



RVfplib: A Fast and Compact Open-Source Floating-Point Emulation Library for Tiny RISC-V Processors

Conference Paper**Author(s):**

Perotti, Matteo; Tagliavini, Giuseppe; [Mach, Stefan](#) ; Bertaccini, Luca; [Benini, Luca](#) 

Publication date:

2022

Permanent link:

<https://doi.org/10.3929/ethz-b-000517867>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)

Originally published in:

Lecture Notes in Computer Science 13227, https://doi.org/10.1007/978-3-031-04580-6_2

RVfplib: A Fast and Compact Open-Source Floating-Point Emulation Library for Tiny RISC-V Processors

Matteo Perotti¹[0000–0003–2413–8592], Giuseppe Tagliavini²[0000–0002–9221–4633],
Stefan Mach¹[0000–0002–3476–8857], Luca Bertaccini¹[0000–0002–3011–6368], and
Luca Benini^{1,2}[0000–0001–8068–3806]

¹ ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland

{mperotti, smach, lbertaccini, lbenini}@iis.ee.ethz.ch

² University of Bologna, Viale del Risorgimento 2, 40136 Bologna BO, Italy
{giuseppe.tagliavini}@unibo.it

Abstract. Small, low-cost IoT devices rely on floating-point (FP) software emulation on 32-bit integer cores when the cost of a full-fledged FPU is not affordable. Thus, the performance and code size of the FP emulation library are decisive for meeting energy and memory-size constraints. We propose RVfplib, the first ISA-optimized open-source library for single and double-precision IEEE 754 FP emulation on RV32IM[C] cores. RVfplib is 59% smaller and 2x faster than the GCC emulation library, on average. On benchmark programs, code size reduction is 39%, and performance boost 1.5x. RVfplib is 5.3% smaller than the leading closed-source RISC-V commercial library.

Keywords: RISC-V · Embedded · IoT · Floating-Point · Library · Size · Performance.

1 Introduction

Low-cost Internet of Things (IoT) devices are often subject to tight constraints on their silicon area and memory, which are precious resources in the embedded systems domain and impact cost and energy consumption [8]. At the same time, processing FP workloads is a common requirement for many applications. FP support enables programmers to satisfy the requirements on dynamic range and precision. In addition, deriving the fixed-point variant of an algorithm proven to be safe with floating-point numbers is often time-consuming and, in some cases, very challenging. However, small cores cannot always afford hardware Floating Point Units (FPUs) and rely on software emulation of FP instructions. Consequently, the code to be stored in memory is inflated, inducing performance overhead and increased total energy consumption due to higher execution times and added memory accesses. The code size cost is particularly relevant since FP emulation support can dominate the total code size of small programs, reaching up to 8 kB just for the single and double-precision basic operations. In this scenario,

using small and fast FP emulation libraries is necessary to be competitive in the market.

The RISC-V Instruction Set Architecture (ISA) is gaining industrial traction in IoT applications where cost is a major concern. The main challenge for RISC-V low-cost microcontroller units (MCUs) is to reduce code size [3], as currently experimental evidence shows that the Arm ISA (ARMv7-M), its mature compilers, and highly size-optimized libraries generate smaller code on average [12, 15]. The code size issue mainly affects applications that require FP arithmetic. In this case, long FP software emulation functions add a remarkable code size overhead, even if only a few FP computations are needed.

In this work, we present the following contributions:

1. *RVfplib*, the first open-source IEEE 754³ FP library for RISC-V, manually optimized for low code size and high performance for both single and double-precision FP. RVfplib is compatible with the RV32IM[C] ISA, and implements addition, subtraction, multiplication, division, as well as comparisons and conversions. Double-precision division is optional in RVfplib; it targets low code size and is compatible with cores without an integer divider.
2. *RVfplib.nd*, the reduced version of RVfplib that considers subnormal inputs/outputs as correctly signed zeroes. RVfplib.nd is compatible with the RV32EM[C] ISA and has smaller code size than RVfplib, making it the perfect candidate for tightly memory-constrained devices.
3. A comparison of the code size and performance of all RVfplib functions with their counterparts provided by *libgcc*. Moreover, we perform a code size comparison between the functions in RVfplib.nd and the ones available within SEGGER *emFloat*, the current state-of-the-art closed-source competitor. We also compare code size of RVfplib with the Arm-optimized libgcc code.
4. An analysis of the real code size and performance impact that RVfplib has on full programs.

The rest of the paper is organized as follows: in Section 3, we describe the structure of RVfplib and the main ideas that led to its development, as well as the techniques used to optimize it and a code comparison with libgcc. In Section 4, we present the experiments used to evaluate RVfplib figures of merit, and we show the corresponding results in Section 5. We close our work with insights about further improvements to RVfplib and the conclusion of the analysis in Section 6 and Section 7.

2 Related work

Researchers have proposed different solutions to provide FP capabilities to a core when the system area is strictly constrained. When a full-fledged FPU leads

³ The library presents some deviations from the standard. It does not support exception flags, it produces only fixed quiet NaNs, and it provides nearest-even or toward-zero rounding only.

to an excessive area increase, designers can integrate a slower but tiny FPU, crafted for tightly constrained IoT cores [2]. Another possibility is to implement hardware/software approaches, in which hardware optimizations in the integer datapath speed up critical operations used in the FP emulation libraries [13]. Nonetheless, both the solutions can be adopted only if the system tolerates the related area overhead, and do not apply to systems that already exist.

