Comparison and analysis of open-source RISC-V cores for resource-constrained FPGAs

Abstract—**ARM and RISC-V are two dominant CPU instruction set architectures (ISAs), each offering distinct advantages in the embedded systems. While ARM's proprietary RISC architecture is widely adopted due to its power efficiency and robust ecosystem, its licensing model and static peripheral set pose limitations for resource-constrained and adaptable systems. RISC-V, on the other hand, is an open-source RISC architecture with a modular, and extensible design. Its flexibility enables hardware-software co-design and customization for specific applications, making it highly suitable for resource-constrained FPGA platforms. However, RISC-V's open-source nature introduces complexities due to its variations, implementation choices, and lack of standardized tooling across ecosystems. This paper compares and analyses RISC-V processor implementations for resource-constrained FPGAs. We compared various RISC-V implementations and analyzed them with the TinyML benchmark. The result helps developers design RISC-V driven reconfigurable applications leveraging tiny FPGA systems.**

Index Terms—**Risc-V; Internet of Things; FPGA**

I. Introduction

The ARM and RISC-V [1] instruction set architectures (ISAs) represent two prominent paradigms in processor design, widely used in embedded systems, Internet of Things (IoT), and FPGA-based applications. ARM, developed by ARM Holdings, is a proprietary Reduced Instruction Set Computing (RISC) architecture known for its power efficiency, performance, and mature ecosystem. ARM cores are extensively integrated into modern embedded designs, offering families of processors tailored for real-time, embedded, and full-fledged operating systems. However, ARM's proprietary licensing model restricts design flexibility, as developers must rely on predefined microcontrollers (MCUs) from semiconductor vendors with fixed peripherals and limited adaptability. This static design presents a bottleneck in applications requiring dynamic changes such as switching from compute-intensive tasks like Neural Networks and logic-based models such as Tsetlin Machines.

RISC-V, an emerging open-source RISC architecture developed by the RISC-V Foundation, addresses these challenges through its modularity, extensibility, and royalty-free licensing model. RISC-V's open-source nature allows for hardware-software co-design, enabling developers to create custom extensions tailored to specific application needs. The base instruction set, RV32I [2], is minimalistic, containing only 47 fixed-length instructions, simplifying implementation and improving portability for resource-constrained devices. Additionally, optional extensions such as the "M" extension for integer multiplication/division and the "C" extension for compressed 16-bit instruction encoding can significantly reduce code size and enhance efficiency. Variants like RV32E, designed for environments with extremely limited resources, further minimize core footprint by reducing the general-purpose registers to 16. A basic comparison between ARM and RISC-V implementation is given in Table I.

Despite its advantages, the adoption of RISC-V introduces certain complexities. The open-source nature of the ISA leads to a fragmented ecosystem with multiple variations and implementations. Developers and researchers face challenges in selecting optimal configurations, as the lack of standardized tools and verification frameworks can result in inconsistencies across designs. Furthermore, fine-tuning RISC-V cores for specific FPGA platforms requires significant expertise in hardware-software co-design. The absence of universally accepted benchmarks and performance metrics complicates comparisons between different implementations. Variability in toolchain support and debugging environments adds another layer of difficulty. The challenge for FPGA-based deployments extends to balancing core features with available FPGA resources, ensuring that the implementation meets performance and resource constraints. Additionally, the dynamic and evolving nature of the RISC-V ecosystem demands continuous learning and adaptation from developers, making it less accessible to those new to the architecture. Such complexities can cause hurdles to the development cycles and impact the overall feasibility of adopting RISC-V in certain projects.

From the research perspective, it is interesting to see the use of low-power small form factor FPGA devices in IoT devices, wearable technology, and embedded systems that require energy efficiency. A good example is the Lattice iCE40, which is supported by the open-source toolchain ecosystem, is easy to use, and is cost-efficient. Several studies have explored RISC-V implementations on FPGAs [3] [4], evaluating their performance, resource utilization, and adaptability in embedded systems. Existing works primarily focus on comparing RISC-V cores with ARM cores or analyzing specific optimizations for resource-constrained devices. However, a systematic evaluation of RISC-V cores tailored for lightweight FPGA platforms, along with an analysis of trade-offs in performance and resource usage, remains limited. Based on the given discussion, this paper tries to answer the following research question:

- **R1:** Which open-source RISC-V cores could be used for resource-constrained FPGA supported by open-source tools?

TABLE I: Comparison of ARM and RISC-V architectures

Aspect	ARM	RISC-V
Licensing	Proprietary, royalty-based	Open-source, royalty-free
Performance	Proven, highly optimized	Customizable, catching up
Ecosystem	Mature, widespread	Growing, community-driven
Customization	Limited without ARM's approval	Highly modular and extensible
Cost	Expensive licensing	No licensing costs

- **R2:** How can RISC-V ISA be extended following hardware-software codesign for open-source TinyML libraries?

