RISC-V Instruction Characterization

Ethan Kaley & David Dionisopoulos

Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island
Kingston RI, 02881

***Abstract*— IoT devices and embedded systems are becoming increasingly popular and more involved in our everyday life. These systems generally run on a microprocessor using a specific Instruction Set Architecture (ISA). The microprocessor of choice is typically a processor that uses a RISC (Reduced Instruction Set Computing) which provides very power efficient performance for embedded applications. Existing ISAs today do not provide for the ability to easily or cost effectively build processors for embedded IoT applications. We are proposing RISC-V as an modular ISA that would solve the issues of cost, simplicity, and efficiency present with the use of existing ISAs. Our goal is to analyze code from practical C++ programs and find common functional processor instruction blocks that are reusable across a wide range of IoT applications for embedded systems and could lead to versatile, low cost, and efficient microprocessor chips.**

This paper makes two key contributions. First, it proposes and describes an easily consumable framework for analyzing applications written in C++ and targeting IoT embedded applications. It describes and details RISC-V toolchain components and how to leverage them to target specific ISA instructions. Second, it presents and details an extensible parsing program to analyze pseudo and non-pseudo static assembly instructions specific to a target ISA. This program can identify unnecessary ISA instructions and enable hardware designs to effectively eliminate implementation for a more simplistic and power efficient programmable embedded system.

I. Introduction and Motivation

RISC-V is an open-source instruction set architecture (ISA) that has gained increasing attention in recent years. It is a relatively new ISA compared to the more established architectures such as ARM and x86, but it has rapidly gained momentum in the industry due to its flexibility, simplicity, and openness. RISC-V stands for "Reduced Instruction Set Computing - Five", which refers to the philosophy of having a simpler and more streamlined set of instructions that can be executed faster and more efficiently.

ISA extensions are typically standardized by industry organizations such as the RISC-V Foundation or the ARM Architecture Advisory Committee, and are often implemented in new versions of processors by hardware manufacturers. As such, ISA extensions play an important role in the ongoing evolution and improvement of computer architecture.

One of the key advantages of RISC-V is that it is an open standard, which means that anyone can use, modify, and distribute it without any licensing fees or restrictions. This open-source nature of RISC-V has led to a growing community of developers and enthusiasts who are working to improve the architecture and build new hardware and software solutions based on it. This can lead to it being more cost-effective and power-efficient than other Instruction Set Architectures (ISAs) due to its simplicity and modularity.

RISC-V instruction set is simpler than many other ISAs, which means that it requires fewer transistors to implement in hardware. This can lead to lower manufacturing costs for processors based on RISC-V. Additionally, RISC-V's simplicity allows for more efficient use of hardware resources, reducing power consumption and improving energy efficiency. Figure 1 details a comparison between two similar processors, one ARM and one RISC-V.

ISA	Width (bits)	Frequency (GHz)	Dhrystone Performance (DMIPS/MHz)	Area mm2 (no caches)	Area mm2 (16 KB caches)	Area Efficiency (DMIPS/MHz/mm2)	Dynamic Power (mW/MHz)
ARM	32	>1	1.57	0.27	0.53	3.0	<0.080
RISC-V	64	>1	1.72	0.14	0.39	4.4	0.034
R/A	2	1	1.1	0.5	0.7	1.5	≥0.4

Fig. 1. ISA analysis process flow diagram [10].

RISC-V's modularity allows designers to tailor the architecture to their specific needs, which can lead to more efficient implementations. The core RISC-V instruction set is small and simple, consisting of only 47 instructions, but it can be extended with additional instructions to support a wide range of applications, from embedded systems to high-performance computing [4]. For example, if an application only requires a subset of the RISC-V instruction set, the processor can be designed to only include those instructions, reducing the size and power consumption of the processor.

RISC-V supports a variety of data types and addressing modes, including integers, floating-point numbers, and vectors. It also includes support for hardware concurrency and atomic memory operations, which are important for multi-core processors and high-performance computing.

Another advantage of RISC-V is its support for variable-length instructions. In contrast to fixed-length instructions used in many other ISAs, variable-length instructions allow for more efficient use of memory bandwidth and cache utilization. This can lead to improved performance and lower power consumption.

