more than one driver, a **Connection Box** may conflict with a **Switch Box** when they drive the same routing track. To solve this conflict, PRGA adds bridging nets to connect **Connection Box** outputs into **Switch Box**es and merges them to the switches inside **Switch Box**es, as highlighted in Fig. 2 (b).

3.3.4 Tile. A **Tile** wraps one **Logic Block** instance or multiple instances of an **IO Block**, together with the **Connection Box**es around them. Fig. 2 (c) shows a tile wrapping one **Logic Block** and two **Connection Box**es. Multiple **Tile**s can be created for the same block, enabling the usage of different **Connection Box**es as mentioned in Sec. 3.3.3. Helper functions are provided for automatically generating **Tile**s for each block, as well as properly creating, populating, and instantiating **Connection Box**es.

This extra level of hierarchy is primarily aimed to improve ASIC implementation, and may be useful for certain configuration protocols. When used as the lowest-level physical block other than the **Logic Block** wrapped in it, it allows the EDA tools to work on a larger design and exploit more optimization opportunities, while guaranteeing that each **Logic Block** has only one physical implementation.

3.3.5 Array. The top level in the FPGA architecture hierarchy is an **Array** composed of **Tile**s, **Switch Box**es, and nested **Array**s. Fig. 2 (a) & (b) shows the nesting and internals of **Array**s. Each **Array** is a 2-dimensional mesh. Each tile in the mesh can accommodate up to four **Switch Box**es (one on each corner) plus one **Tile** or **Array**. Large **Tile**s or **Array**s may occupy more than one tile. **Switch Box**es are not allowed at any routing channel cross-points that are covered by these large **Tile**s or **Array**s, and routing tracks around them are truncated if not allowed to run through or over them. PRGA offers helper functions for automatically, correctly connecting the wires in an **Array**. If any built-in routing box population algorithm is used, PRGA can also automatically create, populate and instantiate proper **Switch Box**es.

PRGA offers great regularity and scalability through the use of nested **Array**s. Special configuration circuitry features such as partial or dynamic reconfiguration can also take advantage of this flexible hierarchy. For example, to implement partial dynamic reconfiguration, we can divide the FPGA into dynamically reconfigurable regions, each region being an **Array**. We can add one configuration controller per region and expose a well-defined interface of the controller into adjacent regions, enabling dynamic reconfiguration from adjacent regions.

3.4 Configuration Circuitry

Configuration circuitry, including memory cells and peripheral circuits, is one of the biggest commercial secrets in the FPGA industry yet is often neglected in FPGA architecture research, despite it making up a large proportion of on-chip area (20%-40% reported [12, 14]) and static energy consumption. Studying and modeling configuration circuitry is necessary to fully understand FPGA architecture implications. In addition, optimization of configuration circuitry enables novel architectures [7, 9, 11, 16].

Two types of configuration circuitry and comprehensive supports for them are included in PRGA at the time of this paper: **Scanchain** and **Pktchain**, respectively.

3.4.1 Scanchain. As its name suggests, **Scanchain** employs a shift register chain across the entire FPGA. The bitstream format is also straightforwardly a literal stream of bits. The chain may be single-bit wide or multi-bit wide, offering a trade-off between faster configuration and higher metal usage. Fig. 3b shows the single-bit configuration chain segment inside a fracturable LUT3.

Though it seems naïve, careful design is still required to avoid potential hazards:

- **Configuration reset and enable.** Even a small FPGA may contain thousands of configuration bits, and this number grows quickly into millions as the size of the FPGA increases. To synchronize the reset and enable signals for this huge number of registers, PRGA automatically registers the reset and enable signals along the hierarchy.
- **Chain ordering with physical considerations.** Logically, the registers on the chain can be ordered arbitrarily. Physically, however, it is better to put the configuration bits close to the modules they control, and order the chain so that registers are connected only to adjacent registers. PRGA makes good guesses by default, but user-provided hints are also accepted.

3.4.2 Pktchain. For large FPGAs, **`Scanchain`** is not only slow but also energy-hungry, because all the registers on the scan chain must be enabled during programming. **`Pktchain`** addresses this issue by dividing the single **`Scanchain`** into segments and adding high-bandwidth NoC routers between them. This design only adds a small amount of extra wires, minimizing the impact on metal resources that are precious for the logical routing resources. The high bandwidth of the NoC allows fast delivery of the bitstream segments. Once delivered, each bitstream segment is independently shifted in the corresponding chain segment. Multiple chain segments can operate in parallel, thus increasing the programming speed. This design also greatly reduces unnecessary switching of the registers, thus reducing the energy consumption.