In this paper, we aim to bridge this gap by conducting a comprehensive comparison and analysis of RISC-V processor implementations for resource-constrained FPGA devices. The paper aims to provide valuable insights into the design considerations, advantages, and limitations of RISC-V processors for resource-constrained FPGA applications, thus, promoting their adoption in next-generation embedded systems. We first review existing RISC-V implementations and extensions, focusing on the iCE40UP5K [5] chip to identify trade-offs and optimization opportunities. Additionally, we propose and validate a profiling workflow for TinyML applications on RISC-V cores, showcasing a practical approach to performance optimization.

The main contributions of this paper are given below:

- detailed comparison of RISC-V and ARM ISAs, highlighting the advantages of RISC-V for resource-constrained FPGA applications,
- review of existing RISC-V implementations and extensions to identify trade-offs and optimization opportunities,
- proposition and validation of a profiling workflow for TinyML applications on RISC-V cores, demonstrating a pathway to optimize performance.

The structure of this study is as follows. Section II presents a background of existing RISC-V implementations and extensions. This also discusses some recent relevant works on evaluating the RISC-V implementation. Section III and Section IV focus on evaluating existing RISC-V implementations that can be run iCE40UP5K chip. An existing TinyML application is tested and a suggested profiling workflow is proposed in Section V. Section VI discusses the results and challenges, while Section VII concludes the research done and presents some future work that can be applied to improve TinyML performance on small RISC-V cores.

II. Related work

Since its introduction, RISC-V has witnessed significant interest from both academia and industry. This is mainly due to the positive aspects including open-source, easily customizable and extendible, highly modular, and free. This section highlights key research and implementations, focusing on their contributions and limitations.

In [6] 16 implementations of RISC-V cores were selected and analyzed for their basic properties, including pipeline length, MMU and FPU support, debug capabilities, and licensing models. Among these, 3 cores were selected for detailed evaluation namely, Freedom Core, RI5CY, and PicoRV32. While Freedom Core and RI5CY demonstrated relatively high performance, achieving 1.61 and 1.10 DMIPS/MHz respectively, they consumed significant resources, requiring over 13,000 LUTs on a Xilinx Artix-7 FPGA. In contrast, PicoRV32, a minimalist implementation, used only 2,123 LUTs but offered lower performance at 0.13 DMIPS/MHz. This trade-off between performance and resource usage underscores the importance of tailoring RISC-V cores to specific FPGA constraints.

Another study contrasted the traditional one-size-fits-all approach of proprietary ISAs with the customizable nature of RISC-V [7]. It emphasized the growing adoption of RISC-V in industry, driven by its ability to support small, power-efficient cores. The study also highlighted the security benefits of open-source architectures, such as the absence of hidden hardware-level backdoors, and discussed the potential for custom security extensions. In proprietary cores, there is no way of ensuring that the manufacturer did not install a hardware-level backdoor. Security extensions can be used to fit specific needs.

Extensions of RISC-V for Specific Applications Several studies have explored custom extensions to optimize RISC-V cores for specific applications. An extensive survey of the RISC-V landscape is carried out in [8] discussing several official and custom extensions. It explains the extensive set of work being done in the Internet of Things area, including floating point computations and digital signal processing. Artificial intelligence also has multiple custom accelerators that have been researched for large neural networks. However, all these extensions are too broad and too large to be used in low-power FPGAs. There is a little research effort into tailored extensions targeting single algorithm.

In [9], a DSP coprocessor was implemented. It adds hardware support for several commonly used operations in FFT, such as complex number dot products and multiplications. Authors modified `liquid-dsp` library to use the new instructions. Hamming coding provided up to 70% clock cycle reduction and 45%-60% for FFT and digital filters. This came at a cost of 15%-30% LUT usage increase.

In [10], a software-defined radio (SDR) concept was approached. the work shows that with SDR implemented in FPGA, dynamic reconfiguration is possible. The article presents a DSP for an SDR coprocessor with packed SIMD instructions for 8/16-bit complex numbers. This is used for software FSK demodulation (used in Bluetooth), LoRa pream-

TABLE II: Open-Source FPGA toolchains

Toolchain	Description	Supported FPGA Families
Yosys	Open-source synthesis tool for RTL designs (Verilog).	Multiple FPGA families (general-purpose).
NextPNR	General-purpose place-and-route tool for multiple FPGA families.	Lattice iCE40, Lattice ECP5, Gowin (early).
IceStorm	Open-source FPGA toolchain for Lattice iCE40 devices.	Lattice iCE40.
Project Trellis	Open-source bitstream tools for Lattice ECP5 FPGAs.	Lattice ECP5.
Apicula	Open-source Gowin toolchain project (early stage).	Gowin FPGA (GW1N series, in development).
SymbiFlow	Unified open-source FPGA toolchain supporting multiple vendors.	Lattice, Xilinx, and other vendors.

ble detection, and FFT. For LoRa, 45% cycle count reduction was achieved with constant energy usage.

