International Journal of Computing and Digital Systems
2025, VOL. 17, NO. 1, 1–10

http://dx.doi.org/10.12785/ijcds/1571050022

Instruction-Level Customization and Automatic Generation of Embedded Systems Cores for FPGA

Omar Yehia[1,2], Sandra Raafat[1], M. Watheq El-kharashi[1], Ayman Wahba[1] and Cherif Salama[1,3]

[1]*Computer and Systems Engineering Department,Ain Shams University, Cairo, Egypt*
[2]*Siemens DISW, Cairo, Egypt*
[3]*Computer Science and Engineering Department, The American University in Cairo, Cairo, Egypt*

Received 14 July 2024, Revised 1 October 2024, Accepted 3 October 2024

Abstract: Reducing power consumption and improving performance are crucial requirements for many applications, especially power hungry and time-consuming applications. This is particularly true when these applications are running in power or time-constrained environments like battery-operated embedded systems or on Internet of Things (IoT) devices. A general-purpose processor is not promising for this kind of application as it cannot provide optimized performance and power consumption for specific applications. That is why domain-specific architectures (DSA) are gaining popularity, as they promise optimized performance for these types of applications in terms of throughput, power consumption, and overall cost. Unfortunately, the use of DSA presents inherent limitations as it requires custom design for each group of applications and cannot offer optimized performance for each specific application. This paper explains how to take advantage of the open standard Instruction Set Architecture (ISA) of the fifth generation of Reduced Instruction Set Computers (RISC-V) to automate the generation of a uni-processor core customized for a certain application such that the processor supports only the very specific instructions needed by this application. The proposed generator is capable of producing the Register Transfer Level (RTL) description of a processor core for any desired application given its source code. This work targets Field Programmable Gate Arrays (FPGAs) due to their reconfigurability. When compared with general-purpose processors, the conducted experiments show that application-specific cores generated by our approach managed to achieve energy and execution time reductions reaching 8% and 5% respectively on some of the used benchmarks. The proposed methodology also offers the added flexibility stemming from the possibility to automatically re-configure the FPGA when a new or upgraded software application that would benefit from modifying the set of supported instructions is deployed.

Keywords: Power Consumption, Performance, Embedded Systems, Instruction-Level Customization, RISC-V, Rocket Chip, FPGA, Core-generation

1. INTRODUCTION

The ever increasing demand for various specific applications presents some implementation challenges [1]; one of the most challenging applications is those encountered to support low-power and high-performance applications. That is why processor customization has witnessed a rise in interest. Researchers have explored innovative approaches to achieve this customization; one promising way is the use of Application Specific Architectures (ASAs) [2], [3]. ASIPs are used in many new technologies, including 5G networks, the Internet of Things (IoT), and machine learning. These processors are designed as general purpose chips but are specially modified to suit specific applications or tasks [4], [5].

Traditional methods of processor customization like Domain Specific Architectures (DSAs), architectures that utilize the hardware more efficiently for a certain group of similar applications, have proven to be time-consuming and costly, requiring extensive expertise in hardware design, computer architecture, and system-level design. Reducing power consumption and improving overall performance can be achieved using DSAs; however, DSAs need to be implemented from scratch, and each application domain needs its own design, which requires resources such as cost and time. Furthermore, DSAs do not offer application-specific optimizations [1], [6], [7].

The primary need for the automatic generation of processor cores is its ability to produce optimized designs efficiently by reducing designers' time and effort, allowing them to focus on higher-level design tasks such as system-level integration and software development. The open standard Instruction Set Architecture (ISA) of the fifth generation of Reduced Instruction Set Computer (RISC-V) is currently the best option for customizable blocks and is often used to create blocks designed for specific applications [8], [9]. This paper introduces a fully automated

E-mail address: omar.yehia@siemens.com, sandra_raafat@eng.asu.edu.eg, watheq.elkharashi@eng.asu.edu.eg, ayman.wahba@eng.asu.edu.eg, cherif.salama@aucegypt.edu

ASA core generator built on top of RISC-V [10], [11]. This generator aims to organize the Field Programmable Gate Array (FPGA) core generation process for specific applications, requiring only the application's source code as input. The result is improved performance and reduced power consumption. The methodology requires establishing the research environment, experimenting with a general-purpose RISC-V core, designing and developing the ASA RISC-V generator, integrating it and conducting experiments.

In this paper, we propose the customization of a RISC-V core for a specific application by generating a core that supports only the needed instructions for this application. RISC-V is a free open source ISA standard that can be used in commercial and open source work. All RISC-V implementations must support a base collection of instructions in addition to any number of optional groups of instructions known as extensions [12], [13]. It is possible to customize a RISC-V core to include an entire extension or to exclude it [14]. In this paper, we propose taking this approach one step further by increasing the granularity of the customization to the instruction level and by automating the full process.