Integer-only cores that cannot afford an area increase can execute FP programs only through FP emulation libraries, usually provided by compiler vendors along with their compilation toolchain. For example, the Arm Keil compiler comes with the IEEE 754-1985 compliant fplib [1], and GCC with FP support within libgcc, its low-level runtime library [10]. Since the optimization of these libraries is essential for producing fast code with a low memory footprint, FP emulation libraries can also be manually crafted at the assembly-level to ensure the best code size and performance possible. libgcc provides optimized code for well established ISAs like Arm but lacks customized support for relatively new ISAs like RISC-V, which should rely on compiling the generic high-level FP emulation C functions. The novelty of the RISC-V solution results in sub-optimal code size and performance that makes it less attractive with respect to the Arm-based alternatives.

In addition to what is available in compiler ecosystems, designers have implemented optimized FP libraries for specific processors [14] [5] and for the maximum flexibility and compliance with the IEEE 754 standard, like SoftFloat [19]. However, these solutions are non-RISC-V specific.

To the best of our knowledge, the only available assembly-optimized RISC-V FP library is emFloat, designed by SEGGER [17]. However, this library is closed-source and does not support subnormal values, flushing them to correctly signed zeroes instead.

3 RVfplib design

RVfplib is the first open-source optimized FP emulation library for RISC-V processors, for both single and double-precision FP. Its main goals are low code size and increased performance. Implicitly, this implies lower energy consumption thanks to the reduced memory bandwidth and execution time.

RVfplib is wholly written in RV32IM assembly. Thanks to the modularity of the RISC-V C extension, it is also compatible with RV32IMC ISA since the compiler can compress all the compressible instructions on request.

Functions in RVfplib adhere to the interface of the corresponding libgcc functions [10] and have their same names to ensure compatibility with GCC and a fast porting to real programs. The aliasing induces GCC to automatically link using RVfplib functions, if implemented, instead of the ones from libgcc, without additional modifications to the program. Therefore, there is no need to explicitly call the RVfplib functions, as the compiler does it automatically.

RVfplib functions have been obtained with ISA-specific assembly level optimizations starting from the libgcc FP functions, with an approach similar to

the one used for Arm [9]. Compliance with the IEEE 754 standard rules for FP encoding and computation presents the same deviations that hold for the libgcc FP support compiled with the default options, namely:

- Exception flags are not supported, and exceptional events only provide their pre-defined output (i.e., divisions by zero result in a NaN).
- All the produced NaNs are quiet, in the form of `0x7FC00000` for single-precision and `0x7FF8000000000000` for double-precision.
- Only the default *round to nearest, ties to even* rounding mode is supported for the majority of the operations (as in the default libgcc implementation, some of the conversion functions round toward zero).

3.1 Structure

RVfplib is a static library that comes in two different variants:

- `RVfplib.a`: the standard version, which targets low code size and increased performance.
- `RVfplib.nd.a`, which treats subnormal values as signed zeroes and shows an even smaller code size.

Each variant includes the functions listed in Table 1, in which both the SoftFloat and the libgcc names are reported. The two *not-equal* functions are aliased with the *equal* ones, as they have the same behavior. Both libraries can be compiled with particular code that can increase performance in the presence of specific input operands, with an additional code size overhead. For example, the multiplication can include code to deal with power-of-two operands, speeding up the processing of specific patterns while increasing the code size. Choosing between one implementation or the other depends on the system constraints and input workloads.

To further push toward reducing the memory footprint of the library, we also implemented part of the same FP support environment provided by SEGGER emFloat, treating subnormal values as correctly signed zeroes. Thanks to the reduced requirement for registers in its design, RVfplib.nd is compliant with the RV32EM ISA (i.e., with only 16 registers in the register file). The library currently comes with a double-precision division that does not use any integer hardware divider, which cannot be included in such small cores.⁴ For this reason, this function is optional and is only included when targeting the smallest code size possible. If performance is a more critical constraint, the standard double-precision division from libgcc is used instead.

⁴ Such a processor would not be fully compliant with the RV32IM/RV32EM ISA since the M extension also requires an integer divider. Nevertheless, the compiler allows for avoiding hardware divisions even when compiling RV32IM code.

3.2 Design choices

RVfplib benefits from some essential ideas that, together with the functional algorithmic choices, contribute to crafting optimized RISC-V functions that reduce code size and execution times.

