IEICE TRANSACTIONS

on Fundamentals of Electronics, Communications and Computer Sciences

DOI:10.1587/transfun.2022VLP0008

Publicized:2022/10/07

A PUBLICATION OF THE ENGINEERING SCIENCES SOCIETY

The Institute of Electronics, Information and Communication Engineers

Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on VLSI Design and CAD Algorithms*

An eFPGA Generation Suite with Customizable Architecture and IDE

Morihiro KUGA[†a)], **Qian ZHAO**[††], ***Members***, **Yuya NAKAZATO**[†††], ***Nonmember***, **Motoki AMAGASAKI**[†], ***Member***, ***and*** **Masahiro IIDA**[†], ***Senior Member***

SUMMARY From edge devices to cloud servers, providing optimized hardware acceleration for specific applications has become a key approach to improve the efficiency of computer systems. Traditionally, many systems employ commercial field-programmable gate arrays (FPGAs) to implement dedicated hardware accelerator as the CPU's co-processor. However, commercial FPGAs are designed in generic architectures and are provided in the form of discrete chips, which makes it difficult to meet increasingly diversified market needs, such as balancing reconfigurable hardware resources for a specific application, or to be integrated into a customer's system-on-a-chip (SoC) in the form of embedded FPGA (eFPGA). In this paper, we propose an eFPGA generation suite with customizable architecture and integrated development environment (IDE), which covers the entire eFPGA design generation, testing, and utilization stages. For the eFPGA design generation, our intellectual property (IP) generation flow can explore the optimal logic cell, routing, and array structures for given target applications. For the testability, we employ a previously proposed shipping test method that is 100% accurate at detecting all stuck-at faults in the entire FPGA-IP. In addition, we propose a user-friendly and customizable Web-based IDE framework for the generated eFPGA based on the NODE-RED development framework. In the case study, we show an eFPGA architecture exploration example for a differential privacy encryption application using the proposed suite. Then we show the implementation and evaluation of the eFPGA prototype with a 55nm test element group chip design.

key words: *FPGA-IP, eFPGA, FPGA CAD, FPGA architecture, application-specific acceleration*

1. Introduction

Over the past few decades, most computer systems—from microcontrollers to high-performance servers—have used a similar CPU-centric hardware architecture with software customization to implement various applications. However, as the high-performance and low-power requirements of computer systems become more increasingly important, it is necessary to extend the customizability of computer systems from software to hardware to achieve an optimal balance of programmability and computing performance. As a result, more and more computer systems have adopted field-programmable gate arrays (FPGAs)—a representative device of reconfigurable computing technology—to implement hardware accelerators for different target applications, such as deep learning, big-data processing, and media encoding, to achieve performance gains of tens to hundreds of times by offloading the processing of critical loads from the CPU to the hardware circuitry. An FPGA can provide hardware-level customization for domain-specified computing and is therefore considered an essential component of future computer systems.

The hardware-level programmability of FPGAs comes from their unique circuit architecture, where programmable lookup tables (LUTs) implement fine-grained logics, and then programmable routing resources connect these LUTs to complete user circuits. In an FPGA, a certain number of LUTs (called a logic cluster) and routing resources are designed to be a repeatable tile block, and a large number of these tiles are arranged in a two-dimensional array style on a chip fabric. Modern commercial FPGAs also include specific functional modules into the tile array, such as block ram (BRAM) and digital signal processors (DSPs), which are frequently used by most applications, to improve the implementation efficiency of user circuits. For example, the Xilinx ZYNQ [1] is a multiprocessor system-on-a-chip (SoC) product that contains ARM processors and an FPGA-IP fabric. Most commercial FPGAs are general-purpose logic devices designed with a full-custom flow. The trade-offs of various architecture parameters, such as the size of the LUTs, the size of a logic cluster, the number of routing resources, and the type and number of function modules, are carefully explored with a benchmark set consisting of various application circuits. Therefore, general-purpose FPGAs are designed to be common devices to achieve generally good performance for various application circuits on average, but they are not optimized for a specific application. So when using general-purpose FPGAs, it is often the case that one type of hardware resource is exhausted while many other types are wasted. Therefore, for computer system designers who care only about a specific application domain, customizable embedded

[†]The authors are with the Faculty of Advanced Science and Thechnology, Kumamoto University, Kumamoto-shi, 860-8555 Japan.

[††]The author's work was completed by February 2022 at Kyushu Institute of Technology, Japan. He is presently with ByteDance, Beijing, 100098 China.

[†††]The author's work was completed by March 2022 at the Dept. of Graduate School of Science and Thechnology, Kumamoto University, Kumamoto-shi, 860-8555 Japan.

a) E-mail: kuga@cs.kumamoto-u.ac.jp

FPGAs (eFPGAs) would be an attractive option for achieving better power, performance, and area than with general-purpose FPGAs.

Many commercial [2]–[4] and acadamic eFPGA [5]–[12] frameworks for SoC systems have been proposed. Such frameworks can automatically generate hardware description language (HDL) design code for eFPGAs based on user input FPGA architecture parameters. These eFPGAs are synthesizable logic devices. Although the performance of a synthesizable design is commonly lower than a full-custom design, the fully automated design flow allows SoC designers to quickly obtain the optimal FPGA-IP for their system and then easily integrate, verify, and manufacture it together with other SoC IPs. Most of these eFPGA frameworks are built on the open-source FPGA CAD toolchain Verilog-to-Routing (VTR) [13], and they develop HDL and bitstream generation tools after the VTR flow, thereby enabling automatic HDL generation and user-circuit compilation for some classic FPGA architectures. By using VTR, it is easy to confirm the performance of various FPGA architecture parameters in implementing the target application, and thus to find the optimized eFPGA needed by a SoC designer.

Many traditional eFPGA frameworks are targeted at being a push-button FPGA-IP generation solution [5], but there are some points that can be improved. First, synthesizable eFPGAs use flip-flops (FFs) as configuration memory. An FF has a larger area than that of the SRAM used in full-custom design, so to narrow the performance gap between an eFPGA and a full-custom FPGA, it is valuable to explore FPGA architectures with less use of configuration memories, but not simply adopting the traditional FPGA architectures. Second, traditional eFPGA frameworks do not have support for the shipping test. Many eFPGA frameworks provide an application-circuit-based test approach, in which a pre-prepared test circuit is loaded on the FPGA, a series of test patterns is input, and then the outputs are observed to verify the functionality of the FPGA. However, it is difficult to cover all hardware resources by using an application circuit. Especially for the routing resources, which occupy more than half of the FPGA area, there are many possible routing paths, so the traditional method cannot ensure 100% test coverage of the routing resources. Therefore, in the case of the traditional eFPGA framework, SoC designers have no sufficient shipping test method for their FPGA-IP to distinguish between good and bad dies, which is important for manufacturing processes with low yields. Third, traditional eFPGAs lack integrated development environment (IDE) support. Although most traditional eFPGA frameworks provide a CAD tool chain to compile user circuits to bitstream for the generated FPGA-IP, the use of these tools is basically based on the command line. It is necessary to provide SoC designers with a GUI-based, easy-to-use, and customizable IDE prototype for FPGA-IP development that they could integrate into their SoC system IDE.

To address these issues, this paper extends the previous work [14] and proposes a novel eFPGA generation suite that can automatically explore the optimal FPGA-IP architecture parameters based on the target applications, and automatically generate the FPGA-IP HDL, shipping test circuit set, and IDE for the FPGA-IP user. The main contributions of this research are summarized as follows. We also mention the four extended points from our previous journal papers.

- A novel less configuration memory logic cell, namely, the scalable logic module (SLM), was proposed to replace traditional LUTs [15]. The SLM reduces the area overhead of the FFs used in an eFPGA. We enable the logic-cell-level optimization by exploring the best SLM structure for the target application, thereby improving eFPGA performance further. The first extended point of this paper is that we explore the eFPGA architecture optimized for a user application and determined an appropriate architecture when designing an SLM-based eFPGA prototype.
- Our eFPGA generation suite has the full support of the shipping test. We adopt an easy-to-test FPGA architecture and a test method proposed in previous research [16], however, the test method only supported wiring resources. A homogeneous architecture that unifies the structure of all logic tiles is used to simplify testing. Therefore, we extend the test method for not only wiring resources but also SLM as a logic cell. It is the second extended point of this paper. Then we propose a test-circuit design method based on the topology of the Wilton-type switch box of global routing, which can quickly test all routing resources and identify the locations of stuck-at errors. Together with the logic test method, we can achieve 100% test coverage on the entire FPGA-IP. In addition, the shipping test bitstreams are automatically generated with the FPGA-IP for SoC designers.
- We propose a method for building a simple, easy-to-use, and customizable IDE based on the NODE-RED development framework, which has attractive features for modern applications such as Web availability and visualization. We find that it is easy for circuit designers to become familiar with and use an eFPGA IDE built using these features, and it is also easy for SoC designers to customize it for their SoC systems. We finally package and distribute all CAD tools as a docker image, which users can install locally or on a remote server conveniently and access and use through the Web interface. The third extended point is that our eFPGA design tool can be integrated as a GUI-based

design tool by NODE-RED instead of the conventional CUI-based operation.
- As a case study, we examine the proposed eFPGA generation suite with a differential privacy (DP) encryption circuit, explore the optimal FPGA-IP for the DP circuit, and implement and evaluate the FPGA-IP with a 55nm chip design as a prototype. And also we have developed the evaluation board for testing the prototype chip. We confirm that the prototype chip is running correctly. In addition, the chip was measured for its actual delay and compared with simulation delay by static timing analysis (STA). It is the last extended point of this paper.

Note that the target eFPGA in this study dealt with simple structures not including of hard macros such as DSP and Block RAM like a recent eFPGA.

The rest of this paper is organized as follows. We introduce related work in section 2 and give an overview of the proposed eFPGA generation suite in section 3. The architecture of our eFPGA is explained in section 4, and the shipping test method is described in section 5. The IDE support is explained in section 6. A case-study test element group (TEG) chip design and its performance evaluation are shown in section 7, and section 8 concludes this work.