The main contributions of this research are:

- Proposing instruction-level customization of RISC-V cores to only support instructions needed by the target application
- Proposing a methodology to automate this customization requiring only one input which is the source code of the target application
- Implementing the proposed methodology and integrating it with the standard Rocket Chip RISC-V core generator
- Assessing the effectiveness of instruction-level customization by comparing it to general-purpose RISC-V and to extension-level customization in terms of power consumption and execution speed.

This research focuses on using FPGAs to implement automatically generated RISC-V processor cores. Its main significance lies in how it can enhance embedded systems and IoT devices by making them more efficient in terms of power usage, improving their overall performance, and giving them the ability to be reconfigured when needed.

Our approach is designed around three key aspects:

- Reducing power consumption: One of our main goals is to minimize how much power the system uses, especially in devices that run on batteries. This will help extend battery life and make these devices more energy-efficient.
- Enabling FPGA reconfigurability: We introduce a method that allows the FPGA to be reconfigured, meaning the system can be adapted to new tasks or updates without needing to replace the hardware. This flexibility makes the system more adaptable to future changes.
- Improving performance: We also focus on optimizing the performance of the system, ensuring that it can operate faster and more efficiently, which is important for many embedded and IoT applications.

The remainder of this paper is structured as follows: Section 2 provides some background on the RISC-V ISA and some of the tools we used. Section 3 presents the proposed methodology, while Section 4 lists and discusses the experimental results obtained showing the superiority of instruction-level customized cores over general-purpose and extension-level customized ones. Section 5 discusses the limitations of our work which is concluded in Section 6 along with a discussion of possible future work and extensions.

2. Background

This section provides a brief overview of the RISC-V ISA since our main methodology is highly dependent on customizing it at the instruction level. It also describes some of the standard tools used to design and test RISC-V systems. These tools were extensively used in this research, and readers need to be at least familiar with them.

A. *RISC-V ISA*

RISC-V is an open standard ISA, It is the fifth generation RISC ISA proposed by the Berkley research lab at the University of California, Berkeley. In addition to being free to use even for commercial applications, RISC-V has many attractive advantages, including its elegant uniform instruction format, its modularity, and its extensibility, among others [10], [11].

The RISC-V architecture includes a base ISA that supports standard base integer instructions in addition to optional standard extensions that improve the architecture's functionality to meet specific application requirements. The base integer instruction set appears as the abbreviation (I) when labeling any RISC-V implementation. For example, an implementation labeled RV32I would indicate a 32-bit processor that supports only the base instructions (exactly 40 instructions), while an implementation labeled RV64I would indicate a similar processor with native support for 64-bit integers and arithmetic operations (adding 15 instructions to the RV32I ISA). Given our interest in embedded systems and IoT devices, this paper focuses only on 32-bit implementations.

The most important optional standard extensions are:

- The Integer Multiply/Divide (M) extension, which introduces instructions for efficient integer multi-

plication and division operations. This extension is handy in applications that require complex arithmetic computations.

- The Atomic (A) extension, which offers atomic memory operations. These operations ensure data integrity and synchronization in multithreaded environments where multiple threads access shared memory concurrently.
- The single-precision floating-point (F) extension introduces instructions for performing arithmetic operations on single-precision floating-point numbers. This extension is beneficial for applications that involve computations with real numbers, such as scientific simulations or signal processing.
- The double-precision floating point (D) extension introduces instructions for performing arithmetic operations on double-precision floating point numbers. This extension is useful for applications that require high-precision computations on real numbers.

Accordingly, a RISC-V implementation labeled RV32IF would indicate a 32-bit processor supporting the base integer instructions in addition to the optional single-precision floating-point instructions. A processor supporting all the above extensions would be labeled with a G for brevity. So an RV32IMAFD would simply be labeled as RV32G indicating that it encompasses all of these extensions; integer instructions (I), multiply/divide instructions (M), atomic instructions (A), single-precision floating-point instructions, and single-precision floating-point instructions (D).

All RISC-V instructions are encoded in 32 bits; however, optional support for a compressed 16-bit version of some of these instructions is possible and is indicated by adding a (C) to the implementation name. Compressed instructions contribute to reducing the static code size of applications, which can be useful in environments with limited memory capacity. An RV32IC implementation supports the base integer instructions and the compressed encoding of some of them as defined in the RISC-V specifications.

Designers can optimize processors for applications that require specific features by including or excluding each of the different extensions at design time.

B. Tools

The field of processor core generation for RISC-V architectures has seen significant advancements in recent years, driven by the need for efficient and customized processors. Researchers have explored various techniques and frameworks to automate the generation of RISC-V processor cores tailored to specific applications. In this subsection, we explore the most relevant tools we relied on in our research.