3.4.3 Extensions. **`Scanchain`** and **`Pktchain`** are both developed upon the low-level API provided by PRGA, and they are great proof-of-concept designs showcasing the strong extensibility of the framework. Other configuration circuitry types, for example, the industry-standard SRAM-based design, can be added in a similar way. It is also possible to implement more complex configuration circuitry and protocols, such as the dynamic partial reconfiguration design described in Sec. 3.3.5.

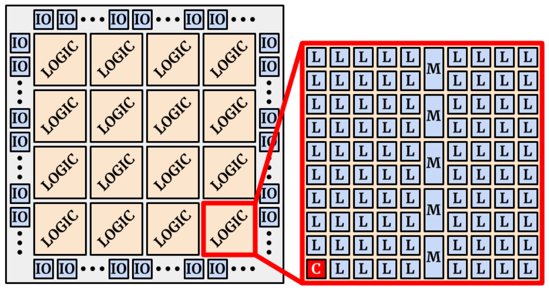
4 CASE STUDY: BUILDING A 14K-LUT6 FPGA

In this section, we illustrate the FPGA design flow by going through the process of building an example FPGA using PRGA. Fig. 4 shows the abstract, hierarchical floorplan and the configuration circuitry settings. Table 1 summarizes the key parameters of the FPGA. Listing 1 shows the Python script for building this example FPGA using the PRGA Python API. This FPGA provides 14240 multi-modal LUT6s, 28480 registers, 2.56Mbits memory, 1279 GPIOs, and one IO dedicated for clock.

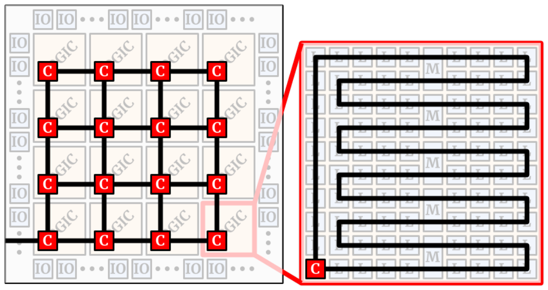
This example design is very similar to the synthesizable FPGAs presented in prior works [12, 24]. PRGA is capable of generating very different architectures, and we choose the similar design so that we can evaluate and compare our design with prior works.

4.1 Database Preparation

We choose the FLE6 multi-modal primitive used in prior works [12, 24] as the basic logic element for this example FPGA. FLE6 supports two modes: (1) one LUT6 and an optional D-flipflop, (2) two BLEs that are independently configurable. Each BLE also supports two modes: (1) one LUT5 and an optional D-flipflop, (2) two LUT4s, one



(a) Floorplan of the example FPGA. Top-level `Array` is composed of LOGIC `Array`s and `IO Tile`s. Each LOGIC `Array` consists of 89 `Logic Block`s (L), 5 BRAMs (M), and 1 physical-only configuration router (C). Each `IO Tile` provides 8 GPIOs. Each `Logic Block` contains 10 `FLE6` (multi-modal LUT6 with hard adders and flipflops). Each BRAM provides 32Kbits memory. Routing boxes are omitted in the figure.



(b) Configuration network and chain segments

Figure 4: Floorplan and configuration circuitry of the example FPGA in Sec. 4.

hard adder, and one optional D-flipflop. The figure describing the structure of this primitive can be found in [12].

In addition, a third-party $512 \times 64b$ SRAM macro is added to the build. To give more flexibility to synthesis and packing, we can add extra muxing logic around the SRAM to support smaller word sizes, for example, $1K \times 32b$, $2K \times 16b$, $4K \times 8b$, etc. This is achieved by creating **`Abstract View`**s for each mode, writing custom Verilog files, and creating a **`Design View`** to link them together, as shown in line 7-40 in Listing 1 (Verilog not shown).

4.2 Architecture Customization

After adding all the custom **`Logic Primitive`**s, we can start customizing the FPGA architecture. First, routing tracks and global nets are added to the FPGA, as shown in line 43-46 in Listing 1. Then, we customize each **`Logic Block`** and **`IO Block`** (line 50-81). As discussed in Sec. 2.2, we only need to describe the functional structures of these blocks. Pack patterns (line 62) and direct inter-block wires (line 70) are added for the carry chain. After describing the blocks, we construct **`Tile`**s (line 85-94) and **`Array`**s (line 100-116), then adopt built-in algorithms to automatically create, populate, and instantiate routing boxes.