2. Related Work

2.1 FPGA CAD Flow

The CAD tools of an eFPGA framework serve both the design and utilize flows of FPGA-IPs. Most of the tools used in both flows are the same, e.g., a typical FPGA CAD flow consists of logic synthesis (e.g., Yosys [17]), technology mapping (e.g., ABC [18]), clustering, placement, and routing phases. Technology mapping is responsible for converting the logic truth tables in a netlist file to ones that can be implemented by the logic cell (such as a LUT) used in the target FPGA. Clustering packs multiple logic cells into a logic cluster, the size of which corresponds to the number of logic cells contained in a logic tile of the FPGA. Placement is responsible for determining the location of each logic tile in the FPGA tile array. Clustering, placement, and routing are typically performed using the versatile place and route (VPR) tool [13]. The Verilog-to-Routing (VTR) project integrates these independently developed tools using a unified FPGA architecture description file to define the architecture parameters across the tool chain. Most eFPGA frameworks develop additional tools to automatically generate eFPGA HDL design code and user-circuit bitstream based on information exported from VTR. Sometimes these tools also generate Tcl scripts to assist in the backend design [6].

2.2 eFPGA Framework

The FPGA design flow needs to evaluate the performance of benchmark circuits implemented by different FPGA architectures to find the optimal FPGA architecture parameters that are consistent with the design goal. Traditional general-purpose FPGA architectures as used in most commercial FPGAs are designed with a benchmark circuit set that includes a wide range of application types, resulting in an FPGA architecture with the highest overall score but not necessarily the most optimized for any given application. After deciding all architectural parameters, a general-purpose FPGA is typically implemented by a full-custom flow to obtain the optimal layout for the target architecture at a cost of a design period of months to years. Once a function module is laid out in full-custom flow, it is difficult to make any modification, so the differences between commercial FPGAs of the same generation are only in the numbers and ratios of various functional modules (e.g., tiles, DSPs, and BRAM), but not in the finer-grained architecture parameters (e.g., LUT size, cluster size, channel width, and bit widths of DSPs and BRAM). On the other hand, the frontend and backend design of eFPGAs such as those in [5]–[12] can be fully automated, and synthesizable eFPGAs can be easily integrated into a variety of SoC designs, the period from architecture exploration to layout design being only a few days [6]. Therefore, the benchmark circuit set used for eFPGA architecture exploration can consist of only the application circuit of a certain domain required by the SoC designer, thereby resulting in an optimal FPGA architecture for that application domain. Of course, there is a non-negligible performance gap between FPGAs implemented by the full-custom flow and automated application-specific integrated circuit (ASIC) flow. Traditional eFPGA frameworks have been focused on generating an eFPGA that is similar to a certain commercial FPGA and then examining the performance gap between the two designs. However, these performance disadvantages of eFPGAs can be improved by domain-specific architectural optimizations, which is not discussed in existing work on eFPGA frameworks.

The eFPGA framework must also consider how to test the FPGA-IP that it generates. Traditional eFPGA frameworks provide application-circuit-based test methods [6]–[8], which load the user-circuit bitstream on the generated FPGA-IP for co-simulation. However, these tests focus mainly on the functional verification of the generated FPGA-IP rather than on the shipping test. Li and Wentzlaff [7] used a user-provided application testbench for testing; their flow supports behavioral simulation, post-synthesis behavioral simulation, and post-implementation behavioral simulation during the backend flow of the FPGA-IP. Mohan

et al. [5] provided an easy-to-use Python-based framework for FPGA-IP frontend verification, and Koch et al. [6] provided support for FPGA-IP verification based on commercial FPGAs, namely, implementing the eFPGA-framework-generated FPGA-IP on a commercial FPGA (implementing a virtual FPGA on a real FPGA) and loading the bitstream of the user circuit into this virtual FPGA for simulation. This has the advantages of speeding up the simulation of the target user circuit and enabling software development before the chip is available. However, all these eFPGA frameworks lack support for the shipping test, which is necessary for SoC designers to distinguish quickly between good and bad dies in the chip test phase.

Finally, traditional eFPGA frameworks provide complete toolchains for FPGA-IP design and utilization, but they all lack IDE support, which is critical for using these eFPGA frameworks efficiently. For chip end users who are developing circuits for the eFPGA, they need a visual tool environment that can edit HDL code, perform simulation, view waveforms, run circuit compilation, check implementation reports, and download bitstream to the device. Commercial FPGA vendors provide such IDEs for their FPGA users, such as Xilinx's Vivado [19] and Altera's Quartus [20]. However, this traditional form of IDE is not suitable for an eFPGA. First, SoC designers need to customize the tool chain and interface of the IDE according to their needs and integrate the FPGA design flow with the design flows of other SoC IPs. Second, cloud-usable electronic design automation tools have become a trend [21], so the IDE must support cloud deployment and be operable through a Web interface.

3. Overview of eFPGA Generation Suite

Fig. 1 shows an overview of the design flow of the proposed eFPGA generation suite. An FPGA-IP's architecture exploration, SoC implementation, and utilization are supported by three main flows: FPGA-IP architecture exploration and generation, chip implementation, and FPGA-IP utilization.

The FPGA-IP architecture exploration and generation flow explores the optimal FPGA architecture parameters for the application circuits provided by the SoC designer and then generates the Verilog of the optimal FPGA-IP and the bitstream for the shipping test. The SoC designer only needs to prepare domain-specific benchmark circuits and specify an architecture space for FPGA-IP exploration. The eFPGA Architecture Library contains all possible FPGA architectures supported by our framework. An FPGA architecture is determined by a combination of architectural parameters that define the SLM structure, cluster structure, routing structure, function blocks, and other modules, as well as parameters that define the tile array structure. The architecture exploration flow implements benchmark circuits on each architecture selected by the SoC designer and gives a performance report, then the SoC designer analyzes the performance report and finds the optimal FPGA-IP to be used based on the design goal (e.g., minimum area or shortest delay).

The CAD flow is explained as follows. First, the user circuit written in Verilog is synthesized with Yosys into a netlist in Blif format. Second, we perform technology mapping on the netlist. The Architecture Library passes the information of an SLM structure to ABC. To support SLM mapping, we added a Boolean matching function for the SLM to the FPGA mapping flow of ABC. In addition, we use an efficient function decomposition method to convert coarse-grained logics into fine-grained logics that can be implemented by the specified SLM. The details of the SLM will be introduced in section 4.1. Then, we use VPR for packing and placement. The Architecture Library provides the standard VPR FPGA architecture description file to VPR, which contains all FPGA architecture information. Finally, we developed our own routing tool—the EasyRouter [22]—for FPGA routing and FPGA-IP generation. Developed in C#, EasyRouter supports FPGA architectures defined by the VPR and can be easily modified to implement new FPGA architectures that are not supported by VPR. We integrate Verilog code generation for FPGA-IP, bitstream generation for user circuits, and shipping-test bitstream generation in EasyRouter. Rather than developing these tools as standalone applications, integrating these functions into the router makes it possible to obtain the required information directly from the routing resource graph and routing results, thereby reducing the development effort.

Chip implementation flow is based on a standard ASIC flow. The optimal FPGA-IP is integrated with the SoC design, and the chip backend design, fabrication, and chip shipping test are completed for a monolithic chip. The FPGA-IP code can be integrated with the SoC code and designed with a flatten design method, or the FPGA-IP and other SoC modules can be laid out separately and then integrated in the top-level layout of the SoC. After the design is completed, the SoC layout is tapeouted to the foundry for fabrication. Before a chip is shipped, we load the shipping-test bitstream generated with the FPGA-IP into the chip under test and analyze the output to check whether the chip is a good die. The specific test method will be explained in detail in section 5.

When the SoC chip product arrives at the user, the latter uses the FPGA-IP utilization flow to develop the application circuit and then loads the user-circuit bitstream into the FPGA module of the SoC to implement the circuit. Basically, the user-circuit design flow uses the same CAD tools as the FPGA-IP architecture exploration and generation flow, and all tools process the user circuit based on the optimal FPGA-

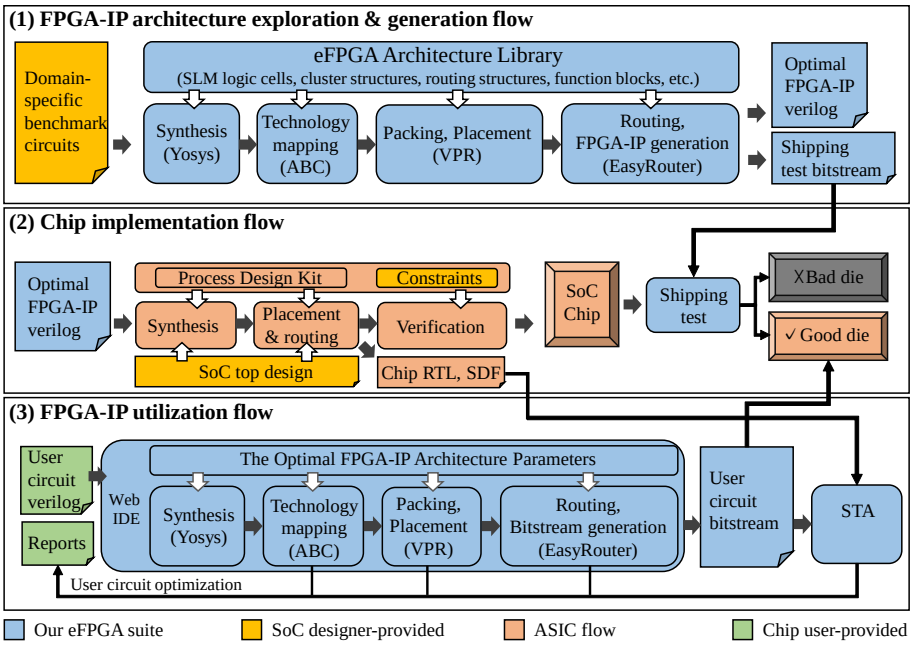


Fig. 1 Overview of design flow of proposed eFPGA suite.