Floating Point Formats with RISC-V Another avenue of research has been the exploration of sub-32-bit floating-point formats. In [11], the authors examined 16-bit and 8-bit formats, which are increasingly popular in domain-specific accelerators for neural networks. While these smaller formats save storage and processing time, they require careful algorithmic adjustments to maintain numerical stability. However, fused-multiply-add (FMA) with an accumulator is an interesting concept and the authors shows that with only one normalization and rounding step, an improvement of accuracy and timing can be achieved.

In the context of IoT and edge computing, [12] introduced a high-performance floating-point unit (FPU) tailored for the RV32F extension. The proposed design offered a modular approach, allowing selective enabling of instruction blocks to balance performance and resource usage. Fully enabling the FPUx extension resulted in a substantial performance boost, with Whetstone benchmark scores increasing from 0.042 (software floating point) to 1.532. This demonstrates the potential of RISC-V to handle complex computations directly on edge devices, reducing reliance on cloud-based processing.

Despite these advancements, there is limited research on tailoring RISC-V extensions for single algorithms for highly resource-constrained environments. Most existing studies focus on broad applications, such as AI or IoT, without delving into specific use cases. Furthermore, the lack of standardized tools and frameworks for benchmarking and debugging RISC-V cores remains a significant challenge. This study aims to address these gaps by evaluating RISC-V implementations on the iCE40UP5K FPGA and proposing a profiling workflow for TinyML applications discussed in Section V.

A. Open-source support for FPGA

Contemporary FPGA hardware (silicon) is proprietary, but related toolchains usually have free licenses, making them accessible to hobbyists, researchers, and developers. However, Lattice iCE40 and ECP5 devices have the best support for open-source toolchains like Yosys, NextPNR, and SymbiFlow. Boards like TinyFPGA, iCEBreaker, and OrangeCrab are excellent choices for fully open-source FPGA development. These tools are summarised in the Table II.

B. Open-source RISC-V cores

Several RISC-V implementations are publicly available, but only open-source was considered to embrace the extendability RISC-V provides fully. It can prove useful in devices where on-site reconfiguration is necessary.

Unfortunately, open tools have limitations. Yosys [13], a synthesis tool, officially only supports Verilog-2005 input. Some SystemVerilog extensions are included, but not enough to fully support a CPU written in it. Several implementations were rejected because of this. Only Verilog or compilable to Verilog implementations are considered here. Surveyed implementations were summarized in Table III.

1) PicoRV32: A simple core named PicoRV32 [14] is a good starting point. This implementation was optimized for maximum frequency and minimum LUT usage. Thus, it can be integrated with high-frequency circuitry without a separate clock domain, or used with slower hardware without much timing penalties. The default CPU variant supports the RV32I base instruction set. Configuration options additionally support synthesizing RV32E, RV32IM, and RV32IMC cores. Also, the clock cycle and instruction counter are optionally available. They are useful for calculating clocks per instruction metric used in section IV. IRQ support is available. Custom *Pico Co-Processor Interface* (PCPI) is available. The extensions use this bus internally. This can prove useful when one would like to extend the core's functionality. A single data/instruction bus is exposed with a custom interface. Furthermore, AXI4-Lite [17] and Wishbone [18] are supported. Authors claim 0.305 DMIPS/MHz Dhrystone benchmark score and 5.232 clocks per instruction (CPI) with RV32IM core with fast multiplication enabled.

2) VexRiscV: Secondly, a more sophisticated implementation was considered. It is written in SpinalHDL, a higher-level language than Verilog. However, it compiles to it and can be used within existing designs. Modularity and parametrization enable the user to create a custom-built RISC-V CPU. *Small* variants support RV32I. Provided plug-ins provide "M", "C" extensions, as well as "A" (atomic instructions), "F" (single-precision floats) and "D" (double-precision floats). Separate data/instruction buses are exposed. AXI4-Lite, Wishbone, Avalon [19] bridges are available. Authors claim a 0.82 DMIPS/MHz Dhrystone benchmark score with a small and productive variant of the core (RV32I). It is a surprisingly high score, provided that it uses 33% less LUT than PicoRV32, which maxes out at 0.305 DMIPS/MHz on its fastest variant.

III. Methodology

This section outlines the research methodology employed to address the research questions posed in the introduction. Two experiments were conducted to evaluate the performance and

TABLE III: Comparison of RISC-V implementations available on the Internet

Repo	Project	Language	ISA	Variant	iCE40 LUT calculated	DMIPS per MHz claimed
[14]	PicoRV32	Verilog	RV32I	default	1984 (38%)	
			RV32IM	mul+div	3422 (65%)	
			RV32IM	mul+div fast	6486 (123%)	0.305
[15]	VexRiscV	SpinalHDL	RV32I	small	1071 (20%)	0.520
			RV32I	small and productive	1322 (25%)	0.820
			RV32IM	full no mmu no cache	5838 (111%)	1.210
			RV32IM	full no mmu no cache simple mul	5828 (110%)	
			RV32IF	small and productive dcache fpu	7431 (141%)	
[16]	FemtoRV32	Verilog	RV32I	quark bicycle	1175 (22%)	
	FemtoRV32	Verilog	RV32IM	electron	5102 (97%)	