RISC-V has already been adopted by several major tech companies, including Google, Nvidia, and Western Digital, who are using it for a wide range of applications such as data centers, artificial intelligence, and storage [12]. As RISC-V continues to gain popularity and support, it has the potential to disrupt the traditional ISA landscape and usher in a new era of innovation and collaboration in the tech industry [10].

II. Prior Work(s)

In a recent article [11], A natively flexible 32-bit Arm microprocessor, John Biggs et al explored the use of flexible electronic devices that aren't built with conventional semiconductor materials. The thin-film technology offers many advantages over traditional silicon-based microprocessors, including improved thinness, reduced weight, and lower manufacturing costs. The authors investigated how these advantages could be leveraged to develop microprocessors that are more flexible and adaptable to a wider range of applications.

While the article focused on an alternative to silicon-based technology, our study explores the benefits of a "reduced" RISC-V instruction set for these flexible microprocessors. By using a minimalistic instruction set architecture, our study aims to reduce the number of instructions required for microprocessors used in flexible electronic devices. This reduction in instruction set size can lead to several benefits, including reduced storage capacity requirements, smaller program sizes, and lower power consumption.

One of the key advantages of using a reduced instruction set architecture in flexible microprocessors is its potential to minimize the footprint of these devices. With fewer instructions, the amount of storage required is reduced, which means fewer transistors are needed to store the instructions. This, in turn, leads to a smaller footprint, making these microprocessors more suitable for a wider range of applications. Moreover, the reduced program size can also lead to lower power consumption, which is particularly important for battery-powered devices.

Our study builds on the findings of the article by Biggs et al and aims to contribute to the ongoing dialogue in the field by exploring the benefits of a reduced instruction set architecture for flexible microprocessors. By reducing the instruction set size, we aim to improve the efficiency and performance of microprocessors in these applications. Our study provides important insights into the relationship between RISC-V instruction characterization and the development of microprocessors for flexible electronic devices, and highlights the need for further research in this area.

Overall, the findings of the article by Biggs et al and our study highlight the potential of flexible microprocessors to revolutionize the field of microelectronics. The use of thin-film technology, combined with a reduced instruction set architecture, can lead to more efficient and adaptable microprocessors that are better suited for a wide range of applications. Further research in this area will be crucial to unlocking the full potential of these devices and advancing the field of microelectronics

III. Concepts and Contributions

RISC-V presents an open standard detailing all instructions and their design rationale. RISC-V details 3 main benefits over other existing ISAs, the potential for greater innovation, sharing core designs license free, and the ability for processors becoming affordable for more devices like IoT applications [10]. RISC-V ISA focuses on the concept of *base-plus-extension* as a design principle. What this

means is that to reduce costs and improve efficiency the ISA focuses on a stable core set of instructions and then builds upon those with extended instructions for optional but more powerful and tailored applications. What we have implemented here is a process for consuming raw source code and generating a programmable or human readable dataset of RISC-V instructions targeting a minimalistic ISA implementation for the least complex and most power efficient embedded applications.

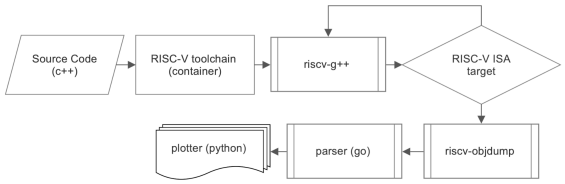

Fig. 2. ISA analysis process flow diagram.

A. Source code and toolchain

We have gathered example programs that exercise different data structures and algorithms of the c++ language [7]. Gathering and leveraging open source code presents unique challenges. Licensing and complete availability of all dependencies prevented us from running our process against the most common examples of IoT applications. In replacement of common raspberry pi and arduino code projects we utilized an extensive library of open source c++ examples which we felt exercised the language in a broad enough manner. These programs use arrays, pointers, loops, math functions, manipulation of common data structures, and implement common algorithms like binary search trees. While not an accurate representation of a real world application for a IoT use case, we felt that they were diverse enough to test the proposed hypothesis. This framework and process is documented clearly and reproducible to allow anyone to bring their own source and run the same analysis for any existing or future programs.

B. Runtime environment

Linux containers have been around for many years and excel when used as a reproducible sandbox to set up a consistent environment for experimentation and testing. We utilized a linux container to install and configure the risc-v toolchain and all the dependencies for our scripts and code to build, analyze, parse, and generate our data.