1. *Make the common case fast*: FP algorithms take different decisions depending on the received inputs and create different control paths within the code. The latency of each function strongly depends on the inputs since different data patterns are processed differently. Optimizing the paths taken by the common input patterns (normal values) is a methodology for reducing the average latency.
2. *Avoid memory references*: RVfplib minimizes data memory bandwidth reducing register spilling in function prologues/epilogues. This is accomplished by using only caller-saved registers. libgcc functions, on the contrary, do not limit register usage and have bloated prologues/epilogues.
3. *No function calls*: Whenever the code makes a call, it must also save the return address and, in general, any other already-used caller-saved registers. This process leads to additional memory operations, stack pointer adjustments, and additional jumps to/from the called function, with a consequent code size increase and degraded performance. RVfplib contains only leaf-functions (i.e., functions that do not make other function-calls). This property enables RVfplib to be independent of other external libraries, minimizing the extra code linked in the final binary. This is not the case for libgcc, as some of its functions depend upon `__clzsi2()` calls and the related table `__clz_tab`.
4. *Maximize potential compression*: The RISC-V C extension allows for compressing the most common RISC-V instructions when precise register patterns are used. For example, the majority of the instructions can be compressed when using registers from the *RVC* (i.e., registers in the set a0, a1, a2, a3, a4, a5, s0, s1). Since s0 and s1 are callee-saved registers, RVfplib does not use them.
5. *Register re-use*: Register allocation is optimized at function level to overcome heuristics of the compiler, whose analysis is mainly limited to the boundaries of basic blocks. As a basic rule, an operand is placed in the first free register; when it is no longer used, the register becomes free again.
6. *Performance vs. code size tradeoff*: Some RVfplib functions use loops to perform iterative processes. For example, the leading zeroes count after a numerical cancellation of an effective subtraction can be reduced to a shift-and-check loop, in which the result is left-shifted until the implicit one returns to its original position. This iterative process is convenient in terms of code size, but it is slow and inefficient. For this reason, it is also possible to use a bisection algorithm to count the leading zeroes, with better performance and increased code size. The choice can be taken at compile time. In general, when the taken-branch penalty is critical, unrolling the loop helps in maximizing the number of non-taken branches.

3.3 Comparison with libgcc

FP functions from libgcc use a complex set of hierarchical C macros to be as flexible and generic as possible. When compiling the library, it is possible to set specific high-level parameters to control how the library will treat exceptions, subnormals, roundings, etc. With the default settings, no exception is raised or handled, subnormal values are not flushed to zero, and the rounding mode is rounding to nearest, ties to even (RNE). Even with these minimalistic options, the generated code is sub-optimal in terms of size and performance.

In List. 1.1, we report the assembly code of `__eqsf2()` compiled with GCC 10.2.0 and optimized for size (`-Os`), together with comments and labels that we added to help the reader understand the code. This function, one of the smallest of the library, returns 1 if the inputs are not equal, and 0 if they are equal. The algorithm is straightforward:

1. If at least one input is NaN, return 1.
2. In the case of +0 and -0, return 0.
3. If the numbers are equal, returns 0; otherwise, return 1.

The libgcc function unpacks both the operands in their sign, exponent, and mantissa before starting the comparison. In `__eqsf2()`, this operation is unnecessary and is probably performed to adopt a common coding standard for the library design. Moreover, separately comparing sign, exponent, and mantissa improves the code readability but discards possible optimizations.

```

1 __eqsf2:
2 # Unpack operands, prepare checks
3     srli    a3,a0,0x17
4     lui     a5,0x800
5     addi    a5,a5,-1
6     srli    a2,a1,0x17
7     andi    a3,a3,255
8     li      a7,255
9     and     a6,a5,a0
10    srli    a4,a0,0x1f
11    and     a5,a5,a1
12    andi    a2,a2,255
13    srli    a1,a1,0x1f
14 # Check and compare
15 checks_0:
16     li      a0,1
17     bne     a3,a7,checks_1
18     bnez    a6,return
19     bne     a2,a3,return
20     beqz    a5,checks_2
21 return:
22     ret
23 checks_1:
24     beq     a2,a7,return

```

```

25     bne    a3,a2,return
26     bne    a6,a5,return
27 checks_2:
28     li     a0,0
29     beq    a4,a1,return
30     li     a0,1
31     bnez   a3,return
32     snez   a0,a6
33     ret

```

Listing 1.1: `__eqsf2()` disassembled `libgcc` code

In List. 1.2, we show the `__eqsf2()` function extracted from RVfplib. Writing in assembly allows to have a better control over the used instructions and registers when the functions are sufficiently small. All the checks are performed without unpacking the operands, and we opportunistically reuse the register `a5` to reach the desired outcome during the final `snez` comparison.

```

1  __eqsf2:
2     lui    a5,0xff000
3     # Check for NaNs
4     slli   a2,a0,0x1
5     bltu   a5,a2,end
6     slli   a3,a1,0x1
7     bltu   a5,a3,end
8     # Check for +0, -0
9     or     a5,a2,a3
10    beqz    a5,end
11    # Effective comparison
12    xor     a5,a0,a1
13    end:
14    snez    a0,a5
15    ret

```

Listing 1.2: `__eqsf2()` RVfplib code

We aimed to reach the same optimization level implementing the algorithm of List. 1.2 using C, and we managed to halve the code size of the `libgcc` function from 84 B to 42 B, showing the importance of choosing an optimized algorithm. However, the generated code is still 16% larger than the one generated from our assembly.

Forcing the compiler to reuse precise registers and take branches in a deterministic way is more natural in assembly than in C; during the compilation of our C function, the compiler creates unexpected intermediate operations and register moves, with negative effects on both code size and performance.