4.3 Transformation and Generation Passes

After specifying the FPGA architecture, we prepare our Jinja [25] text renderer, then set up the transformation and generation passes,

Listing 1: Python script for building the example FPGA described in Sec. 4. Some API syntax is abbreviated for clarity. This script alone drives the entire FPGA design flow. It also generates all the RTL as well as all the files needed by the CAD tools.

```python
from prga import *

# == database preparation ==
#   ctx is an Context object for our workspace
ctx = Pktchain.new_context(noc_width=8, chain_width=1)

# == design and add multi-modal primitive for block RAM ==
#   bdr is a Builder object
bdr = ctx.build_multimode(name="memory")
bdr.create_clock (name="clk")
bdr.create_input (name="D", width=64)
bdr.create_output(name="Q", width=64)
#   ... and more ports

# -- create abstract views for the modes --
for  name,      AW, DW in [
    ["512x64b", 9,  64],
    ["1K32b",   10, 32],
    # ... and more modes
  ]:
  mb = bdr.build_mode(name="mode_"+name)
  # abstract view of non-programmable memory modules
  dpram = ctx.build_memory(name="dpram_"+name,
      addr_width=AW, data_width=DW, vpr_model="dpram",
      ).commit()
  inst = mb.instantiate(model=dpram, name="i_ram_core")
  mb.connect(mb.ports["clk"],     inst.pins["clk"])
  mb.connect(mb.ports["D"][0:DW], inst.pins["data1"])
  mb.connect(inst.pins["out2"],   mb.ports["Q"][0:DW])
  # ... and more connections
  mb.commit()

# -- link design view --
# user Verilog implementation provided by user
bdr = bdr.build_logical_counterpart(
    verilog_template="memory.v")
#   ... design-specific settings

# -- commit the design --
memory = bdr.commit()

# == add routing resources ==
glb_clk = ctx.create_global(name="clk", is_clock=True)
glb_clk.bind(position=(0, 21), subtile=0) # bind to an IO
l4  = ctx.create_segment(name='L4',  width=32, length=4)
l16 = ctx.create_segment(name='L16', width=1,  length=16)

# == customize blocks ==
# -- logic block --
bdr = ctx.build_logic_block(name="clb")
clk = bdr.create_global(global_=glb_clk,      side="south")
in_ = bdr.create_input (name="in",  width=30, side="east")
ci  = bdr.create_input (name="ci",  width=1,  side="south")
out = bdr.create_output(name="out", width=20, side="east")
co  = bdr.create_output(name="co",  width=1,  side="north")
xbar_i, xbar_o = [], []
xbar_i.extend(in_)
for idx, inst in enumerate(bdr.instantiate(
    model=ctx.primitives["fle6"], name="i_fle", reps=10):
  bdr.connect(clk, inst.pins["clk"])
  bdr.connect(inst.pins["out"], out[idx*2:(idx+1)*2])
  bdr.connect(ci, inst.pins["cin"],
      vpr_pack_patterns=["carrychain"])
  ci = inst.pins["cout"]
  xbar_i.extend(inst.pins["out"])
  xbar_o.extend(inst.pins["in"])
bdr.connect(ci, co)
# ... connect each "xbar_o" to 50% "xbar_i"s
clb = bdr.commit()
```