C. RISC-V Compilation

We leveraged the RISC-V toolchain which contains gcc and g++ compilers to build all the example source code. Specifically since we were building c++ code the guidance from the RISC-V organization is to use g++ and the underlying release is currently tracking the 12.X version [8]. As referenced in the appendix the base ISA *riscv32i* and the fully ratified ISA (specification 2.2) *riscv64imafd* (abbreviated to *riscv64g*) were the compilation targets [8][4]. The target ABI of *ilp32* was also specified for the *riscv32i* target to ensure compliance with the target system. When compiling these programs the output was object files not fully executable binaries.

D. RISC-V Object Dump

Utilizing compiler switches for different ISA targets yielded object files and subsequently demanded the use of the objdump. Disassembly of the object files yielded complex assembly files that were difficult to parse. This was partly due to the default for compilers and disassemblers to use pseudo instructions to represent their respective ISA instructions. We utilized the RISC-V objdump argument *-M no-aliases* to control this behavior and output canonical instructions in raw asm files [8]. Another tool, *asm-parser*, was chained together with objdump to cleanup and prepare the assembly files for further parsing. This tool was provided open source [6] and was utilized to facilitate the stdin/stdout behavior of objdump and filter out nuances with the binary and whitespace behavior of GNU Objdump. It should be noted that the use of this tool does no actual instruction parsing nor changes or edits the instruction canonical names in any way.

E. Assembly parser (go)

We implemented a parser to analyze the instructions from the assembly output files. That parser was written in go (golang) and is a standalone compiled program included in the project source. This program ran against the directory containing the entirety of the assembly output from the previous object dump stage. The output of this program was corresponding arrays of instructions and instruction counts output in CSV format. This output was an intermediary format as it was injected into the following plotting stage. However these CSV files could stand alone and be utilized by another program to be further processed or analyzed in a different manner if desired.

F. Plotter (python)

A graphical representation of the output data was plotted using python. Represented in each plot was the name of the source program, the compiled target ISA, and every instruction and its frequency of use within that program. It should be noted that these instruction counts were performed statically and not dynamically so program loops or repeated instruction blocks are not fully accounted for here. These plots were used to visualize the difference and potential implementation savings that the RISC-V ISA enables for embedded applications. Instruction implementation and the *base-plus-extension* nature of the ISA enables efficiency improvements and cost savings [10].

Fig. 3. ISA base integer instructions. [5]

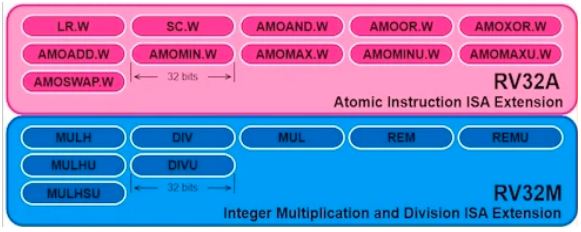

Fig. 4. A and M extension instructions. [5]

IV. Results and Discussion

RISC-V is designed to be modular [2], where it is composed of several base parts and optional extensions. The toolchain passes a flag to denote which configuration of RSIC-V is disassembled. To compare the effectiveness of this project, we used a basic version of RISC-V, 32i, and a more robust version of RISC-V, 64g. RISC-V 32i stands for registers that are 32 bits wide and uses the base integer instruction set which encompasses 47 total instructions [1][2]. See figure 3 for a visual of the instruction set. 64g uses 64 bits wide registers and on top of the integer instruction set, it also includes other extensions; M – Multiplication and Division, A – Atomic Instructions, F – Single-Precision – Point, and others [1][2]. The 64g instruction set contains over 100 total instructions, which can be problematic for systems that have storage requirements. See figure 4 for an example of some of the extensions used. A more extensive list of the instructions is in Appendix A, figure A5. Below are steps that go through the process of the 40 projects that were analyzed for both 32i and 64g. For an exhaustive list, refer to the GitHub repo [13].