The same is true for the more complex functions of the library. Functions from `libgcc` are safe, generic, flexible, and parametric, but this comes at the expense of possible critical optimizations in key functions, where more precise control over the registers and the branch choices would be preferred. Assembly language helps consider a register as a container for a value, without a precise label and

meaning as in C; therefore, a more opportunistic usage of the registers comes more natural, without the need of forcing the compiler to behave in a precise way.

3.4 Testing

To test RVfplib, we relied on TestFloat [20], which provides an extensive IEEE 754 testing suite for generating test-cases and checking the correctness of custom FP implementations. Internally, TestFloat uses the fully IEEE 754 compliant SoftFloat library [19] as a golden reference. We generated the inputs for each function with the TestFloat engine and compared the function outputs with both SoftFloat and libgcc golden models. Since not all functions in RVfplib have a SoftFloat implementation, we used libgcc as a golden model when it was needed (e.g., for the “greater [or equal] to” functions).

4 Experimental setup

To analyze the impact of RVfplib, we evaluated its code size and performance metrics in both a synthetic environment and using real programs. In the first set of experiments, we extracted the code size of each function; in the second one, we evaluated the behavior of RVfplib on real benchmarks.

4.1 Benchmarks

Since we evaluate an FP library useful for area-constrained embedded devices, we selected all the Embench benchmark suite applications [4] that use FP numbers (`cubic`, `minver`, `nbody`, `st`, `ud`, `wikisort`). On the other hand, we selected three popular algorithms that can be run on small systems at the edge, on both single and double-precisions: a convolution (`conv`), a fast Fourier transform (`fft`), and a discrete wavelet transform (`dwt`).

4.2 Code Size

RVfplib implements most of the FP functions provided by libgcc and all the implicit arithmetic functions available in emFloat. Therefore, we evaluated the code size of the functions of our library and compared them against the two competitors. The code size of the emFloat functions is publicly available for RV32IMC ISA [17]; thus, we compiled both RVfplib and libgcc functions with the same target using GCC 10.3 and libgcc originally compiled with the `-Os` flag enabled, its default setting. The functions were linked to a fixed C program, and the code size of the functions extracted from its disassembly-dump. To create realistic conditions for embedded devices and avoid intricate dependencies and code size bloating, we always linked our programs against `libc_nano` and `libm_nano`. For a fair analysis, we compared RVfplib and libgcc since both are compiled for minimum code size and support subnormal values, and RVfplib.nd

with emFloat since both flush subnormal values to zero and target minimum code size as well.

Since libgcc is freely available, we extended our comparison linking our real benchmarks against RVfplib and RVfplib.nd first, and then libgcc. To measure the code size impact that the libraries have on the read-only memory footprint, we added the size of the .text and the .rodata sections. Since some programs use the FP division, we also measured their code size when linked against RVfplib with fast divisions (the double-precision one belongs to libgcc).

To complete the code size analysis, we measured the code size of the Arm-optimized libgcc FP library and compared it with the code size of both the generic RISC-V libgcc support and RVfplib.

4.3 Performance

On the performance side, a full profiling of RVfplib and libgcc was performed for both the single average latencies of the functions and the execution time of the benchmarks. In the following, when referring to a function, the term *latency* indicates the number of cycles required to execute it.

To evaluate the function latencies, we simulated a synthetic C program on the CV32E40P processor [6] with single-cycle latency memories using Mentor QuestaSim, repeating the experiment for each function of the compared libraries. The C program is composed of a loop that makes an explicit call to the function under test during each iteration and measures the latency of each function execution, including the jump/return to/from function cycles, and then averages the total cycle count on the number of iterations. Each function is fed with 10000 randomly generated values within (0,1), and the overhead of the load/store operations before and after the call is not considered. Using 1-cycle fixed-latency memories is a best-case scenario for libgcc performance, as libgcc accesses the stack inside its FP functions while RVfplib does not, as we avoided in-function memory requests. Additional memory latency/miss penalties negatively affect only the functions from libgcc.

We also compare our results to the average latencies reported by SEGGER emFloat [17]. It is unclear, however, whether this reported performance includes latency overheads from function calls and function returns. These overheads, as well as processor-specific branch- and jump penalties, can strongly affect performance, especially for small functions. SEGGER extracted performance metrics using a GigaDevice GD32VF103 [18], which is based on a variable 2-stage pipeline RISC-V core [7]. It is likely that the jump/branch penalties of CV32E40P (from 2 to 4 cycles) [11] are higher. Moreover, SEGGER only reports latency results of their “performance-optimized” emFloat library, which is different from the one used for the code size results. For this reason, we used our fast single-precision division and the double-precision division from libgcc to perform this comparison.

To provide insight into how RVfplib affects the execution time, we simulated our benchmarks with SPIKE, a RISC-V simulator for a simple processor that executes one instruction per cycle, and reported the different instruction counts

linking with `libgcc`, `RVfplib`, and `RVfplib_nd`. Since some benchmarks use the double-precision division, we also reported the execution times of the programs linked with `RVfplib` with fast divisions (the 64-bit division is taken from `libgcc`).

