Comparison and analysis of RISC-V processor implementations for resource-constrained FPGAs

Norbert Morawski `nmorawski@student.agh.edu.pl`

June 2024

1 Introduction

A lot of todays embedded design contain an ARM core. They suit a variety of applications, due to offering different families ranging from embedded, real-time to application cores designed for running a full-fledged OS. However, they require a license to use. This is a hard stop for any hobbyist or small company to deploy such cores. Writing and maintaining your own Instruction Set Architecture (ISA) is also not an option. However, they are open ISAs gaining traction [9] and ready to be used.

RISC-V is an example. It can be used and extended to best suit one's individual needs. It is also supported by GCC. The base instruction set (RV32I) is kept to a minimum of only 47 instructions. These make a viable compile target. The instructions are fixed-length, all 32-bit wide [1]. These simplifications reduce the size of the implementation and make it quite portable, in particular, small FPGA implementations are possible.

Base ISA provides 32 general-purpose registers, but an RV32E variant for resource-constrained environments, in which only half of the registers are supported, is available (as of 2024 it is a draft extension). It can further reduce the core size. Integer multiplication and divisions are supported via "M" extension. Code size can be reduced by 25-30% by implementing "C" extension with 16-bit dense instruction encoding.

This article focuses on evaluating existing RISC-V implementations which can be run on small, affordable, low-power chip iCE40UP5K [10]. RV32I base 32-bit ISA is the main point of interest, however, RV32IM is also considered.

The structure of this study is as follows. In section 2.2 a list of suitable implementation is defined. Section 4 describes testing methodology and results.

2 State of the Art

2.1 Articles

Ever since RISC-V was described in [2], it is gaining more and more interest from academia and the industry. In [3] also several implementations were surveyed. It defined 3 use cases for large, medium, and small cores. Authors agreed that RISC-V can be tailored to ones needs.

In [4] a classic closed-source one-size-fits-all approach is contrasted with revolutionary open approach. It focuses on the possibility to create a small, tailor-made, and power-efficient cores. Reportedly, industry is also implementing more and more of RISC-V cores in their designs. Also, security and transparency issues were raised. In proprietary cores there is no way of ensuring that no hardware-level backdoor was installed by the manufacturer. Security extensions can be employed to fit specific needs. Overall, authors are optimistic about bright future RISC-V will bring.

A lot of effort is being put in custom extensions. In [5] a DSP coprocessor was implemented. It adds hardware support for several commonly-used operations in FFT like 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.

In [6], a software-defined radio (SDR) concept was approached. Multi-chip designs are cumbersome and not energy efficient. Also, dedicated chips are not flexible. With SDR implemented in FPGA, dynamic reconfiguration is possible. The article presents a DSP for SDR coprocessor with packed SIMD instructions for 8/16-bit complex numbers. All of this is used for software FSK demodulation (used in Bluetooth), LoRa preamble detection, and FFT. For LoRa, 45% cycle count reduction was achieved with constant energy usage.

Summing up, a lot of progress is being actively made in RISC-V field. Surveys, articles, and custom

Table 1: Comparison of RISC-V implementations available on the Internet

Repo	Project	Language	ISA	Variant	iCE40 LUT	DMIPS per MHz claimed
[12]	PicoRV32	Verilog	RV32I	default	1984 (38%)	
			RV32IM	mul+div	3422 (65%)	
			RV32IM	mul+div fast	6486 (123%)	0.305
[13]	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%)	
[14]	FemtoRV32	Verilog	RV32I	quark bicycle	1175 (22%)	
	FemtoRV32	Verilog	RV32IM	electron	5102 (97%)	

extensions bring hope to wide adoption and royalty-free computing.

2.2 Open-source RISC-V cores with open-source tools

There is a number of RISC-V implementations available on the Internet. To fully embrace the freedom RISC-V provides, only open-source were considered. No licenses are required to use this software. They can prove useful in devices where on-site reconfiguration is necessary.

Unfortunately, open tools have limitations. Yosys[11], 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 due to this. Only Verilog or compilable to Verilog implementations are considered here.

Surveyed implementation were summarized in Table 1.

2.2.1 PicoRV32

A simple core named PicoRV32 [12] 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 separate clock domain, or used with slower hardware without much timing penalties.

Default CPU variant supports RV32I base instruction set. Configuration options additionally support synthesizing RV32E, RV32IM and RV32IMC cores. Also clock cycle and instruction counter are optionally available. They are useful for calculating clocks per instruction metric used in section 4. 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 custom interface. Additionally, AXI4-Lite [15] and Wishbone [16] 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.

2.2.2 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 enables 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 [17] bridges are available.