IP architecture parameters. At this point, EasyRouter generates a bitstream of the user circuit after routing, then the bitstream is loaded into the FPGA to complete the deployment of the circuit. We use OpenSTA [23] to perform accurate timing analysis of the routed user-circuit bitstream with the SoC chip's register transfer level code and standard delay format information extracted from the layout. Each CAD tool will return design reports to the user, who will know whether the compilation of the circuit is successful and whether the circuit design needs improvement based on these reports. Our FPGA-IP utilization flow provides a complete Web interface to users. The details of the Web IDE will be introduced in section 6.

4. FPGA-IP Architecture

This section describes the architecture design of the FPGA-IP generation. We introduce the proposed SLM logic cell and then describe the FPGA architecture in detail.

4.1 SLM Logic Cell

Most modern FPGAs use a LUT as their basic logic cell. The circuit size of the LUT increases as $O(2^k)$ with an increasing number of inputs, k, so LUTs with more than six inputs negatively affect the overall FPGA performance. There are some logic cells derived from the subset of frequently used logics [24]–[26]. These logic cells implement only partial logics to optimize circuit area. However, the logic appearance frequency of different types of circuits changes, and it is impossible to use one function-reduced logic cell to express all circuits efficiently. In this work, we propose a novel SLM as the logic cell for FPGAs. The SLM shrinks the logic cell by also reducing logic express capability, but a SoC designer can explore the optimal SLM from a set of SLM structures for the target application to find an efficient domain-specific SLM logic cell for the FPGA-IP.

Table 1 Logic appearance rank (five inputs).

Rank	MCNC	VPR
1	A·B·C·D·E	$A{\cdot}\alpha{+}A{\cdot}B{+}B{\cdot}\alpha$
2	A·(B+C)·(D+E)	A⊕B⊕α
3	A·B·C·(D+E)	A⊕B⊕C⊕D⊕E
4	A·E+B·C·D	A·B+(A+B)·(C·D·E+(C⊕D⊕E))
5	A·(B+C+D+E)	A⊕B⊕C⊕D+$\bar{A}$·C·D·E+A·$\bar{C}$·$\bar{D}$·$\bar{E}$
6	A·B·(C+D·E)	A⊕B⊕C⊕(D·E)
7	A·(B+C+D·E)	A·($\bar{C}$+$\bar{D}$+$\bar{E}$)+B·C·D·E
8	A·(B·($\bar{D}$+$\bar{E}$)+C·D·E)	A⊕((B·E)⊕(C·D))
9	A·B·(C+D+E)	A·B·C+(A+B·C)·(D⊕E)
10	A·(B+C·(D+E))	A·(B+C)·(D+E)

*α=C·D+D·E+C·E

4.1.1 Logic Appearance Frequency

We investigated the frequency of logics based on the 20 largest MCNC benchmarks [27] and 13 VPR benchmarks [28]. Circuits in these two benchmarks are very different in terms of functionality and scale. We mapped the benchmark circuits into five-input LUTs with ABC and then analyzed the technology mapping results. We then classified and analyzed the

mapping results following the NPN-equivalence class method [29], as in our previous work [26]. The NPN-equivalence class of a function consists of functions that can be derived by performing input negation, input permutation, or output negation on the original function.

Table 1 lists the top 10 most frequently appearing logics of the two benchmarks. The labels A, B, C, D, E indicate the inputs of the five-input logics. Each logic in Table 1 represents a single NPN-equivalence class. We find that the most frequent logics in the MCNC benchmarks are simple AND-OR operations. However, VPR benchmark circuits use more logics, such as majority voter and XOR operations. Therefore, different circuits may have different high-frequency logics, so an application-specific logic cell made from one benchmark set may be inefficient for other application domains.

We then analyzed frequently appearing logics by performing the Shannon expansion to find common characteristics of these functions. Shannon expansion is one of the most basic logic conversion methods and is defined as

$$\begin{aligned} &f(x_0, x_1, \cdots, x_{k-1}, x_k, x_{k+1}, \cdots, x_{n-1}) = \\ &\quad x_k \cdot f(x_0, x_1, \cdots, x_{k-1}, 1, x_{k+1}, \cdots, x_{n-1}) \\ &\quad + \overline{x_k} \cdot f(x_0, x_1, \cdots, x_{k-1}, 0, x_{k+1}, \cdots, x_{n-1}). \end{aligned} \quad (1)$$

The two derived functions are called partial functions of the original function f. If we perform the Shannon expansion on input argument x_k of function f, it follows that

$$f = \overline{x_k} \cdot f_{(x_k=0)} + x_k \cdot f_{(x_k=1)}. \quad (2)$$

In addition, similar to NPN-equivalence theory, partial functions $f_{(x_k=0)}$ and $f_{(x_k=1)}$ can be interconverted by negating or fixing the inputs or output. For example, a five-input logic function is

$$\begin{aligned} f &= A \cdot (C \cdot D + D \cdot E + C \cdot E) + A \cdot B \\ &\quad + B \cdot (C \cdot D + D \cdot E + C \cdot E). \end{aligned} \quad (3)$$

If we perform the Shannon expansion on argument A, partial functions $f_{(A=0)}$ and $f_{(A=1)}$ become

$$\begin{aligned} f_{(A=0)} &= B \cdot (C \cdot D + D \cdot E + C \cdot E), \\ f_{(A=1)} &= B + C \cdot D + D \cdot E + C \cdot E. \end{aligned} \quad (4)$$

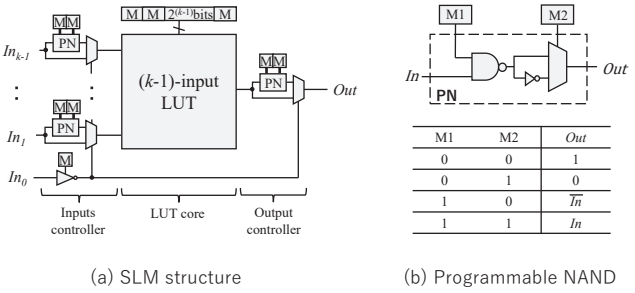
We then negate arguments $B,\ C,\ D,\ E$ and the output of $f_{A=0}$, obtaining

$$\begin{aligned} &\overline{f_{(A=0,B=\overline{B},C=\overline{C},D=\overline{D},E=\overline{E})}} \\ &= \overline{\overline{B} \cdot (\overline{C} \cdot \overline{D} + \overline{D} \cdot \overline{E} + \overline{C} \cdot \overline{E})} \\ &= B + (C + D) \cdot (D + E) \cdot (C + E) \\ &= f_{(A=1)}. \end{aligned} \quad (5)$$

In this example, the number of inputs of a small LUT core is $k-1 = 4$ because $f_{(A=0)}$ has four inputs, and the number of negated inputs is $w = 4$ because arguments $B,\ C,\ D,\ E$ are inverted.

Table 2 Ratio of interconvertible partial functions.

	Item	5-LUT	6-LUT	7-LUT	8-LUT
MCNC	Ratio of k	57%	53%	45%	37%
	Coverage of k	98.34%	99.46%	99.53%	91.33%
VPR	Ratio of k	32%	28%	32%	25%
	Coverage of k	97.31%	87.18%	95.08%	81.46%



(a) SLM structure (b) Programmable NAND

Fig. 2 SLM logic cell structure.

Table 2 gives the ratio of interconvertible partial functions, where "Ratio of k" means the ratio of k input logic functions to all logic functions, and "Coverage of k" means the percentage of interconverted k input logic functions to all k input logic functions. For example, for five-input LUT mapping for MCNC, the ratio of five-input and less than five-input logic functions is 57% and 43%, respectively. 98.34% of five-input logic functions can be interconverted. The results show that partial functions of most logics in the two benchmark sets can be interconverted.

4.1.2 SLM Structure

We can obtain two partial functions by performing the Shannon expansion on any logic, and one of the partial functions will have a large potential for conversion to the other by only negating or fixing its I/Os while retaining core functionality. This means that we can reduce the configuration memory of a conventional LUT while still expressing the same logic. So, we can propose an SLM that can implement logic functions with smaller area and less configuration memory than those of a LUT.

Fig. 2(a) shows the general structure of the SLM, which has three main parts: an input controller, a LUT core, and an output controller. For a partial function with interconvertible logic, the LUT core implements the shared core function of its partial functions, and the input and output controllers switch between partial functions. Fig. 2(b) shows the circuit and truth table of the programmable NAND. This circuit can negate inputs or fix output to 0 or 1 with two configuration memory cells.

When designing the input controller, it is unneces-

Table 3 Available logic cells in Architecture Library.

Logic Cell	# of Mem.	Area[μm^2]	Delay[ps]
4-LUT	17	198.00	327.75
5-LUT	33	370.59	349.34
5-SLM(4,0)	20	235.87	473.00
5-SLM(4,1)	22	261.27	632.45
5-SLM(4,2)	24	286.68	654.56
5-SLM(4,3)	26	312.08	667.65
5-SLM(4,4)	28	337.48	682.53
6-LUT	65	725.99	371.89
6-SLM(5,0)	36	408.47	484.68
6-SLM(5,1)	38	433.87	644.85
6-SLM(5,2)	40	459.27	665.68
6-SLM(5,3)	42	484.67	678.79
6-SLM(5,4)	44	510.07	692.32
6-SLM(5,5)	46	535.47	706.92

sary to make all inputs controllable for most partial function conversions. This allows tradeoffs between SLM logic coverage and area to find the most area-efficient SLM by reducing the number of input controllers. The general k-SLM structure has k inputs (w of which are controllable, with $w \leq k-1$), one output, and a $(k-1)$-LUT core, which is designated as k-SLM($k-1$, w). For some logics, we need to convert $f_{(x=0)}$ to $f_{(x=1)}$, whereas other logics require the reverse conversion. The output controller is programmable for these two cases.