5 Results

5.1 Code Size

We show the code size of the single functions of `RVfplib`, `libgcc`, and `emFloat` in Table 1. Comparing the total code size of the libraries, we achieve a net gain of $\approx 60\%$ by replacing `libgcc` FP functions with the ones in `RVfplib`. In absolute terms, the memory savings reach 7.5 kB, which is a significant code size reduction, especially for small programs. The small embedded systems we target are area/memory size constrained and do not have hardware FPUs. Most commonly, they require performing computations on single-precision data. As such, our high code size reduction for the most frequent single-precision FP operations (i.e., addition, subtraction, multiplication), which is around 67% on average, is very significant. `libgcc` subtraction is automatically re-linked as a function different from the addition, even if their code is shared except for one initial sign change of the second operand. `RVfplib` subtraction flips the sign and then executes an addition, without any other jump that would cause extra latency.

Passing from `RVfplib` to `RVfplib_nd`, which flushes subnormal values to correctly signed zeroes, allows saving another 21.6% of the library code size. This significant gain comes for free when supporting subnormal numbers is not a requirement. `RVfplib_nd` is almost 5.3% smaller than `emFloat`, even if the double-precision division from `emFloat` is 30% smaller than the one from `RVfplib_nd`. The functions that gain the most from removing the subnormal support are multiplication and division, as the addition needs only small adjustments to process the denormalized inputs.

In Fig. 1, we summarize the code size results of our benchmarks. The code size savings on `libgcc` span from 16% of `cubic` (with `libgcc` double-precision division) to 60% (`st` with `RVfplib_nd`) and are relatively high also for large code size programs like `fft64`, which passes from almost 13.8 kB to 8.4 kB with more than 39% of saving. The average code size reductions with respect to `libgcc` are 39.3%, 36%, and 46.5% for `RVfplib`, `RVfplib` with `libgcc` fast divisions, and `RVfplib_nd`, respectively.

5.2 Performance

Average latencies of each function of `RVfplib` and `libgcc` FP support are summarized in Table 2. `RVfplib` functions are always faster than the ones from `libgcc`, except for the two small divisions, which are $1.45\times$ and $2\times$ slower for single and double-precision, respectively. This fact underlines the importance of trying to re-implement these operations, changing the core algorithm; nevertheless, `RVfplib`

Table 1: Code size comparison between RVfplib, libgcc FP support, RVfplib_nd, and emFloat. Only the functions implemented in RVfplib are reported. Target ISA: RV32IMC

Function	libgcc Name	Code Size [B]			
		RVfplib	libgcc	RVfplib_nd	emFloat
f32_add	__addsf3	320	804	274	410
f32_sub	__subsf3	6	818	6	10
f32_mul	__mulsf3	310	542	172	178
f32_div	__divsf3	294	590	188	184
		(416)*		(280)*	
f32_lt	__ltsf2	56	120	56	58
f32_le	__lesf2	60	120	60	54
f32_gt	__gtsf2	52	120	52	50
f32_ge	__gesf2	60	120	60	62
f32_eq	__eqsf2	36	84	36	44
f32_ne	__nesf2	-	-	-	-
f32_i32	__fixsfsi	58	96	58	74
f32_ui32	__fixunssfsi	50	88	50	50
f32_i64	__fixsfdi	120	136	120	146
f32_ui64	__fixunssfdi	80	100	80	98
i32_f32	__floatsisf	60	186	60	66
ui32_f32	__floatunsisf	48	154	48	52
i64_f32	__floatdisf	106	258	106	96
ui64_f32	__floatundisf	82	214	82	70
f32_f64	__extendsfdf2	88	150	56	64
f64_add	__adddf3	736	1542	572	724
f64_sub	__subdf3	6	1560	6	10
f64_mul	__muldf3	506	1080	288	286
f64_div	__divdf3	742	1334	396	278
		(1334)*		(1334)*	
f64_lt	__ltdf2	94	166	94	70
f64_le	__ledf2	96	166	96	70
f64_gt	__gtdf2	90	166	90	70
f64_ge	__gedf2	104	166	104	70
f64_eq	__eqdf2	60	106	60	52
f64_ne	__nedf2	-	-	-	-
f64_i32	__fixdfsi	62	100	62	84
f64_ui32	__fixunsdfsi	54	96	54	54
f64_i64	__fixdfdi	130	164	130	146
f64_ui64	__fixunsdfdi	94	126	94	96
i32_f64	__floatsidf	44	102	44	46
ui32_f64	__floatunsidf	32	78	32	34
i64_f64	__floatdidf	114	372	114	128
ui64_f64	__floatundidf	90	328	90	106
f64_f32	__truncdfsf2	158	284	104	130
Total		5100	12636	3994	4220

* RVfplib small divisions can be replaced with faster versions, yielding better performance and extra code size.

divisions do not use the hardware integer divider, allowing for more flexibility, and the division operation is not common in simple algorithms used on small embedded systems. The fast 32-bit division in RVfplib is slightly faster than one from libgcc, and the 64-bit one is the same. The single-precision comparisons and both the multiplications show important speedups (up to $2.57\times$ for the