1) *Chipyard*

Chipyard is an open-source purpose-built framework to design and explore full-system hardware and software stacks based on the RISC-V ISA. Provides a comprehensive set of tools, libraries, and infrastructure that enable researchers, developers, and designers to build and customize their RISC-V-based chips and systems. Chipyard offers a wide array of open source digital IP blocks that can be easily combined and expanded. Incorporating multiple simulation and implementation tools enables higher-quality development, ensuring verification, validation, and system integration [15].

One of the critical components of Chipyard is the Rocket Chip Generator, which generates RTL descriptions of RISC-V cores that can be customized to suit individual needs. This feature allows users to design their processor cores to meet their specific requirements.

In addition to hardware design aspects, Chipyard also includes support for software development. It provides a complete software stack, including bootloaders, device drivers, operating systems, and development tools, allowing users to develop and run software on their custom RISC-V-based systems.

Chipyard is a versatile platform for generating instruction-based (RTL) designs through the Rocket Chip generator. Facilitates benchmark simulation using the Verilator simulator while supporting FPGA prototyping by providing constraint files and RTLs for FPGA simulation [15].

2) *Rocket Chip*

Rocket Chip is an open-source SoC generator capable of producing many design instances consisting of synthesizable RTL; Rocket Chip enables easy customization for specific applications by changing a single configuration file. Also, it provides a modularity feature as its component libraries are independent repositories. It is a collection of customizable processor parts that enable quick development and customization of anything from small embedded processors to complex multi-core systems. The openness of the Rocket Chip Generator has played a key role in its popularity in research [16].

The Rocket Chip generator constructed the RISC-V platform, which utilizes various configurable chip-building libraries to create numerous SoC variants. The Rocket Chip generator comprises several subcomponents: Core, Cache, Tile, Tile-Link, and Peripherals.

Rocket Chip includes many parts of the SoC in addition to the CPU. By default, Rocket Chip uses Rocket core, which generates an in-order RISC-V CPU. Additionally, it can be configured to use the Berkeley Out-of-Order Machine (BOOM) core generator [17] or another custom CPU generator.

3) Rocket Core

Rocket core is an in-order 5-stage pipeline processor core generator. It is used as a component within the Rocket Chip SoC generator.

The Rocket core is written in the Chisel hardware description language. It supports open-source RV32GC and has a Memory Management Unit (MMU) that supports page-based virtual memory, a non-blocking data cache, and a front-end with branch prediction. Several parameters are exposed, including the optional support of some ISA extensions (M, A, F, D), the number of floating-point pipeline stages, and the cache and Translation Lookaside Buffer (TLB) sizes.

The rocket core consists of the following stages: Program Counter (PC), Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB) in addition to the floating-point pipeline stages, as shown in Figure 1.

4) RISC-V Test Suite

The RISC-V Test Suite focuses primarily on evaluating the functionality and compliance of RISC-V processors. It includes a variety of individual test programs and benchmarks designed to assess the power consumption, utilization, and performance of instruction-level optimization. Twelve different benchmarks were used in this research. Each benchmark tests a specific aspect; for example, some benchmarks exclusively target variables with integer data types, while others work on data floating-point data types. The following list enumerates the twelve benchmarks we used with a brief description for each.

1) **median**: It calculates the median value from a large data set. It tests the processor's ability to perform mathematical operations and manipulate data efficiently.
2) **dhrystone**: A well-known benchmark. It measures the performance of a processor by executing a list of instructions involving integer arithmetic, string operations, and logical operations [18].
3) **mm**: This benchmark is written in both C and assembly language. It uses multi-threading in the matrix multiplication operation.
4) **mt-matmul**: It is a multi-threaded version of the Matrix Multiplication benchmark. It measures the performance of a processor's multi-threaded matrix multiplication operations.
5) **mt-vvadd**: It is a multi-threaded version of the Vvadd benchmark. It assesses the performance of a processor in multi-threaded vector addition operations.
6) **multiply**: It evaluates the processor's performance in multiplication operations. It measures the execution time for multiplying two matrices.
7) **pmp**: It is a security feature in hardware-based found in some computer architectures, including RISC-V. It enables the partitioning and isolating of memory spaces, protecting specific physical memory regions from unauthorized access. It tests the functionality and performance of physical memory protection mechanisms implemented in the processor.
8) **qsort**: It evaluates the performance of a processor's sorting algorithm. It measures the time to sort a given array of elements using the Quick Sort algorithm.
9) **rsort**: Another sorting benchmark. It assesses the processor's performance by measuring the time to sort a given array of elements using the Radix Sort algorithm.
10) **spmv**: The name of this benchmark stands for Sparse Matrix-Vector Multiplication. It tests a processor's performance in multiplication operations involving sparse matrices and vectors.
11) **towers**: It measures the processor's execution speed for recursive operations by solving the Tower of Hanoi puzzle.
12) **vvadd**: It tests the performance of a processor's vector addition capabilities. It measures the time taken to perform the element-wise addition of two vectors.