Once the files were disassembled and parsed from the original .cpp source, an assembly file was created for both architecture types. Each file was upwards of 1000 lines of instructions. Figure 5 represents a small sample of the assembly code that was generated.

```
jalr    ra,0(ra) # b98 <main+0xb74>
addi    a4,a0,0
lui     a5,0x0
addi    a1,a5,0                 #                 0
<_ZL20__gthread_key_deletei>
addi    a0,a4,0
auipc   ra,0x0
jalr    ra,0(ra) # bb0 <main+0xb8c>
addi    a4,a0,0
lui     a5,0x0
addi    a1,a5,0                 #                 0
<_ZL20__gthread_key_deletei>
addi    a0,a4,0
auipc   ra,0x0
jalr    ra,0(ra) # bc8 <main+0xba4>
lw      a0,-20(s0)
auipc   ra,0x0
jalr    ra,0(ra) # bd4 <main+0xbb0>
addi    a5,a0,0
addi    a1,a5,0
lui     a5,0x0
addi    a0,a5,0                 #                 0
<_ZL20__gthread_key_deletei>
```

Fig. 5. Assembly Output.

To view all of the assembly files in their entirety refer to the GitHub link and under asm/examples will give the complete list of assembly code [13].

The assembly language was then parsed further into a CSV file which loops through and makes a list of each individual instruction used, and how many times each instruction was called in the program. An example of this format is shown below in figure 6 with the 32 bit output on top and then 64 bit output on the bottom.

```
[bge,lui,addi,jalr,jal,auipc,bne,sw,lw]
[2,12,54,16,1,11,2,18,22]

[addi,sd,ld,jal,lui,auipc,sw,jalr,lw,
addiw,bge,bne]
[65,9,9,1,12,11,9,16,13,6,2,2]
```

Fig. 6. CSV data structure output.

The number below the instruction is the corresponding quantity of instruction calls. For the complete list of CSV files, they can be found in the GitHub link under csv/examples [13].

Finally, a python script reads the CSV files and splits each row into its own list, and is used to generate a plot as shown in figures 7 and 8. The figures both plot the instructions used, which is displayed in the X-axis of the bar graph, versus how many iterations of that instruction was called displayed in the Y-axis.

In the example shown in figures 7 and 8, we have a 56.25% reduction in the number of total instructions used. This was calculated by equation 1.

```
% = [1 - (32i total number of
instructions / 64g total number of
instructions)] * 100
```

Equation 1. Instruction percentage

On average between all the sample projects, there was a reduction of 26% in the number of instructions used from 64g to 32i.

In the math functions example it is shown that the total number of instructions for 32i is greater than the 64g by 1.5%. However, the average of total instruction calls between all examples does show a 4.84% reduction from 64g to 32i.

Appendix figures A1 and A2 have an example where both instructions and instruction calls are reduced in 32i versus its 64g counterpart where there's a reduction of 25% in instructions and 10% in instruction calls.

Appendix figures A3 and A4 is another example where 32i has a 50% reduction in total instructions used but only a mere 1% of instructions calls

Please refer to the GitHub link under plots for a comprehensive list of all the generated plots [13].