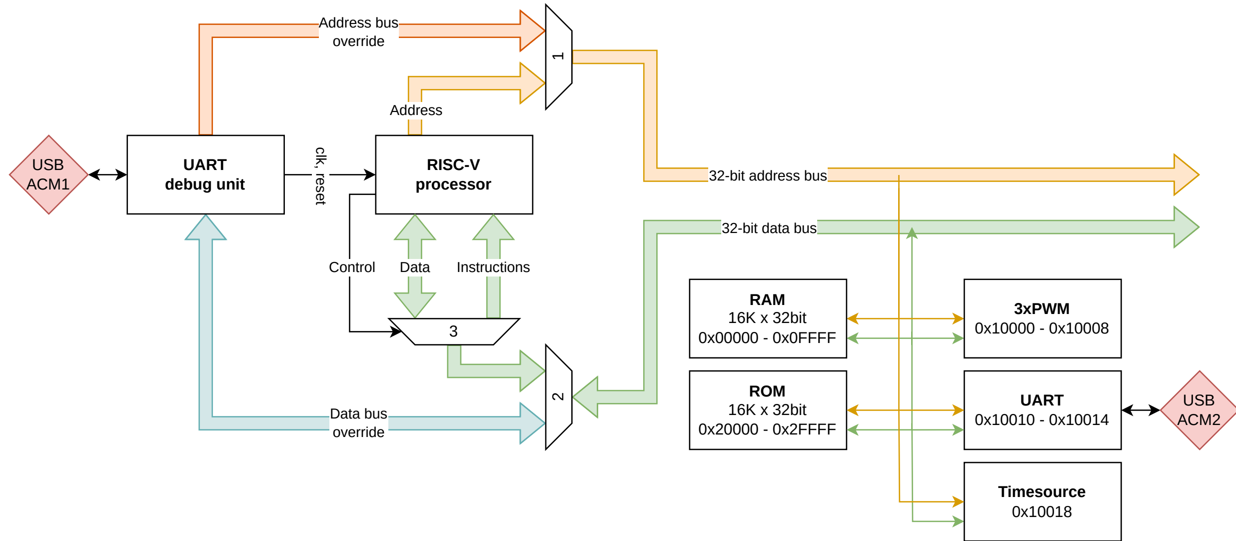

Fig. 1: Test environment used

extendability of RISC-V cores on resource-constrained FPGA platforms. The goal was to assess the suitability of RISC-V for such environments and explore architectural enhancements to improve the efficiency of ML algorithms.

A. Running RISC-V Cores on Resource-Constrained FPGA

The first experiment focused on implementing RISC-V cores on a small, affordable, low-power FPGA platform, specifically the iCE40UP5K. This platform was chosen due to its resource-constrained nature, which mirrors the limitations of embedded systems requiring lightweight solutions. The PicoRV32 core was used as the baseline for this experiment, as it is known for its minimalistic design and adaptability to resource-constrained environments.

The Dhrystone benchmark, a well-established measure of processor performance introduced in [20], was utilized for evaluation. This benchmark evaluates integer and string processing efficiency, which aligns with typical workloads of embedded systems. A statistical analysis of the most commonly used operands during Dhrystone execution was conducted to understand resource utilization patterns. These operands are summarized in Table IV which provided insights into instruction usage frequency critical for assessing the efficiency of the RISC-V ISA in resource-constrained environments.

B. Extending the ISA for ML Algorithms

The second experiment evaluated the extendability of the RISC-V ISA to improve the efficiency of statistical ML algorithms. Unlike accelerators for neural networks, which are well-researched and widely available, this experiment targeted classical statistical algorithms. These algorithms are crucial

for lightweight ML tasks common in IoT and edge computing applications.

The extendability of the ISA was explored by identifying the most frequently used operations in statistical ML workloads and designing custom instruction extensions to accelerate them. The goal was to determine whether the RISC-V core could be tailored to support specialized ML tasks without significantly increasing resource usage or reducing system performance.

A profiling workflow was proposed to integrate the custom extensions into the existing PicoRV32 core. The extended ISA was implemented and tested on the same FPGA platform to ensure consistency in evaluation.

C. Metrics

Three primary metrics were used to evaluate the cores:

- *LUT Usage*: The number of look-up tables (LUTs) consumed by the core was measured, as LUT usage directly impacts the feasibility of deploying the design on resource-constrained FPGAs.
- *Performance per MHz*: To ensure fair comparisons across different platforms and implementations, performance per MHz was analyzed. It is a good metric to compare cores outside of their SoCs. However, implementing extensions in energy-efficient (slow) FPGA chips brings non-trivial timing penalties.
- *Maximum Achievable Performance*: This was determined by multiplying the maximum frequency (obtained from the place-and-route tool) by the Dhrystone MIPS per MHz measured on the FPGA. This metric accounts for timing penalties associated with FPGA implementations, providing a realistic estimate of real-world performance.

IV. RISC-V Core Evaluation

Tests were performed using pico-ice board [21] as presented in Figure 2. The logic analyzer has been connected to one of the PMOD connectors and provided additional information during debugging.