In the next section, we explore how we used and extended the aforementioned tools to implement the proposed instruction-level custom RISC-v core generator.

3. Methodology

As highlighted earlier, the goal of this research is to automate the process of generating a RISC-V core customized to support a single specific software application given its source code. To achieve this goal, we need to automate the generation of an RTL description of a RISC-V core that supports only the specific instructions needed by this application. This proposed instruction-level customization is expected to create the simplest possible core implementation that would support the application of interest, leading to potential performance and power consumption gains, as demonstrated in Section 4.

The process of achieving our goal is depicted in Figure 2. The input to the entire process is the source code of the application that we would like to run. The output is the RTL description of the application-specific instruction-level custom RISC-V core. The process requires going through three main steps implemented by three main components, namely, a RISC-V compiler, a unique instruction Extractor, and a RISC-V Generator. The following subsections are dedicated to explaining each of these three components.

A. Compilation

The first step in the process is to compile the desired application using the RISC-V toolchain compiler (gcc). For this step, we use the `riscv64-unknown-elf-gcc` command. The input is the application source code, and the output is the corresponding assembly code. The RISC-V toolchain compiler can be configured via command-line

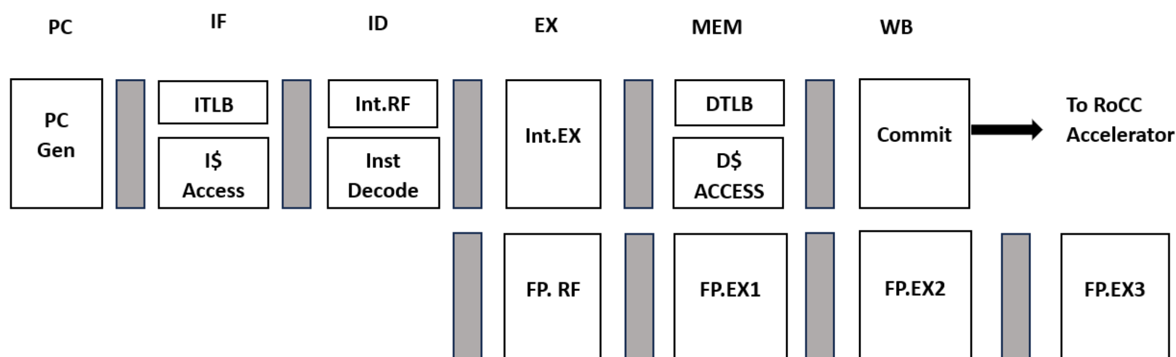

Figure 1. Rocket Core Pipeline. Reproduced from [17].

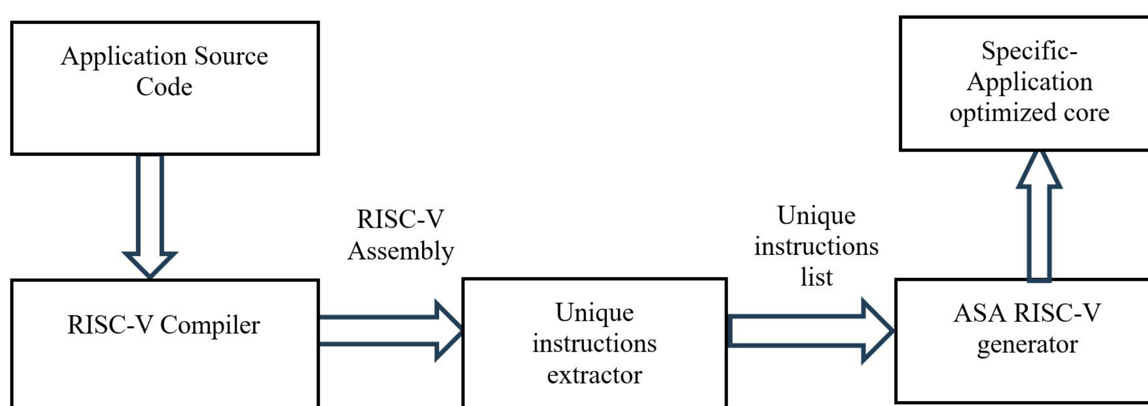

Figure 2. Application Specific Core Generation Process

arguments to target various implementations of the RISC-V processor such as RV32I, RV32IMA, RV32G, RV32GC, etc. Since we are still intending to customize the core to support only the needed instructions at a later stage, at this stage, we want to allow the compiler to use any instruction that it Sees fit. However, we exclude compressed instructions, since these instructions do not provide additional functionality and will require additional hardware to be decoded. Therefore, in this stage, we always configure the compiler to target the RV32G architecture by using the command line argument "-march rv32g" with gcc. Another important aspect of the compilation step is that it can be done with various optimization levels such as -O0, -O1, -O2, -O3, and -Os. Different optimization levels will produce different assembly programs in some cases, avoiding the use of certain instructions or certain constructs. We experiment with different optimization levels to see their effect on the overall performance of the resulting RISC-V code.