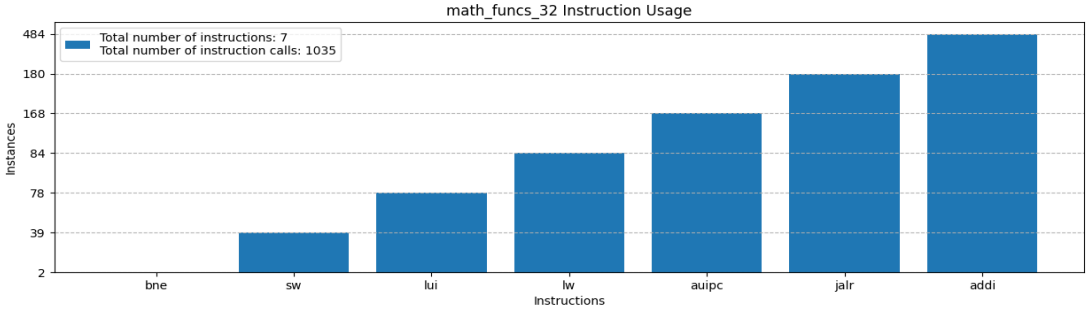

Fig. 7. 32i results for math functions sample program

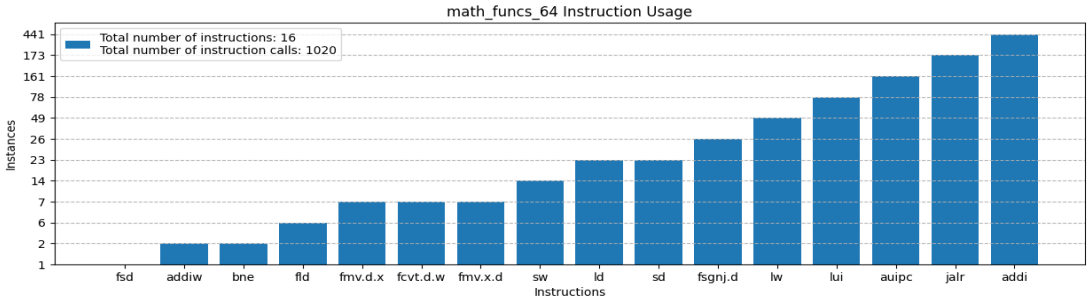

Fig. 8. 64g results for math functions sample program

VI. Conclusion

RISC-V instruction characterization has revealed interesting patterns of instruction use across a wide range of applications. Our study found that by targeting a subset of the RISC-V ISA, individual instruction use was decreased by an average of 20%, while maintaining a similar number of total instruction calls. This observation suggests that many of the instructions included in the full ISA may not be necessary for a wide range of embedded applications. By targeting a smaller subset of the ISA, embedded developers could potentially reduce program complexity and improve the overall efficiency of their designs.

Our study also found that the program's final size varied depending on the complexity of the instructions utilized. When targeting the standard ISA instructions, we observed a smaller program size, while targeting the full ISA, which includes all extensions, resulted in a larger program size. This finding further emphasizes the potential benefits of targeting a reduced subset of the RISC-V ISA, as it may lead to a more streamlined and efficient design and saves space for storing the program.

It is worth noting that these analyses and observations were performed statically on disassembled machine instructions. As such, they do not take into account dynamic factors such as loop iteration or repeated instruction cycles that would be observed when running the program code. However, the ability to analyze RISC-V programs to determine commonality of ISA instruction use across a wide range of embedded applications has the potential to yield cost-effective and low-power programmable processing units.

In conclusion, our study provides evidence that when designing programmable RISC-V processors for embedded applications, implementing only a subset of the RISC-V ISA can reduce complexity, program size, and ultimately, the cost of these devices. These findings are consistent with prior work in the field and highlight the potential benefits of a reduced instruction set architecture for certain applications. By focusing on a smaller subset of the ISA, developers can potentially create more efficient and cost-effective embedded systems.

VII. Future Work

While we have shown a reduction in complexity and size is feasible, theoretically a reduction in power consumption would follow suit. However without more complex simulation to analyze the dynamic runtime performance of these programs making the claim that power was reduced would be inaccurate. This is an area for future research exploring the combined use of software simulation and hardware measurement analysis to produce power consumption results for these embedded programmable processors.

VIII. Statement of Work

Project Component	Ethan Kaley	David Dionisopoulos
Plan	50	50
Meetings	50	50
Proposal	50	50
Presentation	50	50
Final Project Report	50	50
Collection of Example Programs	0	100
RISC-V toolchain setup	50	50
RISC-V compilation scripts/tools	80	20
RISC-V disassembly scripts/tools	20	80
RISC-V runtime environment (container)	100	0
RISC-V assembly parsing (go)	100	0

RISC-V instruction plotting/charts (python)	0	100
RISC-V ISA research/documentation	50	50
Creation and Contributions to Github	50	50
TOTAL	50	50

References

[1]"Standard Extensions - RISC-V - WikiChip," *en.wikichip.org*. https://en.wikichip.org/wiki/risc-v/standard_extensions (accessed May 04, 2023).

[2]"RISC-V," *Wikipedia*, Jan. 13, 2021. https://en.wikipedia.org/wiki/RISC-V

[3]E. Engheim, "RISC-V Instruction-Set Cheatsheet," *Medium*, May 19, 2022. https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

[4]A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2," 2017. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[5]"An Introduction to RISC-V—Understanding RISC's Open ISA - Technical Articles," *www.allaboutcircuits.com*. https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture/

[6]"asm-parser," *GitHub*, Mar. 03, 2023. https://github.com/compiler-explorer/asm-parser (accessed May 04, 2023).