Listing 2: (Continued)

```python
# -- create direct inter-block wires --
ctx.create_tunnel(name="carrychain", offset=(0, -1),
    source=clb.ports["cout"], sink=clb.ports["cin"])

# -- IO block --
bdr = ctx.build_io_block(name="iob")
#   ... ports, instances and connections
iob = bdr.commit()

# -- BRAM block (tall block) --
bdr = ctx.build_logic_block(name="bram", width=1, height=2)
bdr.instantiate(model=memory, name="i_ram")
#   ... ports, instances and connections
bram = bdr.commit()

# == automatically create and populate tiles ==
# -- IO tiles --
iotiles = {}
for edge in ["west", "north", "east", "south"]:
  bdr = ctx.build_tile(block=iob, capacity=8, edge=edge)
  bdr.fill(fc=(0.15, 0.15)).auto_connect()
  iotiles[edge] = bdr.commit()

# -- CLB tile --
bdr = ctx.build_tile(block=clb)
clbtile  = bdr.fill(fc=(0.055, 0.1)).auto_connect().commit()

# -- BRAM tile --
bdr = ctx.build_tile(block=bram)
bramtile = bdr.fill(fc=(0.055, 0.1)).auto_connect().commit()

# == arrays ==
sb_pat = SwitchBoxPattern.cycle_free

# -- LOGIC array --
bdr = ctx.build_array(name="logic", width=10, height=10)
for x, y in product(range(bdr.width), range(bdr.height)):
  if x == 5:
    if y % 2 == 0:
      bdr.instantiate(model=bramtile, position=(x, y))
  elif not (x == 0 and y == 0): # reserved for router
    bdr.instantiate(model=clbtile, position=(x, y))
logic = bdr.fill(sbox_pattern=sb_pat).auto_connect().commit()

# -- top-level array --
bdr = ctx.build_array(name="top", width=42, height=42,
    is_top=True)
#   ... instantiate LOGIC arrays and IO tiles
top   = bdr.fill(sbox_pattern=sb_pat).auto_connect().commit()

# == customize configuration circuitry insertion ==
#   callback function for customize router & chain ordering
def order_submodules(module):
    #   ...

# == apply transformation and generation passes ==
# -- Jinja2 template renderer --
renderer = Pktchain.new_renderer()

# -- transformation and generation flow --
flow = Flow(
    TranslationPass(),  # generate design views
    Pktchain.Insert(order_submodules),
    VPR_Arch_Gen(output_file="vpr/arch.xml"),
    VPR_RRG_Gen(output_file="vpr/rrg.xml"),
    Yosys_Scripts_Gen(output_dir="syn"),
    Verilog_Gen(output_dir="rtl"),
    #   ... and more
    )

# -- fire the workflow --
flow.run(ctx, renderer)

# -- save core data structure on disk --
ctx.pickle("ctx.pkl")
```

Param.	Value	Note
Routing Resources		
W	288	Routing Channel Width
$n \times L$	1×16	Track count & length per direction per channel
	32×4	
F_s	3	`Switch Box` connectivity. Pattern: cycle-free [15]
`Logic Block`		
N_{FLE6}	10	#FLE6s per block
X	50%	Local connectivity
$F_{c,in}$	0.055	`Connection Box` input connectivity
$F_{c,out}$	0.1	`Connection Box` output connectivity
BRAM		
m	32Kbits	Memory capacity
$F_{c,in}$	0.055	`Connection Box` input connectivity
$F_{c,out}$	0.1	`Connection Box` output connectivity
`IO Block`		
c	8	#IOs per block
$F_{c,in}$	0.15	`Connection Box` input connectivity
$F_{c,out}$	0.15	`Connection Box` output connectivity
Configuration Circuitry: `Pktchain`		
B	8b	NoC data width
W_c	1b	Leaf scanchain width

Table 1: Parameters of the example FPGA built in Sec. 4

as shown in line 125-136 in Listing 1. The `Flow` object then resolves the dependencies between the passes and determines the correct execution order. Finally, we start the `Flow` to apply all the passes to the core `Context` data structure, `ctx` (line 139).

After all the passes finish transforming the core data structure or generating files, we save the core data structure onto hard disk for future use (line 142).

4.4 ASIC Implementation

The FPGA design is broken down into 3 levels of hierarchy and 5 physical blocks: 1) CLB `Tile`; 2) BRAM `Tile`; 3) IO `Tile`; 4) LOGIC `Array` composed of CLB `Tile`s and BRAM `Tiles`; and 5) Top-level `Array` composed of LOGIC `Array`s and IO `Tile`s. Note that this partitioning is for this example design only, and can be changed freely for other designs. We then lay out the FPGA using standard cell libraries on a state-of-the-art FinFET technology.

As discussed in Sec. 3.3.4, `Tile`s are chosen as the bottom-level physical block instead of `Logic Block`s to improve the Quality of Result (QoR). In the LOGIC `Array`, `Tile` instances are placed in almost perfect alignment with their logical positions. We adopt the cycle-free `Switch Box` pattern [15] and flatten the `Switch Box` instances in the `Array`s. Compared to a black-boxed approach in which blocks and routing boxes are designed individually then simply stitched together at the top level, this design flow applies proper constraints at `Array` level, enabling the EDA tools to resolve many hazards that are otherwise unidentifiable, for example, hold time violations, clock skew, crosstalk, IR drop, etc.

Different challenges arise when designing the top-level `Array` because of its scale. To minimize EDA tool runtime without sacrificing QoR, we reduce the logic left in the top-level `Array` to the minimum. Pins of the LOGIC `Array`s are also aligned, reducing wiring congestion.