Table 3 gives the number of configuration memories, area, and delay comparisons of LUTs and SLMs with four to six inputs. LUTs are listed for comparison. The area and delay are derived by synthesizing these logic cells with the USJC 55nm process. We can see that all SLMs have a smaller memory usage and area than LUTs with the same number of inputs. However, the delay of SLMs is longer because of the additional delay of I/O controllers. The logic cells listed in Table 3 are all available in our Architecture Library, and we can compare their performances when implementing the target application to find an optimal cell for the FPGA-IP.

4.2 FPGA-IP Architecture

Fig. 3 shows the (a) logic block, (b) tile, and (c) FPGA top designs of the proposed FPGA-IP. The yellow-colored labels indicate architecture parameters that can be optimized for domain-specific application circuits. The details of each part are explained as follows.

4.2.1 Logic Block

A basic logic element (BLE) consists of an SLM with a scan FF and implements the most fine-grained logic and supports both combination and sequential modes: when a BLE works in combination mode, the SLM output is output directly; when a BLE works in sequential mode, the SLM output is first stored in the FF and then output from there at the next clock cycle. The architecture parameter for the BLE is the SLM structure. The parameter K indicates the number of inputs of the SLM.

Then N BLEs are grouped into a logic cluster, where N indicates the cluster size. The BLEs within a logic cluster can be connected with low-latency local connections, thereby reducing the burden of global routing resources. A local connection block (LCB) is used to forward outside inputs to BLE inputs, as well as feedbacks of BLE outputs to other BLE inputs within the same cluster. The logic cluster and the LCB form a logic block (LB), as shown in Fig. 3(a). The optimal *SLM structure* and the value of *cluster size* can be found for the target-application domain during architecture exploration. To save the global routing resources, the number of inputs of an LB, I, is set as less than the number of inputs of all BLEs ($K \times N$), and I is calculated as [30]

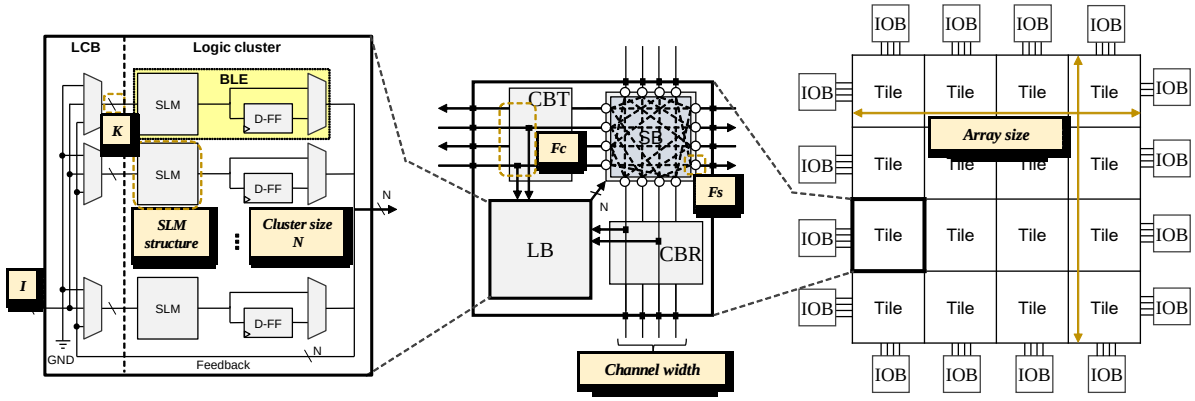
$$I = \frac{K}{2} \times (N+1). \tag{6}$$

4.2.2 Tile

As Fig. 3(b) shows, a tile consists of an LB and global routing resources, the latter being used to connect the fine-grained logics distributed in many LBs to implement the user circuit. The global routing resources include two routing channels, two connection blocks (CBs) named CBT and CBR, and a switch box (SB). Each routing channel contains a group of routing wires. The parameter *channel width* indicates the quantity of wires within a routing channel; each wire has only one direction for signal transfer. The CBs are responsible for connecting the routing channels to the LB . The parameter Fc indicates the ratio of wires connected to the LB. In the case of Fig. 3(b), we have *channel width* $= 4$ and $Fc = 0.5$, so two out of four wires in a routing channel are connected to the LB. The SB is a switch of global routing channels, and we adopt the Wilton-type SB [31] whose structure is shown in Fig. 3(b). The Wilton-type SB has $Fs = 3$, which means that any output multiplexer of the SB can switch its value from three other input directions. In our design, all LB outputs are connected to global routing through SBs, so an SB output multiplexer has $3 + N$ inputs. The optimal Fc and *channel width* can be found for the target-application domain during architecture exploration.

4.2.3 FPGA Top

As Fig. 3(c) shows, by repeating the logic tiles on the chip in an array style, an FPGA top design is constructed. In our design, all the logic tiles have a homogeneous structure to facilitate testing [32]. The optimal *array size* can be found for the target-application domain during architecture exploration.



(a) LB (b) Tile (c) FPGA Top

Fig. 3 FPGA-IP architecture.

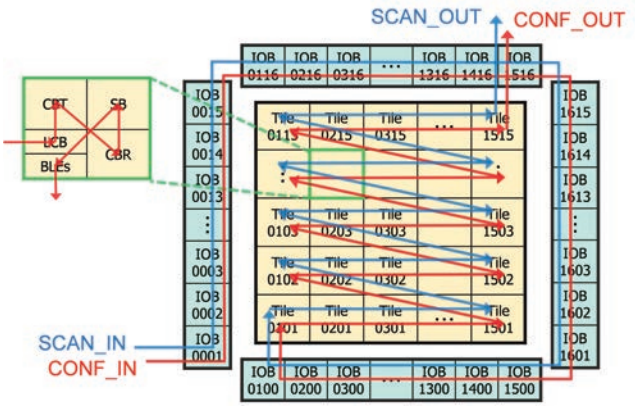
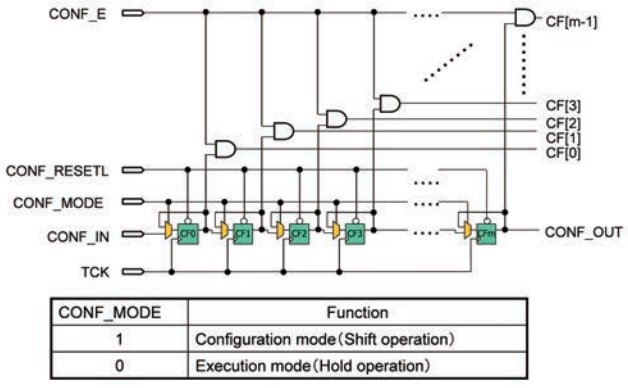


Fig. 4 Configuration chain and scan chain.



CONF_MODE	Function
1	Configuration mode (Shift operation)
0	Execution mode (Hold operation)

Fig. 5 Configuration circuit.

On the outer sides of the tile array, there are $4 \times arraysize$ I/O blocks (IOBs). An IOB is responsible for transferring a signal between a routing channel and an I/O cell outside the FPGA-IP. Normally, FPGA I/O cells for user pins are bidirectional, but the routing wires are unidirectional, so an IOB can be configured as an input or an output and to map a selected wire in a routing channel to the I/O.

4.2.4 Configuration Chain and Scan Chain

In our design, all configuration FFs (CFs) are connected to form a configuration chain, which is shown by the red arrows in Fig. 4. Fig. 5 shows the circuit of the configuration chain. Along this path, a single FF stores one configuration datum. A bitstream of an FPGA consists of all configuration bits arranged in the same order as that of the configuration chain. During configuration, we first disable the CONF_E signal so that all configuration bits in the FPGA receive a stable 0. We then enable the CONF_MODE signal to allow CFs to read values from their inputs. Then the bitstream is input from the CONF_IN port. The configuration is performed by shifting the bitstream according to the clock TCK. When all configuration bits are shifted to their CFs, we disable CONF_MODE and enable CONF_E to transfer the FPGA status from the configuration mode to the user mode.

In addition, there is a scan chain to read out the values of all FFs of BLEs. The scan path is shown by the blue arrow in Fig. 4. A scan chain consists of a series of scan FFs that store the results of the user circuit or chip tests.

4.2.5 Controller

Besides the FPGA, we implement a controller to manage the FPGA-IP status, i.e., configuration mode, user mode, and test mode. One of the main functions of the controller is to decode and execute the bitstream. Fig. 6 shows an example of the format of a bitstream with 104,960 configuration bits. A bitstream consists of multiple 288-bit-long frames. Each frame carries 256-bit configuration bits and 32 cyclic redundancy check error check bits. At the final frame, if there are fewer than 256 configuration bits, the rest are filled as dummy bits. The first eight bits of a bitstream are the in-

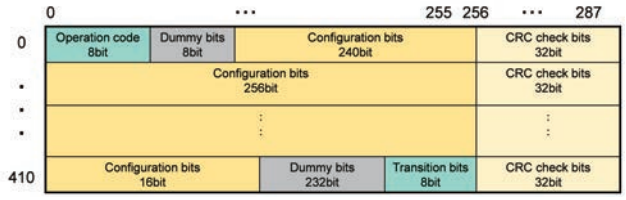


Fig. 6 Bitstream format.

struction code that sets up the mode of the FPGA; the transition bits of the final frame are the transition code that specifies the next state to transfer to after this bitstream.

5. Shipping Test

In general LSI, the automatic test pattern generator (ATPG) tool is used for testing. However, in FPGAs, the test pattern cannot be generated by the ATPG tool because the user circuit is not uniquely determined. In addition, the routing part of an FPGA involves a huge number of switch combinations, which makes it difficult to implement the test circuit. In our eFPGA suite, we adopt a previously developed FPGA test method in reference [16] for stuck-at faults in logics and routing resources.

We choose long line for device delay evaluation of eFPGAs. Here, long line is defined as a path that traverses the FPGA from top to bottom, left to right, and right to left. In discrete FPGAs, a longline is often defined as a path that traverses FPGA without switches, but note that here it passes through the routing part. The evaluation of the long line delay will be discussed in section 7.6.2.