[7]S. Iravanian, "C++ Tutorial Samples," *GitHub*, Apr. 10, 2023. https://github.com/sinairv/Cpp-Tutorial-Samples (accessed May 04, 2023).

[8]"RISC-V Toolchain Conventions," *GitHub*, May 02, 2023. https://github.com/riscv-non-isa/riscv-toolchain-conventions/blob/master/README.mkd (accessed May 04, 2023).

[9]"RISC-V ELF psABI Document," *GitHub*, May 04, 2023. https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc (accessed May 04, 2023).

[10]K. Asanović and D. Patterson, "Instruction Sets Should Be Free: The Case For RISC-V," 2014. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf

[11]J. Biggs *et al.*, "A natively flexible 32-bit Arm microprocessor," *Nature*, vol. 595, no. 7868, pp. 532–536, Jul. 2021, doi: https://doi.org/10.1038/s41586-021-03625-w.

[12]"RISC-V," *RISC-V International*. https://riscv.org

[13]E. Kaley, D. Dionisopoulos "URI ele548 Final Project," *GitHub*, May 04, 2023. https://github.com/ekaley/ele548 (accessed May 04, 2023).

Appendix A

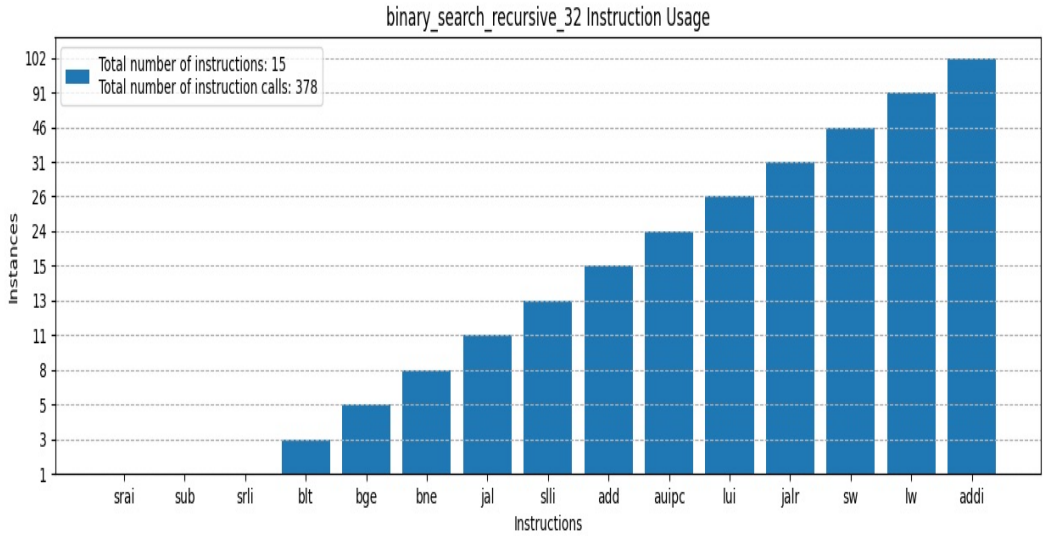

Figure A1. 32i results for recursive sample program

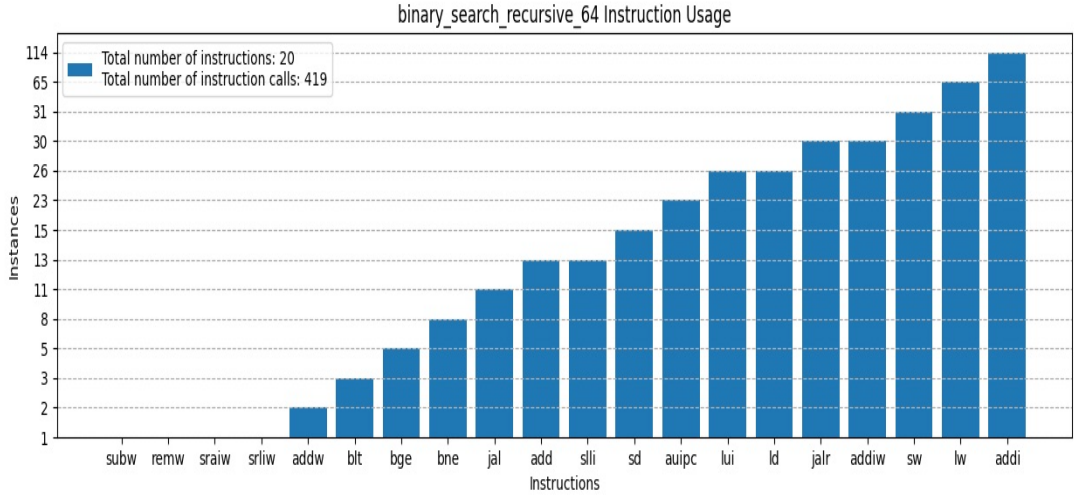

Figure A2. 64g results for recursive sample program

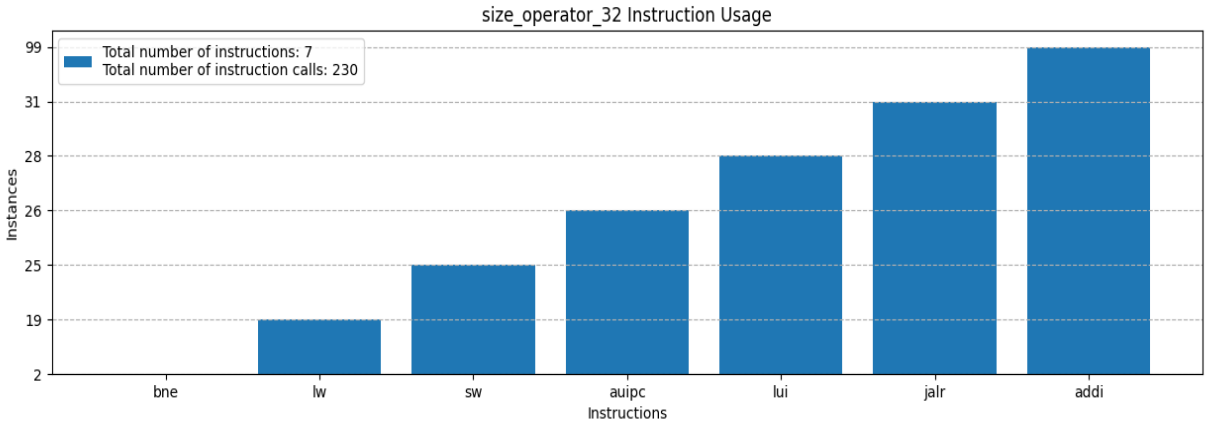

Figure A3. 32i size operator sample program

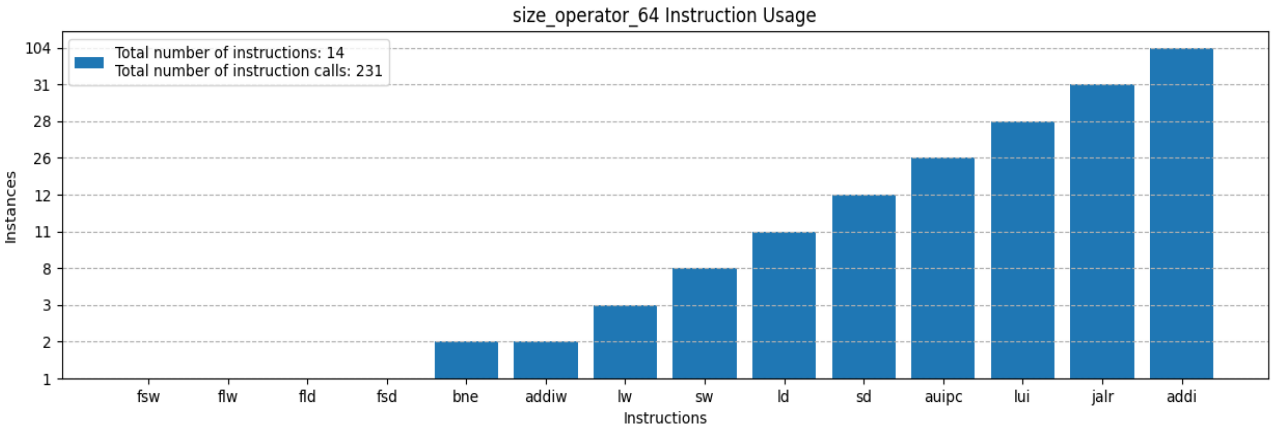

Figure A4. 64g size operator sample program