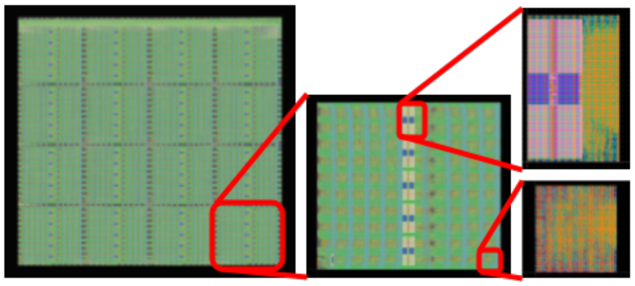


Figure 5: Layout photos of the custom FPGA

Area (μm^2)	[12]	[24]	Stratix IV [1]	This work
LAB/CLB	15333	- [2]	- [2]	15254
Tile	30625 (277%)	17648 (160%)	11050 (100%)	34209 (310%)

[1] Data reported by [12].
[2] Unreported.

Table 2: Area Comparison

Path Delay (ns)	[12]	[24]	Stratix IV [1]	This work
LUT-5	0.46 (170%)	0.14 (52%)	0.27 (100%)	0.55 (204%)
LUT-6	0.50 (179%)	0.15 (54%)	0.28 (100%)	0.64 (229%)
1-bit Adder	0.70 (90%)	0.54 (70%)	0.77 (100%)	0.72/0.09 [2] (94/12%)
20-bit Adder	1.63 (133%)	1.10 (89%)	1.23 (100%)	0.98/0.82 [2] (80/67%)
Local Routing	0.27 (159%)	0.12 (71%)	0.17 (100%)	0.34/0.92 [3] (200/541%)
L4 Track [4]	2.53 (429%)	0.40 (68%)	0.59 (100%)	2.22 (376%)
L16 Track [4]	4.02 (394%)	0.78 (76%)	1.02 (100%)	2.91 (285%)
L4 $\longrightarrow$L4 Switch [5]	-	-	-	1.21
L4 $\longrightarrow$L16 Switch [5]	-	-	-	1.20
L16 $\longrightarrow$L4 Switch [5]	-	-	-	1.20
L16 $\longrightarrow$L16 Switch [5]	-	-	-	1.46

[1] Data reported by [12].
[2] First number is the delay from addends to carry-out; second is carry-in to carry-out.
[3] First number is the delay from block inputs to FLE6 inputs; second is the delay of the feedback connections from FLE6 outputs back to FLE6 inputs.
[4] [12] and [24] have different definitions for this metric. We adopt the definition from [12], i.e. delay from a CLB output pin to a CLB input pin, connected via one straight track.
[5] Delay from the driver of the source track to the driver of the destination track.

Table 3: Representative Path Delays

Once the layout is finished and verified, Static Timing Analysis (STA) using automated EDA tools can be applied to extract the timing and power characteristics from the FPGA. This information can be passed back into the generation passes to get timing-annotated scripts for the HDL-to-bitstream toolchain.

5 EVALUATION

5.1 ASIC Implementation

In this section, we evaluate the layout of the example FPGA built in Sec. 4, and compare the results with previous works [12, 24] and

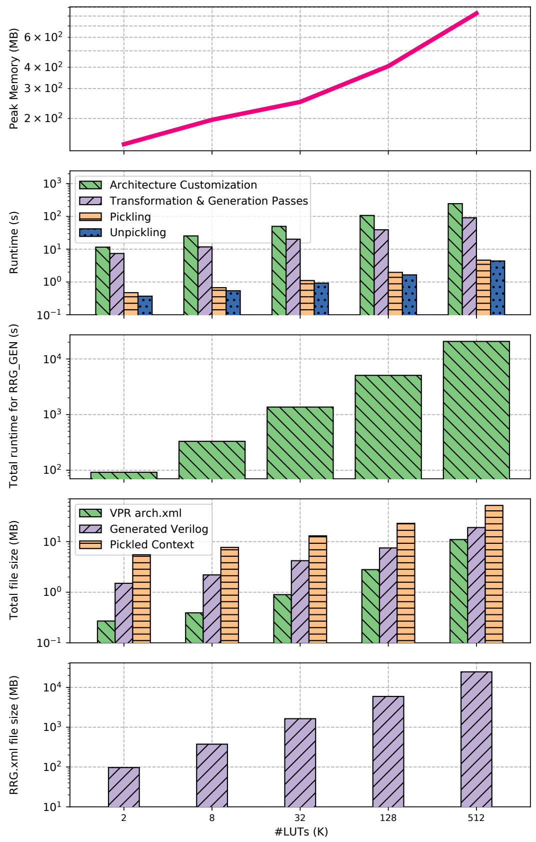


Figure 6: Scalability and performance of PRGA as the platform itself. All axes are in log scale.