For the test of an SLM shown in Fig. 2(a), we use test patterns with different configurations and input patterns to cover all paths of an SLM, then we examine the SLM outputs of these tests by observing its attached FFs in the same BLE. We test the LUT core, PN circuits, and the programmable inverter of an SLM to achieve a 100% SLM test.

To test the LUT core of an SLM, we first configure all the configuration memories of the LUT to be 0 or 1, then all the paths taken by the MUXes of the LUT can be activated by testing all input patterns of the LUT. For the PN circuits of the SLM, we configure all the configuration memories of the LUTs to 0 or 1, and we test all the PN patterns shown in Fig. 2(b), then we can test for 0/1 stuck-at faults in each path of the PN circuits. For the test of the programmable inverter on the In_0 input path, the output PN circuit is configured as inverted output, then the input In_0 works as a switch of the SLM to be either the LUT output or the inverted LUT output.

6. Integrated Development Environment

We have developed a user-friendly graphical IDE for users to develop user circuits for eFPGA more efficiently, as shown in Fig. 7. Our IDE provides main circuit-development support for the FPGA-IP utilization flow shown in Fig. 1(3) such as Verilog code edit, simulation, waveform review, circuit compile, reports check, and bitstream generation to users entirely through the Web interface.

We used the NODE-RED programming platform to develop the main interface of the eFPGA IDE. Fig. 8 shows the complete module structure of the IDE dashboard, whose main modules are project setup, source code, simulation, pin assign, and compile report. Project setup is used mainly to set project parameters and create and clean up the project directory. Project parameters include project name, project path, FPGA architecture, and the path to the FPGA CAD toolchain execution file. In the source-code part, the designer provides the user circuit Verilog files and resource files in the source-code nodes, which are written automatically to the project directory when the save button is pressed. The pin-assign part implements a Web-based CSV data-table editing interface that allows the user to easily configure the mapping between the circuit ports and the FPGA pins. We implement the execution commands of each CAD tool in a separate function node and then connect these CAD tool nodes to form an execution flow. Execution logs of each tool are saved in the project directory, and the report analysis code in the report part extracts the information of most interest to the designer, such as the resource utilization and critical path delay, and displays them in the Web IDE (NODE-RED Dashboard) invoked from IDE Dev. We also integrated a Web version of VS Code (code-server) to allow users to easily browse the project directory and edit files. The simulation result can be verified the circuit behavior on GTKWave of wave viewer via VNC remote desktop. The wave viewer also can be invoked from Web IDE.

Using NODE-RED to develop the IDE dashboard offers the following advantages. First, the graphical programming style of NODE-RED makes defining complex IDE functional modules with complex relationships—especially the FPGA CAD tool chain—very intuitive and clear. As we can see from the compiler part of Fig. 8, the implementation form of the CAD flow in NODE-RED is basically the same as the design description form of FPGA-IP utilization flow shown in Fig. 1(3). Therefore, users can not only understand and use the IDE easily but also easily modify and customize it according to their needs. Second, the NODE-RED flow shown in Fig. 8 can be saved in a single JSON file, which greatly simplifies the storage, sharing, and version control of user circuit projects.

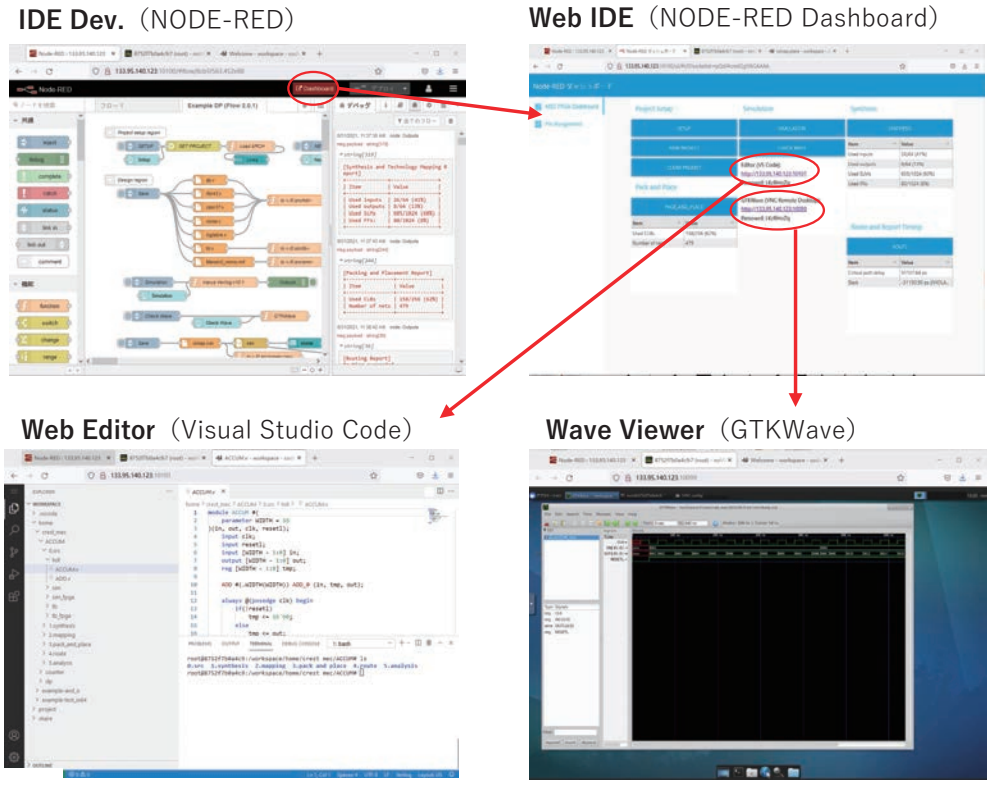


Fig. 7 Main screens of IDE container.

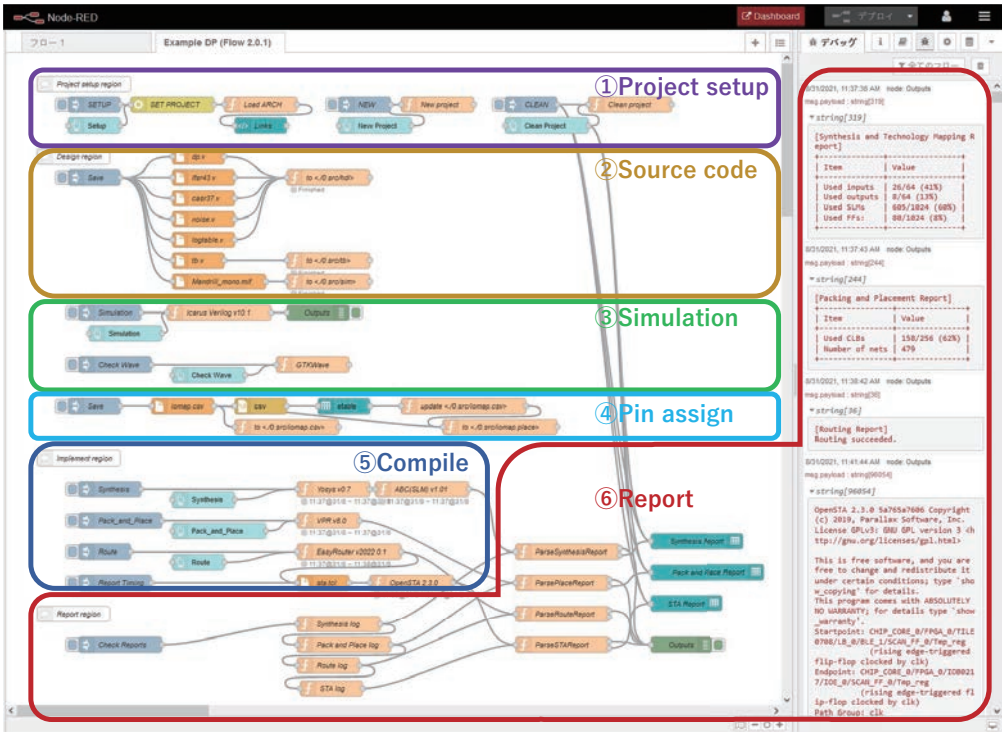


Fig. 8 NODE-RED-based CAD flow implementation.

Compared with the complex project organization of traditional FPGA IDEs, our implementation can reproduce the whole project through only one JSON file. Third, NODE-RED itself is a Web-based development and execution platform, so the IDE developed based on it is naturally usable through a Web browser. Finally, NODE-RED is widely used in Internet of Things (IoT) applications, which makes our IDE more friendly to IoT developers. For example, the eFPGA design flow shown in Fig. 8 can be integrated into the user's IoT flow in the form of NODE-RED subflow, thereby enabling online bitstream generation and configuration. In addition, we package all the IDE parts into a standalone docker image, thereby making it easily deployable to a local or cloud environment for use.

7. Domain-specific FPGA-IP Case Study

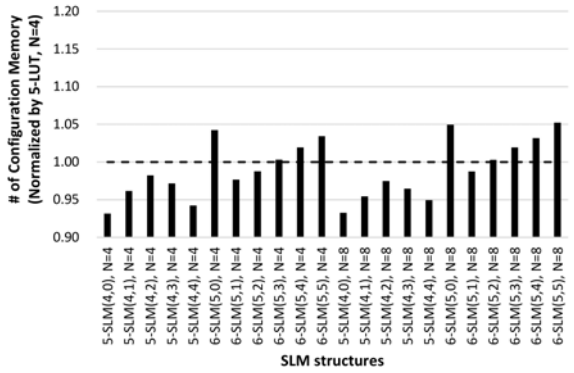
In this section, we show the FPGA-IP architecture exploration, chip design, shipping test, and performance evaluation for an edge-oriented DP application circuit as a case study.

