FloppyFloat: An Open Source Floating Point Library for Instruction Set Simulators

Niko Zurstraßen, Nils Bosbach, Rainer Leupers

RWTH Aachen University, Institute for Communication Technologies and Embedded Systems

***Abstract*—Instruction Set Simulators (ISSs) are important software tools that facilitate the simulation of arbitrary compute systems. One of the most challenging aspects of ISS development is the modeling of Floating Point (FP) arithmetic. Despite an industry standard specifically created to avoid fragmentation, every Instruction Set Architecture (ISA) comes with an individual definition of FP arithmetic. Hence, many simulators, such as gem5 or Spike, do not use the Floating Point Unit (FPU) of the host system, but resort to soft float libraries. These libraries offer great flexibility and portability by calculating FP instructions by means of integer arithmetic. However, using tens or hundreds of integer instructions to model a single FP instruction is detrimental to the simulator's performance.**

Tackling the poor performance of soft float libraries, we present *FloppyFloat* - an open-source FP library for ISSs. FloppyFloat leverages the host FPU for basic calculations and rectifies corner cases in software. In comparison to the popular Berkeley SoftFloat, FloppyFloat achieves speedups of up to $5.5\times$ for individual instructions. As a replacement for SoftFloat in the RISC-V golden reference simulator Spike, FloppyFloat accelerates common FP benchmarks by up to $1.41\times$.

***Index Terms*—Floating Point, Full-System Simulators, Instruction Set Simulators, Virtual Platforms**

I. Introduction

With the development of the first computers, engineers were challenged with finding a way to represent and calculate real numbers. Opposed to integer numbers, a solution to this challenge is not straightforward, which led to a multitude of different approaches in practice. With the emergence of ever newer compute systems and Instruction Set Architectures (ISAs), the development of portable numerical software turned into a complex matter. In order to curb the growing fragmentation, the industry standard IEEE 754 [19] was created in 1985. This standard defines Floating Point (FP) formats and instructions, and is followed by all relevant ISAs, such as RISC-V, x86, or ARM. Nevertheless, there are many discrepancies between these architectures in terms of FP arithmetic. This is partly due to the standard itself, which specifies many aspects, such as the encoding of a canonical quiet Not a Number (qNaN), as implementation-defined. But also the ISAs have contributed to this by introducing non-IEEE 754-adhering peculiarities. For example, x86 SSE features a flush-to-zero mode, which is not compliant with IEEE 754.

These differences are particularly challenging for Instruction Set Simulators (ISSs). ISSs enable the functional simulation of arbitrary ISAs and are often paired with hardware models, such as UARTs or cache systems, which is then described as a Full-System Simulator (FSS) or Virtual Platform (VP). FSSs can be used, for example, to simulate an ARM-based target system, including peripherals, on an x86-based host system. As previously described, the host's Floating Point Unit (FPU) cannot be used without further ado by a simulator, as its behavior may be different from the simulated target's FPU. This is why many popular simulators, such as gem5 [7] or Spike [11], use soft float libraries. Soft float libraries are usually written in a high-level language, such as C or C++, and implement FP arithmetic solely through the use of integer data types and arithmetic. The libraries can easily be configured to resemble a given target ISA. Another advantage is that the host system does not require a FPU, as all calculations are performed using integer arithmetic. The only but severe disadvantage of soft float libraries is their poor performance. Depending on the instruction, soft float can be up to $80\times$ slower than a comparable instruction on a FPU [31]. Especially for high-performance simulators, soft float libraries quickly become the simulator's bottleneck.

This work addresses the poor performance of soft float by presenting *FloppyFloat*[1]- an open-source FP library for ISSs. To the best of our knowledge, FloppyFloat is the first of its kind. FloppyFloat combines the flexibility of soft float with the performance of a native FPU. This is achieved by using the host FPU for basic calculations and doing rectifications in software. The requirements for the host FPU are minimal and met by virtually all ISAs. In particular, FloppyFloat does not rely on the host FPU to provide exception flags, rounding modes, tininess detection, or other mandatory IEEE 754 features. Simply a correct result under the default "RoundTiesToEven" rounding mode suffices for FloppyFloat. Currently, FloppyFloat is able to mimic x86, ARM, and RISC-V FP arithmetic on most host systems. With the features of these ISAs implemented and tested, FloppyFloat can be extended to other ISA with relatively little effort. To demonstrate the practical benefits of FloppyFloat, we replaced the popular Berkeley SoftFloat [15] library in the official RISC-V golden reference simulator Spike [7] by FloppyFloat. Executing an exhaustive set of FP benchmarks showed that FloppyFloat always outperforms Berkeley SoftFloat. Benchmarks, such as SPEC CPU 2017 [2], attain speedups of up to 41%. In a one-to-one comparison for individual instructions without a simulator environment, FloppyFloat outperforms Berkeley SoftFloat by up to $5.5\times$.