Authors claim 0.82 DMIPS/MHz Dhrystone benchmark score with small and productive variant

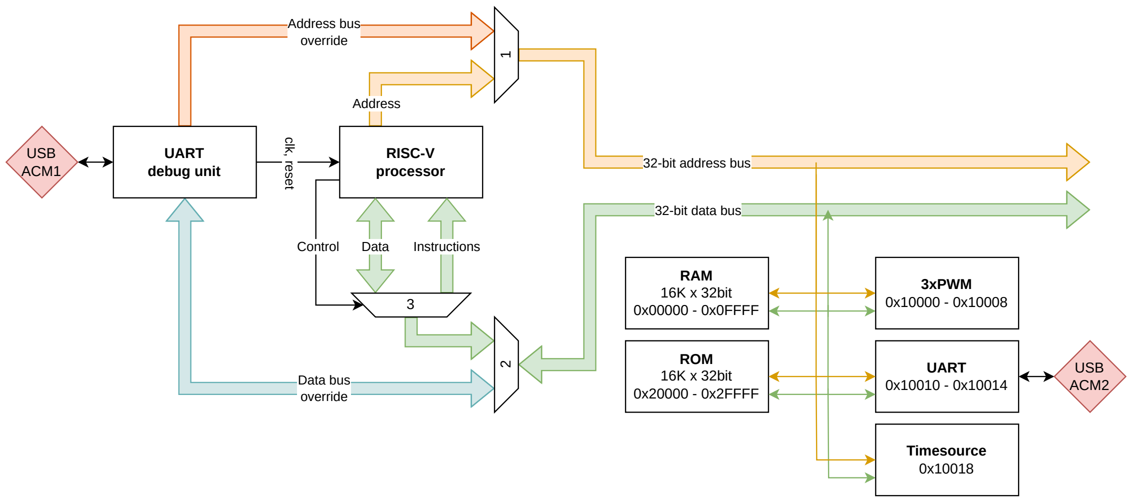

Figure 1: Test environment used

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 it's fastest variant.

3 Methodology

The cores were run in a environment described in Section 4. Author-provided SoC were not used. A Dhrystone benchmark, based on the one found in PicoRV32 repository, was used. This benchmark was introduced in [7] and by performing statistical analysis, most used operands were determined. They were summarized in Table 2.

Three sets of metrics were used to compare the cores, first being the core LUT usage. It is really important to keep this under control on tiny FPGA chips. Secondly, performance per MHz was considered. It is a good metric for comparing cores outside of their SoCs. However, implementing extensions in energy-efficient (slow) FPGA chips brings nontrivial timing penalties. A max frequency was taken from the place-and-route tool and then multiplied by Dhrystone MIPS per MHz measured on-chip. This gives a consistent view of what we can expect from an implementation crunching real numbers.

4 Experiments

Test environment schematic [18] is shown in Figure 1. This design doesn't use any external memories. However, device requires code downloads, thus the *UART debug unit* was created. Through muxes (1) and (2) it can read/write all types of memory. Additionally,

Table 2: 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%

the debug unit supports clock stopping and resetting the core.

Mux (3) is only present when core's implementation provides separate instruction/data bus. Multiplexing them together brings a performance penalty, but it simplifies the design.

FPGA chip provides 4 blocks of single-port memory, each 256kbit large. This memory was organized into 32-bit wide blocks, which form the RAM and ROM of the system. Also memory-mapped I/O space was reserved. Currently it features 3 PWM blocks driving the on-board RGB LED, 1 UART block for 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 binded 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.

Interfacing with the hardware is done via a Python script [19]. It provides a read/write+verify/reset commands. With this design it is required to use `--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.

Test were performed using pico-ice board [20]. It can be seen in Figure 2. Saleae-compatible scope was used. It fits nicely into the PMOD connectors and provides a way to test Verilog code on real FPGA, in addition to simulations.

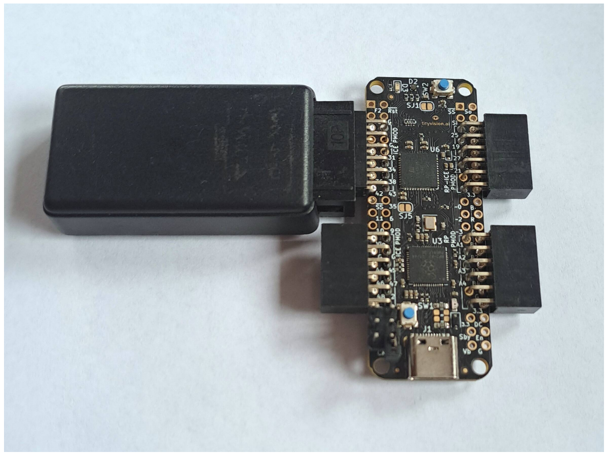

Figure 2: Testbed used in experiments

4.1 Program details

Dhrystone benchmark loop was not modified in any way. It was adjusted to use correct time-source and calculate final results in fixed point representation to reduce code size. The implementations of `malloc`, `strcpy`, `strcmp` were also sourced from PicoRV32. Code was compiled with `-O3 -ffreestanding -nostdlib` for RV32I cores with additional `-mabi=ilp32 -march=rv32im` for RV32IM cores. Linking was performed using 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 output was captured from `ttyACM2`. All source code and compilation infrastructure can be found in `dhrystone` directory of [18].