7.1 Target Application

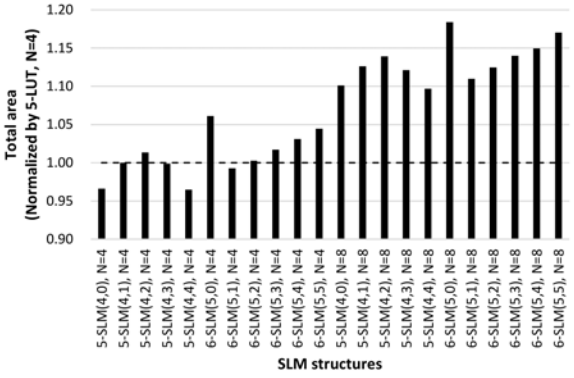
DP is an algorithm that protects the privacy of individuals during data collection and analysis. Because low-end microprocessors on edge devices usually cannot perform as required for DP computation, we must implement hardware acceleration to improve performance. In addition, the DP circuit requires the algorithm to be upgraded for security enhancement at runtime, so it is suitable to take the implementation approach of embedding an eFPGA into the SoC and then implementing the DP circuit on the eFPGA, which can provide sufficient performance to perform real-time DP processing of data collected by the edge device. Our target DP circuit has a data width of eight bits. The DP circuit processes input data with random values, so that the exact value of the input cannot be identified from the output data, but the statistical characteristics of the original data are kept for analysis.

7.2 Architecture Exploration

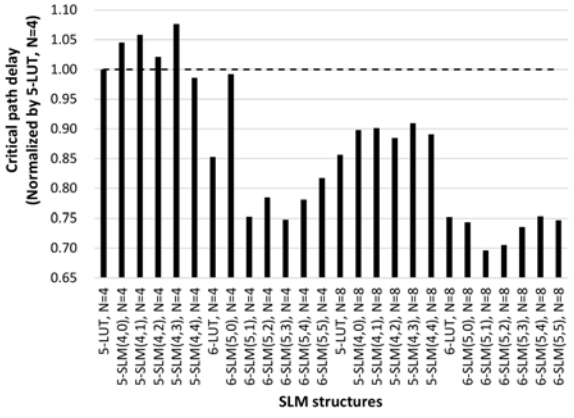
We performed the FPGA-IP architecture exploration for the target DP circuit based on the flow shown in Fig. 1(1). The architecture parameters to be determined during exploration are shown as the yellow-colored labels in Fig. 3. First, we chose all SLM structures with five or six inputs as logic-cell candidates, and we evaluated cluster sizes of four and eight. The exploration results are shown in Fig. 9. From (a) and (b), we can see that the combination of 5-SLM(4,0) with a cluster size of four has the smallest configuration memory usage and total area. For the critical path delay, (c) shows that 6-SLM(5,1) with a cluster size of four



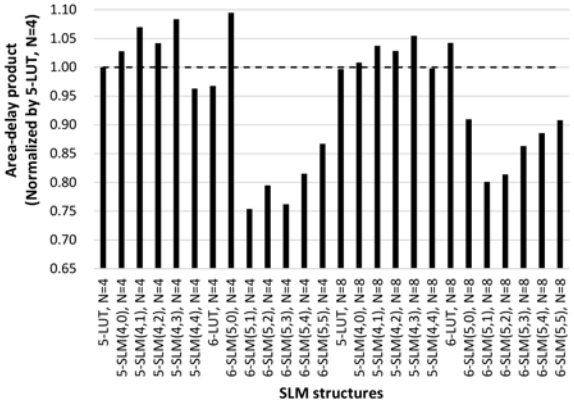
(a) # of configuration memory.



(b) Total area.



(c) Critical path delay.



(d) Area-delay product.

Fig. 9 SLM structure exploration results.

Table 4 TEG chip specifications.

Item	Value
Array size	16 × 16
Channel width	30
Logic cell	5-SLM(4,0)
Logic clusters	4
LB inputs	12
SB structure	Wilton (Fs = 3)
CB structure	normal (Fc = 0.25)
I/O pins	64
Conf. mem. (FPGA array)	104,960bit
Bitstream size	118,368bit
Die size	$4.41mm^2(2.1mm \times 2.1mm)$
Process	USJC $55nm$ LVT
Core voltage	0.9V
I/O voltage	1.8/2.5/3.3V

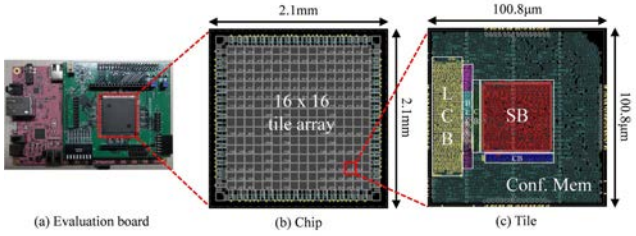


Fig. 10 Layout of TEG chip.

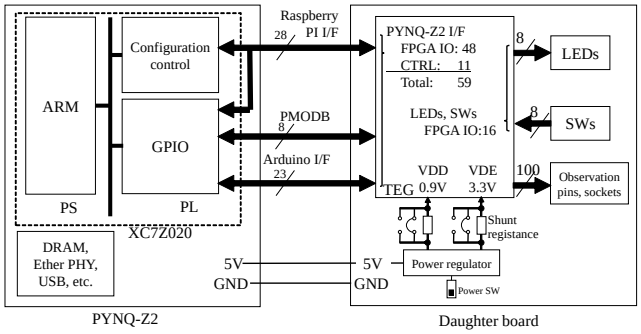


Fig. 11 Block Diagram of Evaluation Board.

Table 5 Test cycle results.

Item	Value
Normal operation[cycles]	2
Scan operation[cycles]	1,087
Other control operations[cycles]	13,410
Test cycles per test pattern[cycles]	119,459
Number of test patterns	128
Total test time[cycles]	15,290,752

performs the best. On the metric of the area–delay product, 6-SLM(5,1) with a cluster size of four has the best overall performance. So we can see that the choice of SLM structure and cluster size will differ according to the design target. In this work, we chose the smallest 5-SLM(4,0) with a cluster size of four to minimize the cost of silicon area on chip.

Based on the logic exploration results, other parameters are then optimized for the DP circuit, and the specifications of the FPGA-IP are given in Table 4. The channel width was set to 30, which is the minimum channel width for a router to successfully complete the DP circuit routing. In this work, we set the array size to 16×16 to fully fill the given $4.41mm^2$ chip area. In a SoC design, we can set the minimum array size to provide the number of tiles required by the target circuit to save the chip area usage. The DP circuit implementation uses 158 out of a total of 256 CLBs (62%). For the bitstream, 104,960 bits are used as FPGA configuration memory; there are additional bits for instruction and error handling, so the total bitstream size is 118,368 bits.

7.3 Chip Design

When the architecture parameters are fixed, we obtain the generated optimal FPGA-IP design from the end of the flow shown in Fig. 1(1). We designed and fabricated the TEG chip using a 55nm ASIC process. Fig. 10 shows the (a) evaluation board, (b) chip top floorplanning, and (c) tile layout. The tile size on Fig. 10(c) is 100.8 μm□. We manually planned the positions of the macros of BLEs, LCB, CBs, and SB and then let configuration memories fill the remaining spaces. The SB is placed at the center to minimize the delay variation of different routing paths. From the chip on Fig. 10(b), we can see that the main tile array occupies most of the area.

7.4 Evaluation board

We have developed an evaluation board. The block diagram is shown in Fig. 11. It is designed as a daughter board of PYNQ-Z2 by Tul using Zynq-7000 SoC by Xilinx so that the control signal and the FPGA I/O of the chip can be flexibly controlled. In PYNQ-Z2, 59 I/Os are connected to the PL (Programmable Logic) part of the Zynq device. By building a control circuit in the PL part according to the purpose of evaluation, the chip can be easily controlled by the PYNQ environment from the ARM processor in the PS (Processing System) part. A total of 59 signals, including 11 chip control signals and 48 out of 64 FPGA I/Os, can be connected to the PL part. The remaining 16 FPGA I/Os are connected to switches (SWs) and LEDs on the daughter board. The I/O voltage is set to 3.3V to interface with the PYNQ-Z2 and core voltage is set to 0.9V for typical operation.

7.5 Shipping Test

To confirm the stuck-at fault testing coverage, we input test patterns to the FPGA and observed the toggle coverage using Synopsys VCS Q-2020.03-SP1. To test all hardware resources, we prepared a total of 128 test patterns, each of which was tested by repeating "configuration operation → normal operation → scan operation." The test results were first stored in the FFs in

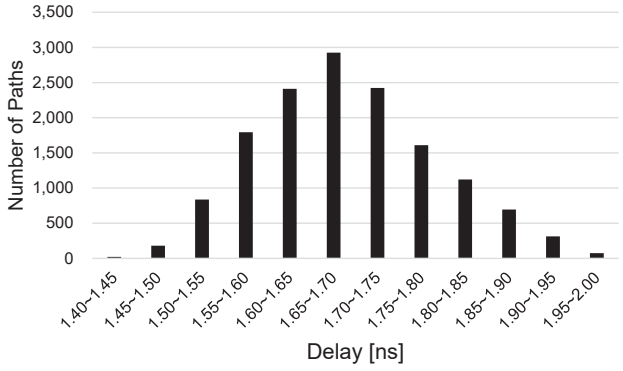


Fig. 12 Path delay distribution from tile inputs to tile outputs.

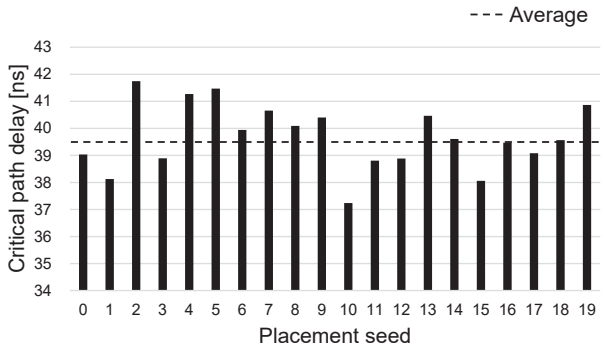


Fig. 13 Distribution of critical path delay for DP circuit implementation (20 initial seeds).