Category	Name	Format	RV32I Base	+RV64	+RV128
Loads	Load Byte	I	`LB rd,rs1,imm`		
	Load Halfword	I	`LH rd,rs1,imm`		
	Load Word	I,Cx	`LW rd,rs1,imm`	`LD rd,rs1,imm`	`LQ rd,rs2,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`		
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`LWU rd,rs1,imm`	`LDU rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`		
	Store Halfword	S	`SH rs1,rs2,imm`		
	Store Word	S,Cx	`SW rs1,rs2,imm`	`SD rs1,rs2,imm`	`SQ rs1,rs2,imm`
Arithmetic	ADD	R,Cx	`ADD rd,rs1,rs2`	`ADDW rd,rs1,rs2`	`ADDD rd,rs1,rs2`
	ADD Immediate	I,Cx	`ADDI rd,rs1,imm`	`ADDIW rd,rs1,imm`	`ADDID rd,rs1,imm`
	SUBtract	R,Cx	`SUB rd,rs1,rs2`	`SUBW rd,rs1,rs2`	`SUBD rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`		
	Add Upper Imm to PC	U	`AUIPC rd,imm`		
Logical	XOR	R	`XOR rd,rs1,rs2`		
	XOR Immediate	I	`XORI rd,rs1,imm`		
	OR	R,Cx	`OR rd,rs1,rs2`		
	OR Immediate	I	`ORI rd,rs1,imm`		
	AND	R,Cx	`AND rd,rs1,rs2`		
	AND Immediate	I	`ANDI rd,rs1,imm`		
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLLW rd,rs1,rs2`	`SLLD rd,rs1,rs2`
	Shift Left Immediate	I,Cx	`SLLI rd,rs1,shamt`	`SLLIW rd,rs1,shamt`	`SLLID rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRLW rd,rs1,rs2`	`SRLD rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLIW rd,rs1,shamt`	`SRLID rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRAW rd,rs1,rs2`	`SRAD rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAIW rd,rs1,shamt`	`SRAID rd,rs1,shamt`