4.2 Core details

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

VexRiscV required a couple of small modifications. Apart from adding performance counters as above, `cmdForkPersistence` option was enabled. It ensures bus request doesn't 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.

4.3 Results

Achieved metrics were presented in Table 3. LUT usages don't generally correspond with numbers seen previously. It's due to additional hardware needed for benchmarking and synthesis optimizations.

PicoRV32 default instance actually got better results that the biggest one advertised by authors. The only other variant that fitted 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, overally (at maximum frequency) gaining only 6.8%.

VexRiscV turned out better than PicoRV32 core. Pipeline implementation is really size-efficient. Small variant is 21% smaller than default PicoRV32, yet it achieved 45% higher per MHz Dhrystone score. Unfortunately, max 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`. Full variant didn't fit into the FPGA. Only extension providing "M" extension was an iterative plug-in, which got 4.3% better per MHz results, but

Table 3: 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

due to lower max frequency worsened it's 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 ARM Cortex M0+ present in RP2040 on board.

5 Future work

Some results provided better performance per MHz but lowered maximum CPU frequency could run. This resulted in worse maximum speed, but it's not always the goal. Power consumption for the cores can also be an important aspect. However, it was not done in this work, mainly because it requires board modifications.

A lot of interest is being put in expanding RISC-V with custom extensions. Cores tested in this work offer either Verilog peripheral interface or SpinalHDL module system. Next step could be to design and implement custom ISA extension. Possible use cases include hardware signal processing, which may enable fast sensory data acquisition for applications like [8].

References

[1] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*. Document Version 20211213. RISC-V International, 2021. URL: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view.

[2] Krste Asanović and David A Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[3] Roland Höller et al. "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. DOI: 10.1109/MECO.2019.8760205.

[4] Samuel Greengard. "Will RISC-V revolutionize computing?" In: *Communications of the ACM* 63.5 (2020), pp. 30–32.

[5] Kai Li, Wei Yin and Qiang 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. DOI: 10.1109/ISCAS46773.2023.10181681.

[6] Hela Belhadj Amor, Carolynn Bernier and Zdeněk Přikryl. "A RISC-V ISA Extension for Ultra-Low Power IoT Wireless Signal Processing". In: *IEEE Transactions on Computers* 71.4 (2022), pp. 766–778. DOI: 10.1109/TC.2021.3063027.

[7] Reinhold P. Weicker. "Dhrystone: a synthetic systems programming benchmark". In: *Commun. ACM* 27.10 (Oct. 1984), pp. 1013–1030. ISSN: 0001-0782. DOI: 10.1145/358274.358283. URL: https://doi.org/10.1145/358274.358283.

[8] Tomasz Szydlo. *Online Anomaly Detection Based On Reservoir Sampling and LOF for IoT devices*. 2022. arXiv: 2206.14265.

Online Resources

[9] *RISC-V Google Trends*. URL: https://trends.google.com/trends/explore?date=all&q=risc-v (visited on 08/06/2024).

[10] *iCE40 UltraPlus – ML/AI Low Power FPGA*. URL: https://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus (visited on 09/06/2024).

[11] *yosys – Yosys Open SYnthesis Suite*. URL: https://github.com/YosysHQ/yosys (visited on 08/06/2024).

[12] *PicoRV32 - A Size-Optimized RISC-V CPU*. URL: https://github.com/YosysHQ/picorv32 (visited on 27/04/2024).

[13] *VexRiscv*. URL: https://github.com/SpinalHDL/VexRiscv (visited on 27/04/2024).

[14] *FemtoRV32: a minimalistic RISC-V CPU*. URL: https://github.com/BrunoLevy/learn-fpga/tree/master/FemtoRV (visited on 27/04/2024).

[15] *AMBA AXI and ACE Protocol Specification*. URL: https://developer.arm.com/documentation/ihi0022/e/ (visited on 08/06/2024).

[16] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. URL: https://wishbone-interconnect.readthedocs.io (visited on 08/06/2024).

[17] *Introduction to the Avalon Interface Specifications*. URL: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html (visited on 08/06/2024).

[18] *RISC-V implementation survey test environment*. URL: https://github.com/MrJake222/riscv-ice40 (visited on 09/06/2024).

[19] *Debug UART interface*. URL: https://github.com/MrJake222/debug_uart (visited on 09/06/2024).

[20] *Pico Ice test board*. URL: https://pico-ice.tinyvision.ai/ (visited on 01/07/2024).