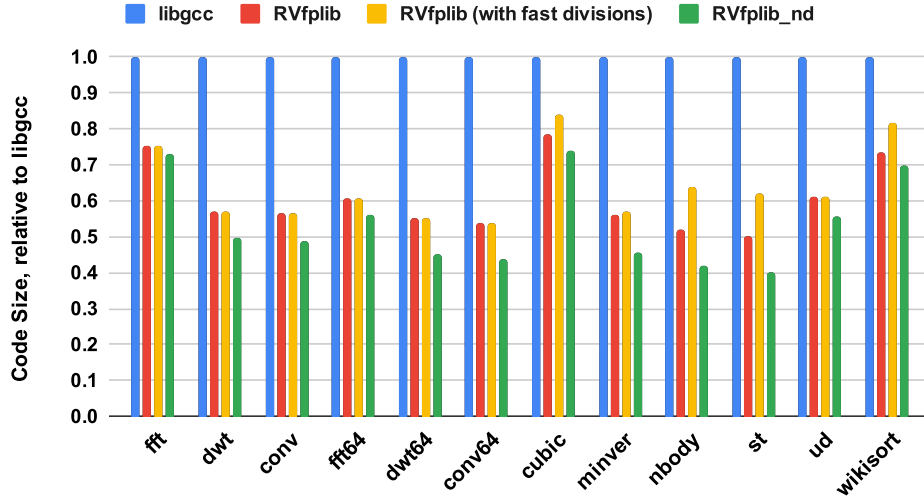


Fig. 1: Relative code size (.text+.rodata) of benchmarks linked against libgcc, RVfplib, RVfplib with fast divisions, RVfplib_nd. The reference is libgcc.

multiplication), and the single-precision addition in RVfplib is faster than the one from libgcc by more than $1.5\times$. These data are promising, as these operations are ubiquitous in almost every FP algorithm. The conversions from integers to FP numbers are the functions that obtain the highest speed gain, which peaks for converting a 64-bit unsigned integer to a double-precision FP value with more than $4\times$ lower latency. Replacing the whole set of libgcc functions with RVfplib gives an average speedup of $2\times$.

As already pointed out, making a comparison between RVfplib_nd and emFloat performance using the average latencies reported by SEGGER is not straightforward. We could not reproduce the experiment in the same conditions since they used a device that is likely to show a lower cycle count if compared to the CV32E40P core. Moreover, it is not specified whether the latency of the jumps to/from functions was taken into account. This is especially valid for the smaller functions, that can be strongly biased by the jump to/from function latency overhead. However, if we focus on the bigger functions, the double-precision addition in emFloat (the subtraction shares the code with the addition) and both divisions are faster than the ones from RVfplib by factors around $2.7\times$ and $1.9\times$, for single and double-precision, respectively.

When we measure the instruction count of the real benchmarks linked against RVfplib and libgcc, we obtain the data shown in Fig. 2. We chose these benchmarks to have a good mix of realistic examples, and we found for RVfplib and RVfplib_nd an average speedup of $1.5\times$ even if the benchmarks that use the double-precision division in RVfplib are actually slower than the ones linked against libgcc. In particular, **wikisort** uses the square root operation that uses the double-precision division, which is also used by **st**. In some benchmarks (e.g., **ud**), RVfplib_nd performance decreases because of its 64-bit addition, which does not have a fast-path for equal operands. All the other programs show high-speed gains

Table 2: Average latency comparison between RVfplib, libgcc FP support, RVfplib_nd, and emFloat. Only the functions implemented in RVfplib are reported.

Function	libgcc Name	Average Latency [cycles]			
		RVfplib	libgcc	RVfplib_nd	emFloat*
f32_add	<code>__addsf3</code>	50.6	79.5	52.1	49.5
f32_sub	<code>__subsf3</code>	72.9	114.7	72.4	62.2
f32_mul	<code>__mulsf3</code>	48	120	47	39.3
f32_div	<code>__divsf3</code>	252	190	252	67
		(182.2) [†]		(182.2) [†]	
f32_lt	<code>__ltsf2</code>	18	41	18	11
f32_le	<code>__lesf2</code>	17	41	17	10
f32_gt	<code>__gtsf2</code>	16	41	16	10
f32_ge	<code>__gesf2</code>	17	41	17	11
f32_eq	<code>__eqsf2</code>	14	26.5	14	10
f32_ne	<code>__nesf2</code>	-	-	-	-
f32_i32	<code>__fixsfsi</code>	18.5	21	18.5	14
f32_ui32	<code>__fixunssfsi</code>	16	23	16	13
f32_i64	<code>__fixsfdi</code>	22	48	22	23.2
f32_ui64	<code>__fixunssfdi</code>	18	44	18	18.9
i32_f32	<code>__floatsisf</code>	29.5	88.5	29.5	32.6
ui32_f32	<code>__floatunsisf</code>	22	78.7	22	33
i64_f32	<code>__floatdisf</code>	41.5	131.4	41.5	49.1
ui64_f32	<code>__floatundisf</code>	32.5	122	32.5	44.1
f32_f64	<code>__extendsfdf2</code>	19	33	18	14.1
f64_add	<code>__adddf3</code>	87.2	101.1	83.9	62.8
f64_sub	<code>__subdf3</code>	116.5	138.5	114.9	82.8
f64_mul	<code>__muldf3</code>	85	219	85	75
f64_div	<code>__divdf3</code>	769.5	382.2	769.5	197.2
		(382.2) [†]		(382.2) [†]	
f64_lt	<code>__ltdf2</code>	27	46.2	27	16
f64_le	<code>__ledf2</code>	26	46.2	26	16
f64_gt	<code>__gtdf2</code>	25	46.2	25	16.1
f64_ge	<code>__gedf2</code>	27	46.2	27	16.1
f64_eq	<code>__eqdf2</code>	24	30.5	24	14
f64_ne	<code>__nedf2</code>	-	-	-	-
f64_i32	<code>__fixdfsi</code>	19	19	19	16.8
f64_ui32	<code>__fixunsdfsi</code>	17	21	17	13.8
f64_i64	<code>__fixdfdidi</code>	33	38.4	33	26.9
f64_ui64	<code>__fixunsdfdidi</code>	30.5	43	30.5	21.5
i32_f64	<code>__floatsidf</code>	24.9	61	24.9	31.6
ui32_f64	<code>__floatunsidf</code>	18.5	53	18.5	23.9
i64_f64	<code>__floatdidf</code>	41	142	41	45.1
ui64_f64	<code>__floatundidf</code>	31	134	31	39.3
f64_f32	<code>__truncdfsf2</code>	25	61	28	25.1