Compare	Set <	R	`SLT rd,rs1,rs2`		
	Set < Immediate	I	`SLTI rd,rs1,imm`		
	Set < Unsigned	R	`SLTU rd,rs1,rs2`		
	Set < Unsigned Imm	I	`SLTIU rd,rs1,imm`		
Branches	Branch =	SB,Cx	`BEQ rs1,rs2,imm`		
	Branch ≠	SB,Cx	`BNE rs1,rs2,imm`		
	Branch <	SB	`BLT rs1,rs2,imm`		
	Branch ≥	SB	`BGE rs1,rs2,imm`		
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`		
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`		
Jump & Link	J&L	UJ,Cx	`JAL rd,imm`		
	Jump & Link Register	UJ,Cx	`JALR rd,rs1,imm`		
Synch	Synch threads	I	`FENCE`		
	Synch Instr & Data	I	`FENCE.I`		
System	System CALL	I	`SCALL`		
	System BREAK	I	`SBREAK`		
Counters	ReaD CYCLE	I	`RDCYCLE rd`		
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`		
	ReaD TIME	I	`RDTIME rd`		
	ReaD TIME upper Half	I	`RDTIMEH rd`		
	ReaD INSTR RETired	I	`RDINSTRET rd`		
	ReaD INSTR upper Half	I	`RDINSTRETH rd`		

32-bit Formats

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	opcode
R4	rs3	funct2	rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

16-bit Formats

	15–14–13–12	11–10	9–8–7	6–5–4–3–2	1–0
CI1	funct4	rd		rs1	op
CI2	funct4	rd		imm[4:0]	op
CJ	funct4	jump target			op
CI3	funct3	imm[2:0]	rd′	imm[4:3]	rs1′ op
CB	funct3	rs2′	imm[2:0]	imm[4:3]	rs1′ op
CR	funct3	rs2′	rd′	op	rs1′ op

Figure A5 RISC-V Base ISA [10]