B. *Unique Instructions Extraction*

The second step in the process is to analyze the assembly program resulting from the compilation step by extracting a list of unique instructions used in that program. Depending on the input program and on the optimization level used in the compilation program, we will end up with a different mixture of instructions that need to be supported by the underlying core. We wrote a simple Python script to accomplish this task. The script takes the assembly program as input and produces a list of all unique instructions, which is then saved into a JSON file.

C. *RTL Generation*

The third step in the process is the generation of the RTL description of the RISC-V core that supports only the instructions generated by the second step. The input is the JSON file that contains the list of unique instructions, and the output is a synthesizable RTL description of the desired core. Instead of building this ASA RISC-V generator from scratch, we extended the open source Rocket Chip generator to support a finer granularity of customization. So in this step, the extracted instructions are forwarded to our ASA RISC-V generator built upon the Rocket Chip generator, whose configuration files are written in Chisel.

Chisel is an open source Hardware Description Language (HDL) that facilitates circuit generation for ASIC and FPGA; Chisel extends the capabilities of the Scala programming language by introducing hardware construction primitives. By empowering designers with the capabilities of a modern programming language, Chisel enables them

to create parameterizable circuit generators that generate synthesized Verilog code.

The main configuration Scala file of the Rocket chip generator allows it to generate either:

1) A general-purpose RISC-V core: This is the default RTL description of a general-purpose RISC-V core (RV32G), or
2) An extension-level customized RISC-V core: This is done by enabling or disabling certain extensions by setting parameter values in the configuration file. For example, if a user wants to generate a core suitable to execute an application that does not utilize the multiplier, nor the Floating Point Unit (FPU) extensions, both (M) and (F) extensions can be disabled in the configuration file. This feature allows the Rocket Chip generator to generate the following six combinations of RTLs to be generated: RV32I, RV32IM, RV32IF, RV32IA, RV32IMF, RV32IMA, and RV32IAF.

The Rocket Chip generator does not support instruction-level customization. So, it either includes support for the full extension (a collection of related instructions) or not, but it cannot include support for individual instructions from an extension or even from the base instruction set while ignoring the rest of the instructions in the set. Here are the main modifications to the Rocket Chip generator that we implemented in order to support this instruction-level customization that we are proposing:

- In the Rocket Chip configuration files, some updates have been made to incorporate instruction parameters from a JSON file; a spray-json library is used in the files to provide an easy way to work with JSON and make these files read their parameters from the JSON files extracted from the unique extractor. Five specific configuration files have been modified: ALU scala file, Atomic scala file, FPU scala file, Multiplier scala file, and Main Configurations scala file.

- In each of these files, a new class has been introduced. This class takes the JSON file as input and assigns "true" to the instruction parameter if enabled or "false" if disabled. These instruction parameters control the main class, allowing it to disable the hardware components associated with the corresponding instructions based on their enabled or disabled status.

- The necessary verilog constructs supporting needed instructions are generated, while unused constructs are eliminated. For example, if an application only requires multiplication functionality from the Multiplier Extension, instructions such as division and remainder will be entirely disabled despite being part of the same extension. This level of optimization allows for the customization of instructions based on the precise needs of the application, ensuring that unnecessary instructions are disabled to optimize performance and resource utilization.

Further improvements are still required to achieve more optimization in instruction-level customization since some limitations in the current implementation still need to be addressed as explained in Section 5.

4. Results

To evaluate the usefulness of the proposed instruction-level customization, we needed to run several experiments that assessed the effect of such customization on the execution time and the power consumption of various applications. The benchmarks compiled were simulated on the generated RISC-V cores using the cycle-accurate Verilator simulator that captured the number of cycles required for their execution. This information is crucial for calculating the execution time and evaluating the performance of the chip design. The design utilization was calculated using Xilinx Vivado assuming a Zedboard Zynq-7000 FPGA board which we used in our experiments. Vivado was used for synthesis, mapping, placing, and routing of the generated cores and to produce post-implementation utilization and power consumption reports.

This section is split into subsections that list our experimental results. The first subsection deals with extension-level customization, while the second subsection is concerned with instruction-level customization.