The test environment schematic is based on [22] is shown in Figure 1. This design does not use any external memories. However, the device requires code downloads thus the *UART debug unit* was created. It can read/write all types of memory through muxes (1) and (2). Additionally, the debug unit supports clock stopping and resetting the core. Mux (3) is only present when the core implementation provides a separate instruction/data bus. Multiplexing them together brings a performance penalty but simplifies the design.

FPGA chip provides 4 blocks of single-port memory, each 256 Kbit large. This memory was organized into 32-bit wide blocks, forming the system's RAM and ROM. Also, memory-mapped I/O space was reserved. Currently, it features 3 PWM blocks driving the onboard RGB LED, 1 UART block for the core's text communication, and a time source register incrementing once per $1\mu s$.

PC communication is handled by the RP2040 available on the board. By default, it provides one UART interface bound with `ttyACM1`. This one is used for debug unit communication. However, a second UART was required to receive messages printed by the core itself. Due to the open-source nature of the firmware, it was modified to provide additional UART via `ttyACM2` interface.

TABLE IV: Dhrystone operations summary

Category	Type	Usage
Statements	Assignment	53%
	Control	32%
	Call	15%
Operators	Arithmetic	52.9%
	Comparison	39.2%
	Logic	7.8%
Operand types	Integer	54.4%
	Character	19.5%
	Enumeration	12.4%
	Boolean	4.6%
	Pointer	5.0%
	String	2.5%
	Array	0.8%
	Record	0.8%
Operands locality	Local	48.5%
	Global	7.9%
	Parameters	18.7%
	Function results	2.1%
	Constants	22.8%

Interfacing the hardware is done via a Python script [23]. It provides read/write+verify/reset commands. With this design, it is required to use the `--d32` flag. A `upload.sh` script is provided for convenience. It takes one argument, the hex file to upload. Also, it resets the core afterwards.

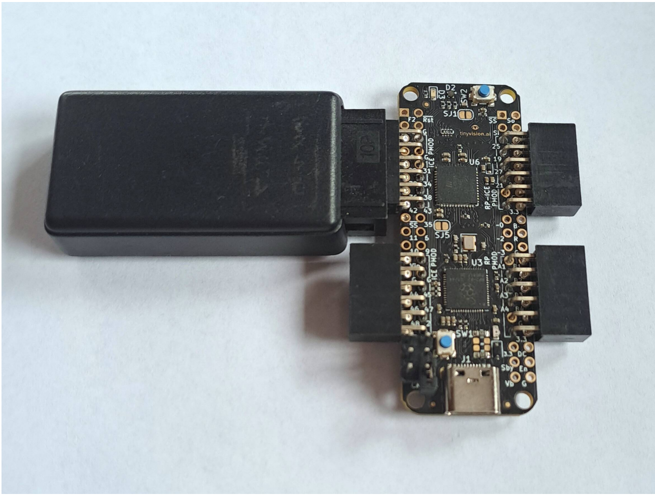

Fig. 2: Testbed used in experiments

A. Program details

The dhrystone benchmark loop was not modified in any way. It was adjusted to use the correct timesource and calculate final results in fixed point representation to reduce code size. The implementations of `malloc`, `strcpy`, `strcmp` were also sourced from PicoRV32. The code was compiled with `-O3 -ffreestanding -nostdlib` for RV32I cores with additional `-mabi=ilp32 -march=rv32im` for RV32IM cores. Linking was performed using a custom linker script. `_start` routine initialized stack, zeroed `bss` section, and performed a jump to `main`. Resulting `.hex` files were uploaded to the device via `upload.sh`, and the output was captured from `ttyACM2`. All source code and compilation infrastructure can be found in `dhrystone` directory of [22].

B. Core details

PicoRV32 default variant was instantiated with no Verilog parameters provided, except for the cycle and instruction counters enabled.

VexRiscV required a couple of small modifications. Apart from adding performance counters as above, the `cmdForkPersistence` option was enabled. It ensures that the bus request does not change in the middle of transactions. Some additional hardware is required, but it makes the core's bus much more predictable and easier to use.

C. Results

Achieved metrics were presented in Table V. LUT usages don't generally correspond with numbers seen previously. This is due to the additional hardware needed for benchmarking and synthesis optimizations.

PicoRV32 default instance actually got better results than the largest one advertised by authors. The only other variant that fits under 5K LUT is the simple multiplication and division variant conforming to RV32IM. It enlarged CPU by over 68% and brought only 9% performance gain per MHz, overall (at a maximum frequency) gaining only 6.8%.

VexRiscV turned out better than the PicoRV32 core. Pipeline implementation is really size-efficient. The small variant is 21% smaller than the default PicoRV32, yet it achieved 45% higher per MHz Dhrystone score. Unfortunately, the maximum frequency was lowered. At `F_MAX`, the VexRiscV core is only 6.8% better. Small-and-productive variant with as little as 5.5% LUT usage increase performs 27% better per MHz and 24% at `F_MAX`. The full variant didn't fit into the FPGA. The only extension providing the "M" extension was an iterative plug-in, which got 4.3% better per MHz results, but due to lower max frequency, it worsened its overall performance by 5.1%. It also takes 41% more space.