Stratix IV (data reported by [12]). Since we are not using 40nm technology, we use the CMOS scaling model [22] to scale our evaluation results to 40nm.

5.1.1 Area. Table 2 summarizes the area comparison. For **`Logic Block`**, our result is almost the same as the previous standard-cell-based FPGA [12] (within 1% difference). As for **`Tile`** area (including **`Connection Box`**es and **`Switch Box`**es), our area is 11.7% larger than the previous standard-cell-based FPGA [12], 1.94x the area of the previous FPGA built with standard cells and custom switches [24], and 3.1x the area of Stratix IV. However, the overall utilization rate is much lower than 80% as reported by prior works due to the use of a FinFET technology, and the area may be optimized further by using a more efficient type of configuration circuitry.

5.1.2 Performance. The typical path delays inside the **`Logic Block`** in this work are slightly worse than the previous standard-cell-based FPGA [12], but the routing track delays are slightly better. However, when compared to Stratix IV, or the FPGA built with both standard cells and custom switches [24], the example FPGA in this work is almost 2-4x worse.

5.2 Scalability

In this section, we evaluate the scalability and performance of PRGA as a software. We scale the architecture described in Sec. 4 from 2K LUT6s up to 512K LUT6s, and compare the memory usage, runtime, and output file sizes.

As shown in Fig. 6, PRGA has a low memory footprint, and except for *Routing Resource Graph* (RRG) generation, PRGA only needs a short amount of time to generate all the files needed. Memory usage and runtime both scale linearly as the capacity of the architecture increases, but even for the half-million-gate architecture, PRGA finishes with less than 1GB memory within 5 minutes.

The bottleneck is RRG generation. RRG can be read by VPR in addition to the architecture specification XML, overriding the default auto-generated routing resource graph in VPR at a per-wire, per-connection basis, thus enabling the high customizability PRGA aims for. However, RRG was initially an internal file format designed for debugging and not intended to scale. Therefore, it stores the the entire routing graph of an FPGA in human-readable XML format, which easily grows beyond Gigabytes as the FPGA size grows. This has been noticed by the maintainers of VPR, and updates are being planned.

6 ENABLED APPLICATIONS

This section outlines some of the compelling applications that PRGA enables. PRGA can be used not only as a research platform but also as a generator of custom FPGA IP cores.

6.1 FPGA Architecture Research

PRGA enables research into the architectures of FPGAs. The need for a good low-level FPGA framework comes in multiple forms. First, PRGA can be used as a perfect platform to perform FPGA design optimization. Coupled with an optimization engine (hill-climbing, genetic algorithm, simulated annealing, etc.), PRGA is the ideal platform to automate the optimization of the hardware side of an FPGA design. The high configurability of PRGA enables automated searching of a huge design space. FPGA optimization becomes even more powerful when optimizing for a set of applications or application domains [13].

Second, PRGA is complementary to high-level design tools such as VPR. While high-level FPGA design tools enable even faster architectural design space exploration, they can miss many of the implementation details which are especially important in advanced device nodes. This leads to a complimentary need for low-level design tools such as PRGA which can be run through commercial EDA tools to gather accurate power, performance, and area results that can be used by researchers to evaluate different FPGA designs.

An important feature of PRGA is that the configuration circuitry, configuration topology, and storage strategy are all customizable and independent of the design hierarchy. This flexibility enables research into the design of the configuration circuitry, which inherently requires the low-level capabilities that PRGA provides. PRGA also supports dynamic and partial reconfiguration via its SoC interface which opens the door to architectures like DPGA [9] and Tabula [26].

6.2 FPGA CAD Research

PRGA can be used as a target for FPGA CAD tool research. FPGA CAD tools need FPGA platforms to target. PRGA provides copious different designs and, most importantly, can be characterized to

Project	[17]	[14]	[12]	[24]	This work
Std. Cells	✓	✓	✓	✓	✓
Custom Cells	✗	✗	✗	✓	✓
Hetero. Blocks [1]	✗	✗	✓	✓	✓
Custom Config.	✗	✗	✗	✓[2]	✓
Split SB/CB	✗	✗	✗	✗	✓
Custom Hier. [3]	✗	✗	✗	✗	✓
Open-Source	✓	✗	✗	✓	✓

[1] Support for tall/wide blocks.
[2] Configuration pins may be exposed for manual connection; No bitstream generation or RTL support.
[3] Customizable hierarchy of generated RTL.

Table 4: Taxonomy of FPGA Prototyping Works

provide area, timing, and energy information to feed the FPGA CAD tools.