* emFloat data were obtained from [17]. It is unclear whether reported numbers include function call / function return latency overheads.

[†] RVfplib small divisions can be replaced with faster versions, yielding better performance and extra code size.

thanks to the massive use of multiplications and additions. For RVfplib with fast divisions, the average speedup grows to $1.6\times$, and the programs linked with RVfplib are always faster than the ones linked with libgcc.

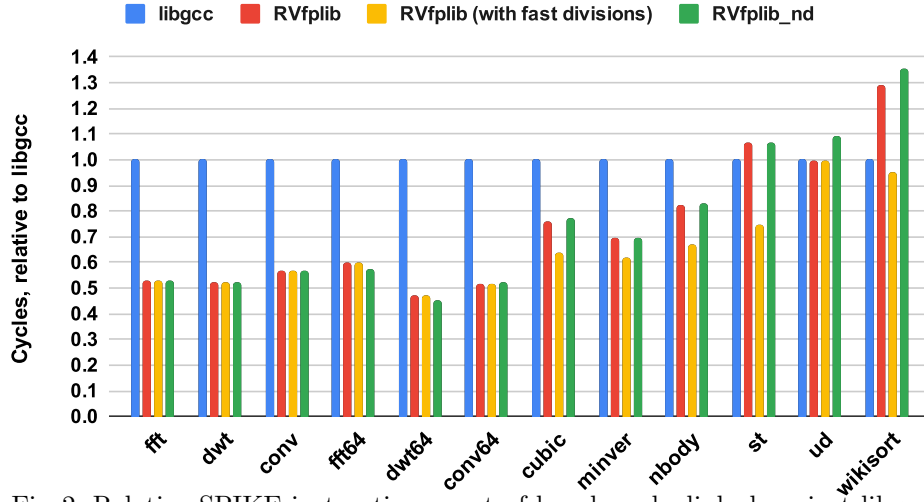


Fig. 2: Relative SPIKE instruction count of benchmarks linked against libgcc, RVfplib, RVfplib with fast divisions, RVfplib_nd. The reference is libgcc.

5.3 Comparison with Arm

In Fig. 3, we show the code size comparison between the generic RISC-V libgcc FP support, RVfplib, and the Arm-optimized libgcc FP support. RVfplib brings the existing FP library code size gap between RISC-V and Arm from 8376 B to 840 B (10 \times less), reducing the Arm to RISC-V code size inflation from 196.6% to 19.7%. Arm addresses many comparison-function calls to a generic *compare*, reducing the total number of implemented functions and the library code size. This choice can be implemented in future versions of RVfplib as well.

6 Further improvements

RVfplib will be released as an open-source project under GPL license, and everyone will be allowed to contribute to its enhancement, improving and extending it. SEGGER results unequivocally show that the 64-bit addition in RVfplib can be further improved to decrease its average latency. Both the divisions can reach increased code size and performance, maybe with different algorithmic choices and exploiting the hardware divider. The optimal solution would be to offer both a version that exploits the divider and one independent from it. On the other hand, the library misses important functions, such as the square root and the trigonometric ones, to be more versatile and further save precious memory space and cycle counts. As already evaluated in [13], hardware support for Count Leading Zeroes (CLZ) helps in speeding up the FP functions (e.g., addition, truncation) and can also decrease their code size, replacing a block of instructions with only one. Such support is already present in the PULP extension and in the draft of the RISC-V B extension [16]. Another improvement to further save code size would be merging in common functions the repeated code for dealing with