Overall, VexRiscV small-and-productive variant offers a sweet spot for LUT, performance per MHz, and `F_MAX`. It is half as performant per MHz as the ARM Cortex M0+ present in RP2040 on board.

V. TinyML Benchmark

For a plausible usage scenario, a modified version of the FogML SDK [24] and the Arduino example [25] were used. The exact code being run and tested can be found in [26] [27]. SDK was modified to provide bug fixes, size reductions (no implicit double conversions), and a RISC-V port. The example contains time/cycle measuring functions and, most importantly, executes only on non-zero reservoir data vectors.

A. TinyML pipeline

Benchmarking TinyML algorithms for inferencing based on NN has already been presented in the literature []. Therefore, we have decided to analyse the on-device learning algorithms for sensor data. Our benchmark uses the FogML set of tools designed to bring TinyML capabilities to resource-constrained microcontrollers, including ARM M0 cores. Unlike many other frameworks, FogML relies on traditional machine learning techniques, such as density-based anomaly detection and classifiers built with Bayesian networks, decision forests, and basic multi-layer perceptrons (MLPs). It also offers *on-device training* for anomaly detection. It employs reservoir sampling and local outlier factor (LOF) detection algorithms. The library also includes functionality for time-series processing, generating feature vectors with metrics like RMS, FFT, amplitude, and other low-level signal characteristics. Fig. 3 depicts the TinyML pipeline used in the experiments. The tests were run on time-series data windows containing 50 data points.

One of FogML's standout features is its use of source code generation for inference functions on embedded devices. This results in a significantly smaller memory footprint than computationally intensive approaches like deep neural networks.

B. Profiling

The results for the LOF algorithm are presented in Table VI. Tests were run using VexRiscV core or PicoRV32 default with counters. Interestingly, due to its higher maximum core clock, PicoRV32 can perform the learning 10% faster. In further tests, the `-Os` compilation flag and VexRiscV@16MHz were used.

TABLE V: Comparison of benchmark results for different RISC-V implementations

Core	Variant	iCE40 LUT CPU only	CPI	DMIPS per MHz	F_MAX MHz	DMIPS max
PicoRV32	default + counters	2017 (38%)	5.220	0.348	27.28	9.49
	mul+div + counters	3399 (64%)	5.282	0.380	26.69	10.14
VexRiscV	small (custom)	1588 (30%)	3.597	0.505	20.08	10.14
	small and productive (custom)	1676 (32%)	2.840	0.640	19.71	12.61
	small and productive (custom) with MulDivIterativePlugin	2369 (45%)	3.010	0.668	17.91	11.96
RP2040	ARM Cortex M0+			1.426	125	178

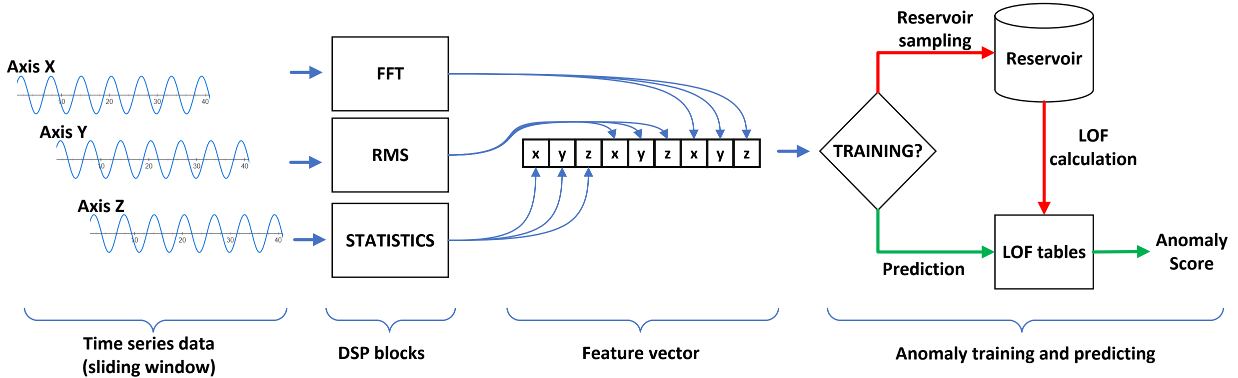

Fig. 3: TinyML processing pipeline used in the evaluation

Listing 1: Euclidean distance function in FogML

```
float tinyml_lof_normal_distance_vec(float *vec_a, float *vec_b, int N) {
  float x, dist = 0;
  int s, c1=0, c2=0, c3=0, csq=0;

  for(int i=0; i<N; i++) {
    s=C();     x = vec_a[i] - vec_b[i];     c1+=C()-s; // diff
    s=C();     x = pow2f(x);                c2+=C()-s; // square
    s=C();     dist += x;                   c3+=C()-s; // accum
  }
  s=C();       dist = sqrtf(dist);          csq=C()-s; // sqrt

  return dist;
}
```