Beyond CAD tools for FPGAs, PRGA is also a good candidate for being a test design for the greater ASIC CAD tool research area. By providing large and parameterized designs, PRGA can be used as a parameterizable benchmark to test chip-design CAD tools. PRGA has already been used as a test case for some emerging open source CAD tools.

6.3 SoC Integration

PRGA is not only a research platform. It generates FPGA fabrics that can be implemented in chips. One of the most promising use cases where free (open-source) or customized FPGAs are desirable is as an embedded FPGA in a System on Chip (SoC). SoCs are becoming increasingly heterogeneous, and it is difficult to always know what functionality is needed in an SoC before fabrication. Likewise, applications are evolving faster than ASIC design cycles, pushing more designs to reconfigurable hardware. This has created a growing need for embedded FPGAs [3]. Unlike commercial embedded FPGA providers (e.g., Achronix, Flex Logix, QuickLogic) which require licensing fees, PRGA's open-source nature provides an opportunity to add embedded FPGA functionality without IP licensing cost. This can be especially impactful in two use cases (1) when only a very small amount of FPGA resources are needed or (2) when the embedded FPGA functionality is needed in low-budget or academic designs.

7 RELATED WORK

In this section, we qualitatively compare PRGA with other FPGA prototyping works. Table 4 summarizes the key differences between this work and previous works.

Archipelago [17] is the first open-source project for designing standard-cell-based FPGAs. It is developed in Chisel [2] and coarsely parameterized, providing limited customizability. It uses an early version of VTR for bitstream generation. Various designs generated by Archipelago were placed and routed using a 65nm process technology. Unfortunately, the project is no longer maintained.

There are two [12, 14] successful, standard-cell-based FPGAs built with commercial EDA tools. Based on the VPR/VTR series [18], these projects provide independent data points in the design space, but are not open sourced as publicly available research tools.

OpenFPGA [24] is another open-source FPGA prototyping framework that is well-maintained and actively developed. OpenFPGA is capable of generating synthesizable RTL from an extended XML schema based on the VTR architecture description file format. Most VTR features are supported, including heterogeneous blocks, multimodal primitives, carrychains, BRAMs, and so on. In addition, user-defined components and custom cells are also supported. The major differences between PRGA and OpenFPGA are the followings:

(1) Currently, OpenFPGA can only generate RTL in a fixed hierarchy: Top-level array → logic and IO blocks, routing boxes → programmable switches and primitives. PRGA provides more customizability in module hierarchy and RTL generation, which not only enables more diverse ASIC implementation strategies but also improves scalability.
(2) OpenFPGA is developed on top of VTR [18] and tightly integrated with the VTR codebase. PRGA is instead decoupled from VTR and uses VTR only as an open-source FPGA place-and-route tool. This decoupling, along with the modularized workflow, greatly reduces the bar for PRGA users to customize and extend the framework, and allows them to focus on their interested part.
(3) OpenFPGA currently supports 4 types of configuration circuitry: chain-based, frame-based, memory-banked, and flattened, in which the last one exposes all configuration bits as ports of the RTL modules, enabling custom configuration circuitry with limited support for bitstream generation. PRGA allows users to directly modify RTL generation, and provides various levels of support for bitstream generation.

8 CONCLUSION

In conclusion, PRGA is a great platform for building custom FPGAs. Whether it be enabling researchers or SoC chip builders, the open-source nature, easy-to-use design, and high configurability make it an excellent low-level FPGA framework in a post-Moore's Law world. By providing full Verilog RTL of all of the generated designs, area and performance can be modeled with ultimate accuracy. PRGA is scalable and provides reasonable runtimes even for large designs (an important consideration for commercial use). PRGA provides comparable area and timing delay as other projects built out of the same technology (standard-cell-based FPGAs). We look forward to many years of supporting PRGA and expect it to serve as the basis for FPGA research and FPGA designs for years to come.

9 ACKNOWLEDGEMENTS

We want to thank our colleagues from the Princeton Parallel Group, especially Ting-Jung Chang, and Fei Gao, for their help with the ASIC backend flow. This material is based on research sponsored by the NSF under Grant No. CCF-1453112, Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7852. 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 Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), the NSF, or the U.S. Government.

REFERENCES

[1] Amazon. 2020. *Amazon EC2 F1 Instances.* https://aws.amazon.com/ec2/instance-types/f1/

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12).* Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[3] Brian Bailey. 2019. *The Case For Embedded FPGAs Strengthens And Widens.* https://semiengineering.com/embedded-fpga-becomes-a-viable-option/

[4] Vaughn Betz. 2000. *Architecture and CAD for Speed and Area Optimization of FPGAs.* Ph.D. Dissertation. University of Toronto.