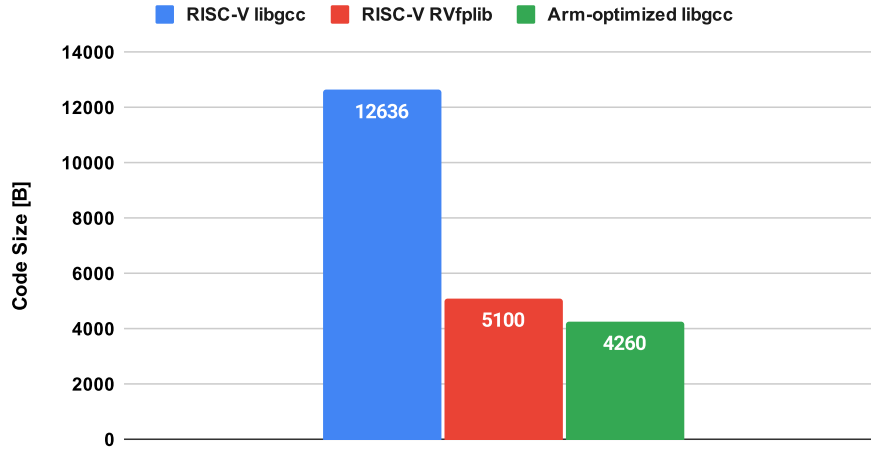


Fig. 3: FP libraries code size comparison between RISC-V generic libgcc, RVfplib, Arm-optimized libgcc.

subnormals/special cases, especially when such input patterns are uncommon, and various comparison into one.

7 Conclusion

In this paper, we presented RVfplib, the first open-source assembly-optimized FP emulation library for RISC-V small integer-only processors. The library implements the primary and most common single and double-precision FP operations like addition, subtraction, multiplication, division, comparisons, conversions, and adopts the same interface as libgcc to be easily linked by GCC against real programs without any source-code modification. The library follows IEEE 754 standard guidelines for encodings and computations, with only minor and easily modifiable differences. RVfplib is smaller than the libgcc FP support by almost 60% and, on average, $2\times$ faster. We showed that, on real benchmarks, RVfplib reduces the code size by 39% and speeds up the execution by $1.5\times$ on average, even when considering benchmarks that heavily use the less optimized functions in RVfplib. If compared to the Arm-optimized libgcc library, RVfplib reduces the Arm to RISC-V code size inflation from 196.6% (vs. RISC-V general libgcc FP support) to 19.7%. We also presented RVfplib_nd, which treats subnormal values as correctly signed zeroes, and shown that its code size is 5.3% smaller than the SEGGER emFloat FP library, the only available RISC-V optimized FP emulation library, which is closed-source and treats subnormal values in the same way.

References

1. Arm Keil - fplib, https://www.keil.com/support/man/docs/armlib/armlib_chr1358938941317.htm
2. Bertaccini, L., Perotti, M., Mach, S., Schiavone, P.D., Zaruba, F., Benini, L.: Tiny-fpu: Low-cost floating-point support for small risc-v mcu cores. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1–5 (2021). <https://doi.org/10.1109/ISCAS51556.2021.9401149>
3. Code size reduction official sub-committee, <https://lists.riscv.org/g/tech-code-size>
4. Embench benchmark suite, <https://github.com/embench/embench-iot>
5. FLIP library, <http://flip.gforge.inria.fr/>
6. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10), 2700–2713 (2017). <https://doi.org/10.1109/TVLSI.2017.2654506>
7. GigaDevice Semiconductor Inc.: GD32VF103 RISC-V 32-bit MCU User Manual, Revision 1.2 (October 2019)
8. Gottscho, M., Alam, I., Schoeny, C., Dolecek, L., Gupta, P.: Low-cost memory fault tolerance for IoT devices. ACM Trans. Embed. Comput. Syst. **16**(5s) (Sep 2017). <https://doi.org/10.1145/3126534>
9. libgcc - Arm floating-point support, <https://github.com/gcc-mirror/gcc/blob/master/libgcc/config/arm/ieee754-sf.S>
10. libgcc library, <https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html>
11. OpenHW Group CV32E40P user manual, https://core-v-docs-verif-strat.readthedocs.io/projects/cv32e40p_um/en/latest/index.html
12. Perotti, M., Davide, P.S., Tagliavini, G., Rossi, D., Kurd, T., Hill, M., Yingying, L., Benini, L.: HW/SW approaches for RISC-V code size reduction (2020). <https://doi.org/10.3929/ethz-b-000461404>
13. Pimentel, J.J., Bohnenstiehl, B., Baas, B.M.: Hybrid hardware/software floating-point implementations for optimized area and throughput tradeoffs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(1), 100–113 (2016)
14. Qfplib library, <https://www.quinapalus.com/qfplib.html>
15. RISC-V - Arm comparison, Embench, <https://riscv.org/wp-content/uploads/2019/12/12.10-12.50a-Code-Size-of-RISC-V-versus-ARM-using-the-Embench%E2%84%A2-0.5-Benchmark-Suite-What-is-the-Cost-of-ISA-Simplicity.pdf>
16. RISC-V Bit-Manipulation extension draft, <https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-0.92.pdf>
17. SEGGER emFloat, <https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/>
18. SEGGER wiki, https://wiki.segger.com/SEGGER_Floating-Point_Library
19. SoftFloat, <http://www.jhauser.us/arithmetic/SoftFloat.html>
20. TestFloat, <http://www.jhauser.us/arithmetic/TestFloat.html>