TABLE VI: FogML LOF algorithm results

Opts	Core	Freq [MHz]	Learn [s]	Infer [s]	Size [kB]
-O0	Vex	16	12.65	0.26	51.6
-O1	Vex	16	12.16	0.25	40.0
-O2	Vex	16	12.17	0.25	39.8
-Os	Vex	16	12.20	0.26	35.7
-Os	RV32	16	16.58	0.34	35.7
-Os	RV32	24	11.05	0.22	35.7

TABLE VII: FogML DSP pipeline results

FFT	Feature vector length	DSP [s]	Learn [s]	DSP / Learn
no	12	0.063	12.19	0.52%
yes	18	0.241	16.70	1.44%

DSP FogML pipeline stages were also profiled. Test results are presented in Table VII. By default, 3 blocks are used: `BASE` (peak-to-peak and average), `ENERGY` (energy of the signal), and `CROSSINGS` (number of mean crossings). Enabling the `FFT` block (amplitude and frequency) causes the feature count to change to 18, DSP time rises 3.8 times, and learning takes 37% more than before. In the first case, DSP

TABLE VIII: Euclidean distance selective cycle count

Dims	diff	square	accum	sqrt
N	4919	14640	4215	2081
1	410	1220	351	173
	19%	57%	16%	8%

takes 0.52% of what the LOF algorithm does. This rises to only 1.44%, which suggests that DSP pipeline processing time, even with FFT enabled, is negligible. Focus should be put into the learning part of the workflow.

The long-running function was determined to be `tinyml_lof_learn()`. Each call to the learning routine costs 195 million cycles. All of its callees eventually call `tinyml_lof_normal_distance_vec()`, the Euclidean distance function, which consumes 192 million cycles in total. That is around 98% of what the learn call costs.

Said function is shown in listing 1. It calculates Euclidean distance in `N`-dimensional space. For this benchmark `N` is 12. It also uses the `C()` function to retrieve the number of cycles since the CPU boot. `pow2f(x)` returns `x*x`. Adding profiling code, increased the summed-up cycle count by 1.3%. The results were presented in Table VIII. On average, per 1 dimension, squaring 32-bit float numbers takes the most time, topping at around 57%. Next, we have the additions and subtractions at 16%/19%. The square root is only calculated once per call, hence the smallest impact.

These cycle counts are enormous, compared to what has been achieved with hardware accelerators. In [11] authors claim that their unit is capable of two 32-bit fused-multiply-additions in one cycle. This would reduce a square-addition step from 1500 cycles to only half. FPUx accelerator [12] can do additions, subtractions, and multiplications in 1 cycle, square rooting in 2. Keeping in mind that these were done on *big* and fast FPGAs. On iCE40, it may take a few more cycles. Estimating squaring at 100 cycles, this would still give a 50% speed boost to the learning process.

VI. Discussion

A. Instruction Set Extension

Extending the cores is a nontrivial task requiring extensive expertise and knowledge. At first, the core implementation should support some peripheral interface or be well enough written and documented to integrate custom modules directly into the core. Some cores are written in custom SDKs or languages. For example, VexRiscV is implemented in Scala-based SpinalHDL. Secondly, the extension should be tested and verified with simulation tools such as ICARUS Verilog or (in later production steps) by logic analysers on the FPGA itself.

The compiler has to be modified to recognize instructions and their respective mnemonics and opcodes. Lastly, the software running on the enhanced CPU should be modified to use the new optimized instructions. For example, floating-point multiplication functions could be re-defined not to emulate float support in software but to use provided FPU instructions.

Instead of developing general-purpose extensions to the processing core, the processing can be enhanced by passing selected heavy tasks directly to the hardware. Literature [9] gives the example of modifying existing libraries to exploit new functionalities. Such an approach could be used for FogML tools discussed in the paper by hardware accelerating distance metric calculations - in that case, a whole function for the Euclidean distance could be implemented as a dedicated hardware extension.

B. Optimization Techniques for Low-Power FPGA Implementations

Low-power FPGAs such as the iCE40UP5K are increasingly being used in resource-constrained environments, but achieving optimal performance on such platforms remains a challenge. Future research could focus on advanced optimization techniques to improve RISC-V implementations on these devices. For instance, pipeline optimizations, including dynamic instruction scheduling and hazard mitigation, could enhance throughput while maintaining a low resource footprint. Power gating and clock gating can also be employed to selectively turn off unused components, reducing energy consumption without compromising functionality.

Memory management is another critical area where optimizations can have a substantial impact. Strategies such as intelligent caching, memory compression, and shared memory architectures can help reduce latency and resource utilization. Additionally, researchers could investigate hardware-software co-design approaches to create architectures that are specifically tuned for RISC-V's modular ISA. By aligning the FPGA's architecture with the core's design, it would be possible to achieve significant improvements in both efficiency and adaptability for various applications.