A. Extension-Level Customization Results

Benchmark testing is performed on all RISC-V extensions to assess their performance and power consumption. This process involves establishing a baseline and reference point for comparison. The reference benchmark used is the general purpose RISC-V (RV32G) extension.

In Table I, the utilization information for various extension-level customizations is compared to that of the RV32G extension shown in the first row. As expected, the RV32G extension has the highest number of Lookup Tables (LUTs), Multiplexers (MUXes), D Flip flops and digital signal processors (DSPs). The remaining extensions follow in the following order: RV32IMF, RV32IAF, RV32IF, RV32IMA, RV32IM, RV32IA, and RV32I, which are also expected since the FPU followed by the multiplication/division unit is expected to be the largest components.

Table I also presents the minimum clock period utilized in the RTL files generated for the general-purpose RISC-V core and various extension combinations. The table shows that only the RV32I core has an advantage in this regard.

Since the power consumption of an FPGA is bound to increase with its utilization, these results demonstrate that generating RTL designs that support only the required extensions for each application has the potential to enhance performance and power consumption. These findings motivate the work in this paper, which mainly proposes taking

TABLE I. All Extensions Utilization Information

RTL	Number of LUTs	Number of D Flip flops	F7 Muxes	F8 Muxes	DSPs	Clock period (ns)
RV32G	28528	15824	911	121	4	22
RV32IMF	28349	15741	903	118	4	22
RV32IAF	27924	15702	910	120	2	22
RV32IF	27739	15620	944	111	2	22
RV32IMA	23131	13396	391	107	2	22
RV32IM	22880	13914	418	98	2	22
RV32IA	22442	13875	425	99	0	22
RV32I	22270	13793	420	98	0	21

TABLE II. Clock Period and Number of Cycles of Different Benchmarks on three RTLs: General-purpose, Extension-level Customized RTL, and Instructions-level Customized RTL

TestName	RTL	Clock Period (ns)	Number of cycles
median	RV32G	22	83900
	RV32IMF	22	83900
	RV32-instructions-level	21	83900
dhrystone	RV32G	22	171398
	RV32IMF	22	171398
	RV32-instructions-level	22	171398
mm	RV32G	22	141535
	RV32IMAF	22	141535
	RV32-instructions-level	22	141535
mt-matmul	RV32G	22	96029
	RV32IMAF	22	96029
	RV32-instructions-level	22	96029
mt-vvadd	RV32G	22	320803
	RV32IMAF	22	320803
	RV32-instructions-level	22	320803
multiply	RV32G	22	67830
	RV32IMF	22	67830
	RV32-instructions-level	22	67830
pmp	RV32G	22	203729
	RV32IMF	22	203729
	RV32-instructions-level	22	203729
qsort	RV32G	22	192261
	RV32IMF	22	192261
	RV32-instructions-level	22	192261
rsort	RV32G	22	260976
	RV32IMF	22	260976
	RV32-instructions-level	21	260976
spmv	RV32G	22	373790
	RV32IMF	22	373790
	RV32-instructions-level	22	373790
towers	RV32G	22	64277
	RV32IMF	22	64277
	RV32-instructions-level	21	64277
vvadd	RV32G	22	87252
	RV32IMF	22	87252
	RV32-instructions-level	21	87252

extension-level customization one step further to implement instruction-level customization.[19]

B. *Instruction-Level Customization Results*

To assess the effectiveness of instruction-level customization, we need to conduct more experiments assessing

TABLE III. Power Consumption In Instruction Level For Each Benchmark

Benchmark	RTL	Total On-Chip Power (mW)	Energy (uJ)
median	RV32G	322	594.3
	Instruction Level	311	547.95
	Reduction Percentage	3.4%	7.8%
dhrystone	RV32G	322	1214
	Instruction Level	301	1134.9
	Reduction Percentage	6.5%	6.5%
matmul	RV32G	322	1002
	Instruction Level	319	993
	Reduction Percentage	0.9%	0.9%
mt-vvadd	RV32G	322	2272.56
	Instruction Level	317	2237.2
	Reduction Percentage	1.6%	1.6%
mt-matmul	RV32G	322	680
	Instruction Level	296	625
	Reduction Percentage	8.1%	8.1%
multiply	RV32G	322	480.5
	Instruction Level	301	449.17
	Reduction Percentage	6.5%	6.5%
pmp	RV32G	322	1443
	Instruction Level	298	1335.6
	Reduction Percentage	7.5%	7.5%
qsort	RV32G	322	1361
	Instruction Level	311	1315
	Reduction Percentage	3.4%	3.4%
rsort	RV32G	322	1848.7
	Instruction Level	311	1704.4
	Reduction Percentage	3.4%	7.8%
spmv	RV32G	322	2647.9
	Instruction Level	316	2598.588
	Reduction Percentage	1.9%	1.9%
towers	RV32G	322	455.3
	Instruction Level	311	419.7
	Reduction Percentage	3.4%	7.8%
vvadd	RV32G	322	618
	Instruction Level	311	569.8
	Reduction Percentage	3.4%	7.8%