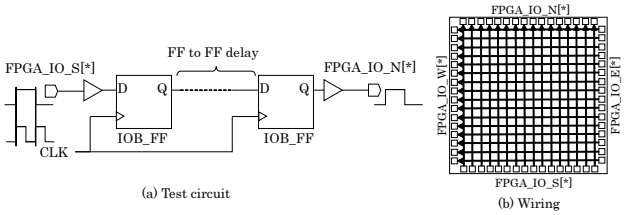


Fig. 14 Test circuit for long line delay.

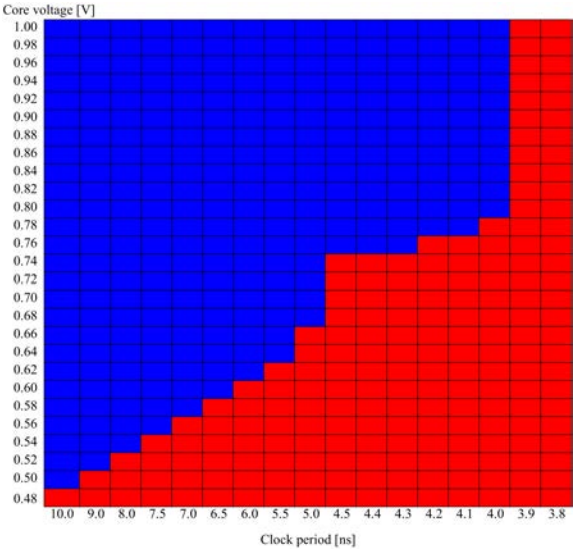


Fig. 15 Shmoo plot of long line delay.

the BLEs and IOBs and then scanned out for analysis.

The test-cycle evaluation results are given in Table 5. In total, 15,290,752 test cycles were required to perform the test with all 128 test patterns. The total test time with a 100-MHz clock was calculated to be 0.153 s. This indicates that our test method can complete the test in a short time for shipping-test requirements. In addition, we observed 100% toggle coverage in the entire FPGA-IP using the proposed method, which means that 100% stuck-at faults can be detected during the shipping test.

7.6 Results and Discussion

7.6.1 Delay evaluation by STA

In this evaluation, we performed static timing analysis using Synopsys PrimeTime P-2019.03-SP3 under the typical conditions, operating voltage of 0.9V and temperature of 25°C. First, we got the delay values of each path from the primary inputs to the primary outputs in the tiles of the FPGA. In the tested chip, there are 14,400 paths from input to output when the channel width is 30, there are four LB clusters, and the input and output have four directions. The DP circuit was implemented using the design flow described in Fig. 1(3).

Fig. 12 shows the path delay distribution from tile inputs to tile outputs. The delays are distributed in the range of 1.43–1.98 ns, with an average value of 1.69 ns. It can be seen that the delay variation is within about 6% of the mean value. The delay distribution of the target DP circuit is shown in Fig. 13. To examine the delay variation of the initial seed of placement, the initial seed was varied from 0 to 19 during placement, and the variation of the critical path delay was checked. In this case, the delays are distributed in the range of 37.23–41.75 ns, with an average value of 39.68 ns. It can be seen that the delay variation is kept within 3%. So the DP circuit can operate at a maximum frequency of 26.8 MHz, and 1 byte of data can be processed at each clock cycle, so the throughput is 27.8MOPS (Mega Operations Per Second). When Python executes DP processing, it is achieved about 2,215 OPS (Operations Per Second). The DP circuit is 12.55×10^3 times faster than Python.

7.6.2 Long line delay

To evaluate long line delay on the chip, we use the test circuit shown in Fig. 14. To create a long line, connect the Q output of IOB-FF located on one side to the D

input of IOB-FF located on the opposite side.

Firstly, STA results are shown in Table 6. There are 16 paths from one side to the opposite side, and each path have a different propagation delay. All path delays are in the range 3.586–4.389ns. In particular, the path from west side to east side is routed with a short delay.

Then, we observe the propagation status of the one-shot pulse when the frequency and core voltage are changed, and a shmoo plot is created from the results. Fig. 15 is one of the shmoo plot results which indicates operating status from south side to north side path. The shmoo plot shows that the blue area indicates correctly operation, on the other hand the red area indicates not. We confirm the correct behavior up to 250MHz (4.0ns) on the actual chip, however STA results show that maximum delay is 4.389ns, minimum delay is 4.179ns, and average is 4.266ns. It is confirmed that the operation is slightly faster than the result of STA.

Table 6 STA results of long line delay

Path	Min.[ns]	Ave.[ns]	Max.[ns]
North to South	3.860	3.969	4.091
South to North	4.179	4.266	4.389
East to West	4.092	4.197	4.292
West to East	3.586	3.802	3.943

7.6.3 Validation of an application behavior

To validate the application behavior, we implement two applications, simple 24-bit binary counter and LDP (Local Differential Privacy) circuits. The implementation result is shown in Table 7. Both applications are correctly operated on the chip.

24-bit binary counter is implemented on the actual chip as a simple application. STA result shows that the generated FPGA-IP can achieve a maximum frequency of 176.7 MHz (5.66ns). However, actual operating frequency is observed 117MHz (8.55ns). And also, propagation delay of clock-to-pad is measured about 12ns. Although the inside of the chip can operate at high speed, it should be noted that the delay of the I/O buffer is large. The power consumption is measured about 162mW under the condition of operating frequency of 117MHz and core voltage of 0.90V.

Next, we also implement an LDP (Local Differential Privacy) circuit to modify the interface of DP circuit used in section 7.1. To connect host processor on the evaluation board, the LDP has streaming interface by AXI stream so that many data is modified with random values using DMA (Direct Memory Access) transfer. And it is modified internal functions to reduce calculation error, so the circuit is larger than the DP circuit in section 7.1. Resource usages of BLE and FF are 97.0% and 7.6%, respectively, so the LDP is a large application for the actual chip. Critical path delay is 57.3ns (17.4MHz) by STA. The delay is larger than the DP circuit in section 7.1. It is a reason that The LDP circuit has long critical path from random generator to final adder for adding a noise. However operating frequency can be improved using pipelining by optimizing. Finally, we measured power consumption of 139mW at 10MHz operating frequency.

Table 7 Implementation result of application

Resource usage	24-bit counter	LDP
BLE (SLM)	39/1,024 (3.8%)	993/1,024 (97.0%)
LB	10/ 256 (3.9%)	250/ 256 (97.6%)
FF	24/1,024 (23.4%)	102/1,024 (10.0%)

Item	24-bit counter	LDP
Operating frequency [MHz]	117	10
Measured Power [mW]	162	139

7.6.4 Discussion

The design target of our eFPGA suite is to generate an application-specific FPGA-IP, which is different with commercial FPGAs and other eFPGA frameworks, so it is difficult to give a direct performance comparison between our work and existing FPGAs. One of the advantages of this work is the reduction of configuration memory and are of logic cells by replacing LUTs with SLMs. From the implementation results of the DP application-specific FPGA-IP case study, we can say that the generated FPGA-IP of this work can provide sufficient performance within an acceptable area for SoC designers of IoT application domains.

8. Conclusion

In this study, we proposed an eFPGA generation suite with customizable architecture, testability, and IDE and that covers all the eFPGA design generation, testing, and utilization stages. For the eFPGA design generation, our IP generation flow can explore the optimal logic cell, routing, and array structures for a given target application. For the testability, we introduced a novel shipping-test method that can detect 100% of the stuck-at faults in the entire FPGA-IP. In addition, we proposed a user-friendly and customizable Web-based IDE framework for the generated eFPGA based on NODE-RED. In the case study, we used a real DP circuit as the target application and showed the entire FPGA-IP architecture exploration, chip design, shipping test, and performance evaluation using the proposed eFPGA generation suite. The TEG chip evaluation showed that the generated FPGA-IP had small area for embedding into SoC designs and sufficient performance for edge devices.

We will extend to our design framework to be able to design an eFPGA including of hard macro such as

DSP, Block RAM, and so on as a future work.

Acknowledgments

This work was partially supported by JST CREST Grant JPMJCR19K1, Japan. This work was also supported through the activities of VDEC, The University of Tokyo, in collaboration with NIHON SYNOPSYS G.K.