C. Security Enhancements for IoT and Edge Devices

As RISC-V continues to gain traction in IoT and edge computing, ensuring the security of these devices will be a critical area of future research. Lightweight cryptographic accelerators, secure boot mechanisms, and runtime integrity monitoring are just a few of the areas that can be explored. For example, RISC-V's extensibility can be used to implement custom instructions for cryptographic operations, enabling secure data encryption and authentication while minimizing the computational overhead. Additionally, hardware-enforced isolation mechanisms can be developed to protect sensitive data and code from unauthorized access, ensuring robust security in multi-tenant environments.

Researchers could also explore dynamic security mechanisms that adapt to evolving threats in real-time. For instance, security-focused extensions could enable the detection and mitigation of malicious activity during runtime, providing an additional layer of defence against sophisticated attacks. Such mechanisms could include anomaly detection based on behavioural analysis or integrity checks that verify the

authenticity of software and firmware. By addressing these security concerns, RISC-V could establish itself as a reliable and scalable platform for secure IoT and edge computing deployments.

VII. Conclusions

The paper aimed to analyse the RISC-V cores available with open-source licences. The results provide a comprehensive analysis of their requirements and processing speeds. Simple PicoRV32 core, implemented directly in Verilog, provides a simplistic RISC-V core that can be used for extensions and research on ISA. A better, more complex VexRiscV core achieved 68% more DMIPS per MHz in the Dhrystone benchmark. However, it was still around two times slower per MHz than the Raspberry Pi RP2040 chip with Cortex M0+.

A lot of interest is being put into expanding RISC-V with custom extensions. However, the LUT count limitations need to be obeyed. Large extensions are not feasible to implement in tiny, affordable FPGAs. It would be already profitable to just boost the squaring part of the Euclidean metric. Maybe, an FMA unit would fit inside the iCE40 lattice and, provided sub-1500 cycles per operation, would boost both learning and inference performance.

References

[1] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[2] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*. RISC-V International, 2021, document Version 20211213. [Online]. Available: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view

[3] T. Zheng, G. Cai, and Z. Huang, "A Soft RISC-V Processor IP with High-performance and Low-resource consumption for FPGA," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 2538–2541.

[4] T. Harmina, D. Hofman, and J. Benjak, "DCT Implementation on a Custom FPGA RISC-V Processor," in *2023 International Symposium ELMAR*, 2023, pp. 163–167.

[5] iCE40 UltraPlus – ML/AI Low Power FPGA. [Online]. Available: https://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus

[6] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer, "Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation," in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, 2019, pp. 1–6.

[7] S. Greengard, "Will RISC-V revolutionize computing?" *Communications of the ACM*, vol. 63, no. 5, pp. 30–32, 2020.

[8] 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.

[9] K. Li, W. Yin, and Q. Liu, "A Portable DSP Coprocessor Design Using RISC-V Packed-SIMD Instructions," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.

[10] H. B. Amor, C. Bernier, and Z. Přikryl, "A RISC-V ISA Extension for Ultra-Low Power IoT Wireless Signal Processing," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 766–778, 2022.

[11] L. Bertaccini, G. Paulin, M. Cavalcante, T. Fischer, S. Mach, and L. Benini, "MiniFloats on RISC-V Cores: ISA Extensions with Mixed-Precision Short Dot Products," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–16, 2024.

[12] X. Lin, H. Liu, X. Zheng, H. Gao, S. Cai, and X. Xiong, "FPUx: High-Performance Floating-Point Support for Cost-Constrained RISC-V Cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 10, pp. 1945–1949, 2024.

[13] yosys – Yosys Open SYnthesis Suite. [Online]. Available: https://github.com/YosysHQ/yosys

[14] PicoRV32 - A Size-Optimized RISC-V CPU. [Online]. Available: https://github.com/YosysHQ/picorv32

[15] VexRiscv. [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[16] FemtoRV32: a minimalistic RISC-V CPU. [Online]. Available: https://github.com/BrunoLevy/learn-fpga/tree/master/FemtoRV

[17] AMBA AXI and ACE Protocol Specification. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/

[18] WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. [Online]. Available: https://wishbone-interconnect.readthedocs.io

[19] Introduction to the Avalon Interface Specifications. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html

[20] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, p. 1013–1030, oct 1984. [Online]. Available: https://doi.org/10.1145/358274.358283

[21] Pico Ice test board. [Online]. Available: https://pico-ice.tinyvision.ai/

[22] RISC-V implementation survey test environment. [Online]. Available: https://github.com/MrJake222/riscv-ice40

[23] Debug UART interface. [Online]. Available: https://github.com/MrJake222/debug_uart

[24] FogML SDK (original). [Online]. Available: https://github.com/tszydlo/fogml_sdk

[25] FogML Arduino example code. [Online]. Available: https://github.com/tszydlo/FogML-Arduino

[26] FogML SDK (modified). [Online]. Available: https://github.com/MrJake222/fogml_sdk/tree/riscv

[27] FogML RISC-V example code. [Online]. Available: https://github.com/MrJake222/FogML-RiscV