[5] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49).* IEEE Press, Article 7, 13 pages.

[6] Yao-Wen Chang, D. F. Wong, and C. K. Wong. 1996. Universal Switch Modules for FPGA Design. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (Jan. 1996), 80–101. https://doi.org/10.1145/225871.225886

[7] Paul Chow, Soon Ong Seo, Jonathan Rose, Kevin Chung, and P Gerard. 1999. The Design of an SRAM-Based Field-Programmable Gate Array — Part I : Architecture. 7, 2 (1999), 191–197.

[8] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. 2010. Single-chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* (2010), 225–236. https://doi.org/10.1109/MICRO.2010.36

[9] André DeHon. 1996. DPGA Utilization and Application. In *Proceedings of the 1996 ACM Fourth International Symposium on Field-Programmable Gate Arrays (FPGA '96).* Association for Computing Machinery, New York, NY, USA, 115–121. https://doi.org/10.1145/228370.228387

[10] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: Versal™ Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19).* Association for Computing Machinery, New York, NY, USA, 84–93. https://doi.org/10.1145/3289602.3293906

[11] Mingyu Gao, Christina Delimitrou, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Christos Kozyrakis. 2016. DRAF: A Low-Power DRAM-Based Reconfigurable Acceleration Fabric. *SIGARCH Comput. Archit. News* 44, 3, 506–518. https://doi.org/10.1145/3007787.3001191

[12] Brett Grady and Jason H. Anderson. 2018. Synthesizable Heterogeneous FPGA Fabrics. In *2018 International Conference on Field-Programmable Technology (FPT).* 222–229.

[13] Mark Hammerquist and Roman Lysecky. 2008. Design Space Exploration for Application Specific FPGAs in System-on-Chip Designs. In *2008 IEEE International SOC Conference.* 279–282.

[14] Jin Hee Kim and Jason H. Anderson. 2015. Synthesizable FPGA Fabrics Targetable by the Verilog-to-Routing (VTR) CAD Flow. In *25th International Conference on Field Programmable Logic and Applications, FPL 2015.* https://doi.org/10.1109/FPL.2015.7293955

[15] Ang Li, Ting-Jung Chang, and David Wentzlaff. 2020. Automated Design of FPGAs Facilitated by Cycle-Free Routing. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL).* 208–213. https://doi.org/10.1109/FPL50879.2020.00042

[16] Ting-Jung Lin, Wei Zhang, and Niraj K Jha. 2012. FPGA Based on 10T Low-Power SRAMs. 20, 11 (2012), 2151–2156.

[17] Hao Jun Liu. 2014. *Archipelago - An Open Source FPGA with Toolflow Support.* Master's thesis. University of Toronto.

[18] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 2, Article 6 (July 2014), 30 pages. https://doi.org/10.1145/2617593

[19] Gordon E. Moore. 1965. Cramming More Components Onto Integrated Circuits. *Electronics* (April 1965).

[20] Duncan J. M. Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H. W. Leong. 2017. High Performance Binary Neural Networks on The Xeon+FPGA™ Platform. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL).* 1–4.

[21] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling. *ACM Trans. Reconfigurable Technol. Syst.* 13, 2, Article 9 (May 2020), 55 pages. https://doi.org/10.1145/3388617

[22] Aaron Stillmaker and Bevan Baas. 2017. Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180nm to 7nm. *Integration, the VLSI Journal* 58, January (2017), 74–81. https://doi.org/10.1016/j.vlsi.2017.02.002

[23] SymbiFlow. 2020. *FPGA ASM (FASM).* https://fasm.readthedocs.io/en/latest/

[24] Xifan Tang, Edouard Giacomin, Aurélien Alacchi, Baudouin Chauviere, and Pierre-Emmanuel Gaillardon. 2019. OpenFPGA: An Opensource Framework Enabling Rapid Prototyping of Customizable FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL).* 367–374. https://doi.org/10.1109/FPL.2019.00065

[25] The Pallets Projects. 2020. *Jinja.* https://palletsprojects.com/p/jinja/

[26] Tom R. Halfhill. 2010. Tabula's Time Machine. Microprocessor Report.

[27] Steven J.E. Wilton. 1997. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory.* Ph.D. Dissertation. University of Toronto.

[28] Clifford Wolf. 2020. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[29] Xilinx. 2020. *Zynq-7000 SoC.* https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html