the effect of such customization for different benchmarks. We use the previously described list of twelve benchmarks in our experiments. As described in Section 3, each of the twelve benchmarks in the RISC-V test suite is compiled for RV32G using the RISC-V toolchain, and then a list of unique instructions is extracted into a JSON file which, in turn, is fed to our extended version of the Rocket Chip generator. The resulting output is an RTL description of an instruction-level customized core suitable for each benchmark. Vivado is used to analyze the resulting RTL in terms of utilization, minimum clock period, and total on-chip power, while Verilator is used to simulate the execution of the benchmark on the generated core to compute the number of cycles needed to execute the benchmark. The results are compared with the general-purpose RISC-V and the extension-level customized RISC-V. The following tables show this comparative analysis.

Table II shows that generating an instruction-level-based core for some benchmark tests (median, rsort, towers, vvadd) leads to a 5% decrease in the clock period used; this reduction, as expressed by the subsequent equation, translates into a 5% decrease in execution time compared to running the same test on the general-purpose core and on the extension-level custom core. However, it is noteworthy that other tests have maintained the same execution time. The number of cycle values presented in Table II corresponds to the level of optimization of -O2. Similar results were obtained for other optimization levels.

$$\text{Execution Time} = \text{Clock Period} \times \text{Number of Cycles}$$

TABLE IV. Full Utilization Data For Each Benchmark

RTL	Number Of LUTs	D-Flip flops	F7 Muxes	F8 Muxes	DSPs
RV32G	28528	15824	911	121	4
median Instruction level	25972	15369	870	174	2
dhrystone Instruction level	25936	15369	869	174	2
mm Instruction level	27287	15710	921	121	4
mt-vvadd Instruction level	27243	15710	921	121	4
mt-matmul Instruction level	26071	15451	868	175	2
multiply Instruction Level	25936	15369	869	174	2
pmp Instruction Level	25927	15369	869	174	2
qsort Instruction Level	25972	15369	870	174	2
rsort Instruction Level	25972	15369	870	174	2
spmv Instruction Level	15628	15640	921	120	4
towers Instruction Level	25972	15369	870	174	2
vvadd Instruction Level	25972	15369	870	174	2

TABLE V. Number of Cycles of Different Benchmarks on Five Optimization Levels

Benchmark	O0	O1	O2	O3	Os
median	115268	83836	83900	110103	80804
dhrystone	210766	170813	171398	197601	165461
mm	473007	142051	141535	169606	127397
mt-vvadd	546247	321187	320803	347326	317570
mt-matmul	155089	95513	96029	122237	90549
multiply	98294	67570	67830	94033	64798
pmp	269945	203729	203729	236773	203729
qsort	227305	192389	192261	218464	188905
rsort	296797	260976	260976	288148	257880
spmv	404467	373099	373790	399754	369747
towers	95965	61633	64277	91645	58925
vvadd	116684	86736	87252	113460	83836

Table III presents the total power and energy of each benchmark on the chip. The Vivado tool is utilized to extract the total power values of the chip, while the energy consumption is calculated using the following equation:

$$\text{Energy} = \text{Power} \times \text{Number of Cycles} \times \text{Clock Period}$$

This equation quantifies the energy consumption by multiplying the power by the number of cycles and the time period. Then, the general-purpose RV32G and the custom-generated RTL at the instruction level are compared. It also illustrates the reduced power consumption achieved using the benchmark's custom-generated RTL at the instruction level compared to the general-purpose RV32G. The percentage presented represents the extent of power and energy reduction achieved by the customized RTL implementation, The table also shows that tests with clock period optimization (median, rsort, towers, and vvadd) have a maximum energy reduction percentage, as illustrated by the equation above. In comparison, other tests have the same power and energy reduction percentage.

Table IV presents the utilization data extracted from Vivado for RV32G general purpose and the RTL generated custom to the benchmark at the instruction level.

Table V demonstrates the variation in the number of cycles for different benchmarks when executed on simulators using different compiler optimization levels (-O0, -O1, -O2, -O3, -Os). Expectedly, the findings reveal that the utilization of Size optimization (-Os) leads to a reduction in the number of cycles, while the absence of any optimization (-O0) results in the highest number of cycles.