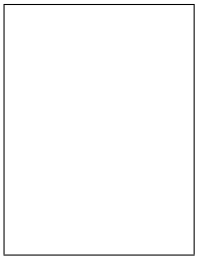
References

[1] Xilinx, Zynq-7000 SoC Overview, 2018. Retrieved Oct. 14, 2021 from https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.
[2] Achronix, Speedcore eFPGA Product Brief (PB028), 2020. Retrieved Oct. 14, 2021 from https://www.achronix.com/sites/ default/files/docs/Speedcore_eFPGA_Product_Brief_PB028.pdf.
[3] QuickLogic, ArcticPro eFPGA, 2021. Retrieved Oct. 14, 2021 from https://www.quicklogic.com/products/efpga/arcticpro-2/.
[4] FlexLogix, EFLX®eFPGA, 2021. Retrieved Oct. 14, 2021 from https://flex-logix.com/efpga/.
[5] P. Mohan, O. Atli, O. Kibar, M. Zackriya V, L. Pileggi and K. Mai, "Top-down Physical Design of Soft Embedded FPGA Fabrics," Proc. ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays (FPGA'21), pp.1–10, 2021. https://doi.org/10.1145/3431920.3439297
[6] D. Koch, N. Dao, B. Healy, J. Yu and A. Attwood, "FABulous: An Embedded FPGA Framework," Proc. ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays (FPGA'21), pp.45–56, 2021. https://doi.org/10.1145/3431920.3439302
[7] A. Li and D. Wentzlaff, "PRGA: An Open-Source FPGA Research and Prototyping Framework," Proc. ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays (FPGA'21), pp.127–137, 2021. https://doi.org/10.1145/3431920.3439294
[8] X. Tang, E. Giacomin, B. Chauviere, A. Alacchi and P-E. Gaillardon, "OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs," Proc. IEEE Micro, vol.40, no.4, pp.41–48, July/Aug. 2020. https://doi.org/10.1109/MM.2020.2995854
[9] J. H. Kim and J.H. Anderson, " Synthesizable FPGA fabrics targetable by the Verilog-to-Routing (VTR) CAD flow," Proc. 25th Int'l Conf. on Field Programmable Logic and Applications (FPL). pp.1–8, 2015. https://doi.org/10.1109/FPL.2015.7293955
[10] S.J.E. Wilton, C-H Ho, B. Quinton, P. H.W. Leong, and W. Luk, "A Synthesizable Datapath-Oriented Embedded FPGA Fabric for Silicon Debug Applications," ACM Trans. Reconfigurable Technol. Syst., 1, 1, Article 7 (March 2008), 25 pages, 2008. https://doi.org/10.1145/1331897.1331903
[11] N. Kafari, K. Bozman, S.J.E. Wilton, "Architectures and algorithms for synthesizable embedded programmable logic cores," Proc. ACM/SIGDA 11th Int'l Symp. on Field programmable gate arrays (FPGA'03). pp.3–11, 2003. https://doi.org/10.1145/611817.611820
[12] Andy Yan and S.J.E. Wilton, "Sequential synthesizable embedded programmable logic cores for system-on-chip," Proc. IEEE 2004 Custom Integrated Circuits Conf. (IEEE Cat. No.04CH37571), pp.435-438, Orland, Oct. 2004. https://doi.org/10.1109/CICC.2004.1358844
[13] J. Rose, J. Luu, C.W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: architecture and CAD for FPGAs from verilog to routing," Proc. ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays (FPGA'12), pp.77–86, 2012. https://doi.org/10.1145/2145694.2145708
[14] Y. Nakazato, M. Amagasaki, Q. Zhao, M. Iida, and M. Kuga, "Automation of Domain-specific FPGA-IP Generation and Test," Proc. 11th Int'l Symp. on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'21). Article 4, pp.1–6, 2021. https://doi.org/10.1145/3468044.3468048
[15] M. Amagasaki, R. Araki, M. Iida and T.Sueyoshi, "SLM: A Scalable Logic Module Architecture with Less Configuration Memory," IEICE Trans. on Fundamentals, vol.E99-A, no.12, pp.2500–2506, Dec. 2016. (LETTER) http://dx.doi.org/10.1587/transfun.E99.A.2500
[16] M. Amagasaki, Y. Nishitani, K. Inoue, M. Iida, M. Kuga and T. Sueyoshi, "Physical Fault Detection and Recovery Methods for System-LSI Loaded FPGA-IP Core," IEICE Trans. on Inf. & Syst., vol.E100-D, no.4, pp.633–644, Apr. 2017. https://doi.org/10.1587/transinf.2016AWI0005
[17] C. Wolf, Yosys Open SYnthesis Suite, Retrieved Oct. 14, 2021 from http://www.clifford.at/yosys/.
[18] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Retrieved Oct. 14, 2021 from http://www.eecs.berkeley.edu/ alanmi/abc/.
[19] Xilinx, Vivado® ML, Retrieved Oct. 14, 2021 from https://www.xilinx.com/products/design-tools/vivado.html.
[20] Intel, Quartus® Prime, Retrieved Oct. 14, 2021 from https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.
[21] S. Obilisetty, "Chip Design and Cloud: the Good, the Emerging, and the Potential," Design Automation Conf. (DAC'21). Session 58, Dec. 2021.
[22] Q. Zhao, K. Inoue, M. Amagasaki, M. Iida, M. Kuga and T. Sueyoshi, "FPGA Design Framework Combined with Commercial VLSI CAD," IEICE Trans. on Inf. & Syst., vol.E96-D, no.8, pp.1602–1612, Aug. 2013. https://doi.org/10.1587/transinf.E96.D.1602
[23] OpenSTA, Retrieved Oct. 14, 2021 from http://opensta.org/.
[24] J. H. Anderson and Q. Wang, "Area-efficient FPGA Logic Elements: Architecture and Synthesis," Proc. 16th Asia and South Pacific Design Automation Conf. (ASP-DAC'11), pp.369–375, 2011. https://doi.org/10.1109/ASPDAC.2011.5722215
[25] Z. Zilic and Z. G. Vranesic, "Using Decision Diagrams to Design ULMs for FPGAs," IEEE Trans. on Comput. vol.47, no.9, pp.971–982, Sep. 1998. https://doi.org/10.1109/12.713316
[26] M. Iida, M. Amagasaki, Y. Okamoto, Q. Zhao and T. Sueyoshi, "COGRE: A Configuration Memory Reduced Reconfigurable Logic Cell Architecture for Area Minimization," IEICE Trans. on Inf. & Syst., vol.E95.D, no.2, pp.294–302, Feb. 2012. https://doi.org/10.1587/transinf.E95.D.294
[27] K. S. McElvain, IWLS'93 Benchmark Set: Version 4.0, Distributed as part of the MCNC Int'l Workshop on Logic Synthesis '93 benchmark distribution, 1993.
[28] V. Betz, 175.vpr SPEC CPU2000 Benchmark Description File, Retrieved Oct. 14, 2021. from http://www.spec.org/osg/cpu2000/CINT2000/175.vpr/docs/175.vpr.html.
[29] D. Chai and A. Kuehlmann, "Building a better Boolean matcher and symmetry detector," Proc. Conf. Design, automation and test in Europe (DATE'06), pp.1079–1084, 2006. https://doi.org/10.1109/DATE.2006.243959
[30] E. Ahmed and J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density" IEEE Trans. Very Large Scale Integr.

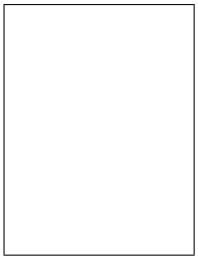
(VLSI) Syst., vol.12, no.3, pp.288–298, Mar. 2004. https://doi.org/10.1109/TVLSI.2004.824300

[31] G. G. Lemieux and D. M. Lewis, "Analytical framework for switch block design," Proc. of 12th Int'l Conf. on Field-Programmable Logic and Applications (FPL'02), pp.122–131, 2002. https://doi.org/10.1007/3-540-46117-5_14

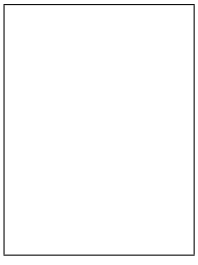
[32] K. Inoue, M. Koga, M.i Amagasaki, M. Iida, Y. Ichida, M. Saji, J. Iida and T. Sueyoshi, "An easily testable routing architecture and prototype chip," IEICE Trans. on Inf. & Syst., vol.E95-D, no.2, pp.303–313, Feb. 2012. https://doi.org/10.1587/transinf.E95.D.303



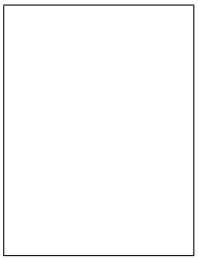
Morihiro Kuga received his B.E. degree in Electronics Engineering from Fukuoka University in 1987. Further, he received his M.E. and D.E. degrees in Information Systems from Kyushu University in 1989 and 1992, respectively. From 1992 to 1998, he was a lecturer at Kyushu Institute of Technology. He has been an associate professor at Kumamoto University since 1998. He has been belonging to the Faculty of Advanced Science and Technology since 2016. His research interests include parallel processing, computer architecture, reconfigurable system, and VLSI system design. He is a member of IEICE and IPSJ.



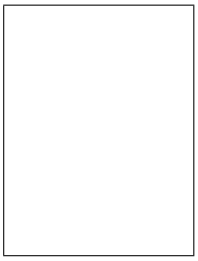
Qian Zhao received his B.E. degree in the College of Automation and Electronic Engineering from Qingdao University of Science and Technology, China, in 2007. Further, he received his M.E. and D.E. degrees in Computer Science and Electrical Engineering from Kumamoto University in 2011 and 2014, respectively. He was a postdoctoral researcher at Kumamoto University from 2014 to 2018. He was an assistant professor in the Faculty of Computer Science and Systems Engineering at Kyushu Institute of Technology from 2018 to 2022. He is currently an engineer of ByteDance, Beijing, China. His research interests include architecture, CAD, and design method of reconfigurable computing systems. He is a member of IEICE and IEEE.



Yuya Nakazato received his B.E. and M.E degrees in computer science and electrical engineering from Kumamoto University in 2020 and 2022. He is currentry an engineer of Fanuc Corporation.



Motoki Amagasaki received the B.E. and M.E. degrees in Control Engineering and Science from Kyushu Institute of Technology, Japan in 2000, 2002, respectively. He was a engineer at NEC Micro Systems Co., Ltd. from 2002 to 2005. He received the D.E. degree from Kumamoto University, Japan, in 2007. He has been a professor in the Faculty of Advanced Science and Technology at Kumamoto University since 2021. His research interests include Machine learning, Reconfigurable system and VLSI design. He is a member of IEICE, IPSJ, JSAI and IEEE.



Masahiro Iida received his B.E. degree in Electronic Engineering from Tokyo Denki University in 1988. He was a research engineer at Mitsubishi Electric Engineering Co., Ltd. from 1988 to 2003. He received his M.E. degree in Computer Science from the Kyushu Institute of Technology in 1997. He received his D.E. degree from Kumamoto University, Japan, in 2002. He was an associate professor at Kumamoto University until 2015. From 2002 to 2005, he held an additional position as a researcher at PRESTO, Japan Science and Technology Corporation (JST). He has been a professor with the Faculty of Advanced Science and Technology at Kumamoto University since January 2016. His current research interests include high-performance, low-power computer architectures, reconfigurable computing systems, and VLSI devices and design methodologies. He is a senior member of IPSJ and IEICE, and a member of IEEE.