5. Limitations

One limitation of this work stems from difficulties in excluding certain instructions from the generated RTL due to how the Rocket Chip generator is implemented. For instance, when the (F) extension is enabled, only multiplication and division instructions can be excluded from the RTL generation while other instructions remain enabled. Similarly, in the (A) extension, instructions such as "amomax", "amomin" and "amoswap" cannot be excluded. On the other hand, in the (M) and (I) extensions, all instructions can be enabled or disabled individually except for the "add" instruction which cannot be excluded. Surpassing these

constraints requires implementing more significant changes in the Rocket Chip generator.

6. Conclusion and Future Work

This paper proposed a methodology to automate the implementation of custom RISC-V cores optimized for their target applications. Given the source code of an application, a generator will produce the RTL of a RISC-V core that supports only the specific mixture of instructions needed by that application. Compared with general-purpose processors, the experiments conducted demonstrated that the cores generated by our approach managed to achieve energy and execution time reductions reaching 8% and 5%, respectively, on some of the benchmarks used. These savings can be beneficial for battery-operated embedded systems and IoT devices. FPGAs are the target platform for this research, as they are naturally designed to be reconfigured for each new application.

In conclusion, the research demonstrates the potential benefits of using the RISC-V ISA and the automated approach to developing ASAs. Future refinements and optimizations in ISA extensions and customization techniques hold promise to achieve even more remarkable power efficiency and resource utilization improvements, reinforcing the value of our approach to develop efficient and specialized chip designs.

7. Acknowledgment

We would like to thank the Information Technology Industry Development Agency (ITIDA) in Egypt for generously funding this research through a Preliminary Research Proposal (PRP) fund “PRP2021.R31.11 - OneAppRISCV: Application Specific RISC-V Core Generator”. We would also like to thank the American University in Cairo, which hosted most of the project activities. Finally, we would like to thank the broader research community, which continues to develop and support open-source solutions.

References

[1] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, “A domain-specific architecture for deep neural networks,” *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[2] C. Weaver, R. Krishna, L. Wu, and T. Austin, “Application specific architectures: a recipe for fast, flexible and power efficient designs,” in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, 2001, pp. 181–185.

[3] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, “A novel methodology for the design of application-specific instruction-set processors (asips) using a machine description language,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, pp. 1338–1354, 2001.

[4] E. Hussein, B. Waschneck, and C. Mayr, “Automating application-driven customization of asips: A survey,” *Journal of Systems Architecture*, p. 103080, 2024.

[5] L. Luchterhandt, T. Nellius, R. Beck, R. Dömer, P. Kneuper, W. Mueller, and B. Sadiye, “Implementation of different communication structures for a rocket chip based risc-v grid of processing cells,” in *MBMV 2024; 27. Workshop*. VDE, 2024, pp. 79–89.

[6] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge, “Domain-specific architectures: Research problems and promising approaches,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, jan 2023. [Online]. Available: https://doi.org/10.1145/3563946

[7] M. Rothmann and M. Porrmann, “A survey of domain-specific architectures for reinforcement learning,” *IEEE Access*, vol. 10, pp. 13 753–13 767, 2022.

[8] H. Yu, G. Yuan, D. Kong, and C. Chen, “An optimized implementation of activation instruction based on risc-v,” *Electronics*, vol. 12, no. 9, p. 1986, 2023.

[9] A. Kumar M, V. Kumar, D. John, and S. Shanker, “Implementation and analysis of custom instructions on risc-v for edge-ai applications,” in *Proceedings of the 14th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2024, pp. 126–129.

[10] A. S. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[11] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[12] A. Waterman and K. Asanović, “The risc-v instruction set manual, volume i: Unprivileged architecture,” RISC-V Foundation, Tech. Rep. version 20240411, 2024.

[13] A. Waterman, Y. Lee, R. Avižienis, D. Patterson, and K. Asanović., “The risc-v instruction set manual, volume ii: Privileged architecture,” RISC-V Foundation, Tech. Rep. version 20240411, 2024.

[14] E. Cui, T. Li, and Q. Wei, “Risc-v instruction set architecture extensions: A survey,” *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[15] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, “Chipyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[16] L. Luchterhandt, T. Nellius, R. Beck, R. Doemer, P. Kneuper, W. Mueller, and B. Sadiye, “Towards a rocket chip based implementation of the risc-v gpc architecture,” in *MBMV 2023; 26th Workshop*. VDE, 2023, pp. 1–7.

[17] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[18] G. Du, “Evaluating a risc-v processor running benchmarks using the qemu virtual platform tool.” 2022.

[19] A. Gundrapally, Y. A. Shah, N. Alnatsheh, and K. K. Choi, “A high-performance and ultra-low-power accelerator design for advanced deep learning algorithms on an fpga,” *Electronics*, vol. 13, no. 13, pp. 7–11, 2024.