[1]https://github.com/not-chciken/FloppyFloat

978-3-9826741-0-0/DATE25/© 2025 EDAA

TABLE I
COMPARING THE FP CHARACTERISTICS OF INDIVIDUAL ISAS.

ISA	32-bit qNaN	Class	Exception Flags	FTZ	NB	NP	Rounding Modes	Tininess	$fma(0, \infty, qNaN)$
ARM	0x7fc00000	-	✓	✓	-	✓[8]	4	BR[9]	invalid[10]
LoongArch	0x7fc00000	FCLASS	✓	-	-	✓	4	AR	undefined
MIPS64	0x7fc00000[1]	CLASS	✓	✓	-	✓	4	AR	undefined
OpenRISC 1000	undefined	CLASS[3]	✓	-	✓	-	4	BR	undefined
PowerPc	0x7fc00000	FPSCR_FPRF[4]	✓[5]	✓[7]	-	✓	4	BR	invalid
RISC-V	0x7fc00000[2]	FCLASS	✓	-	✓	-	5	AR	invalid
x86 SSE	0xffc00000	-	✓[6]	✓	-	✓	4	AR	valid

AR = After Rounding, BR = Before Rounding, FTZ = Flush-To-Zero, NB = NaN Boxing, NP = NaN Propagation
1: With $FCSR_{NAN2008}$=0 the canonical qNaN changes to 0x7fbfffff; 2: Canonical qNaN was 0x7fffffff before version 2.1 of F/D;
3: sNaN and qNaN can also be detected by the FPCSR; 4: Implicit via $FPSCR_{FPRF}$ register;
5: Has an additional non-sticky exception flags in the FPSCR register; 6: Has an additional denormal operand exception;
7: Only enabled if NI flag in FPSCR is set; 8: NaN propagation can be disabled using FPCR.DN;
9: With FEAT_AFP, FPCR.AH allows to change tininess detection to after rounding; 10: With FEAT_AFP, FPCR.AH allows to change the validity;

II. BACKGROUND

A. IEEE 754 Basics

After its publication in 1985, the IEEE 754 [19] FP standard has become the de facto standard for all modern ISAs. While IEEE 754 defines several types of FP numbers, the most widespread formats are binary FP numbers, with binary32 and binary64 in particular. These formats are also available in most programming languages under names such as float, f32, float32, and double, f64, float64 respectively. They describe a number by a sign, a significand, and an exponent (see Figure 1). The real number represented by the FP number can be calculated as;

$$f = (-1)^{sign} \cdot (1.s_{p-1}s_{p-2}...s_1)_2 \cdot 2^{exponent-bias} \quad (1)$$

If the exponent reaches the lowest possible value, the representation switches to so-called subnormal numbers. Here, the implicit leading 1 turns into 0:

$$f = (-1)^{sign} \cdot (0.s_{p-1}s_{p-2}...s_1)_2 \cdot 2^{e_{min}} \quad (2)$$

This is one of the most controversial IEEE 754 features, as it significantly complicates FP calculations. However, subnormal numbers entail helpful mathematical properties, which are also used later in this work in Section IV. Besides subnormal numbers, IEEE 754 leaves part of the encoding space for other exceptional values, such as infinity and Not a Number (NaN). To direct the possible rounding errors in FP instructions, the most recent standard defines five different rounding modes. In addition to the arithmetic result of an operation, IEEE 754 also mandates five sticky exception flags which indicate irregularities during an instruction's execution. In particular, the flags comprise the following exceptions: invalid, inexact, overflow, underflow, and divideByZero. Sticky means that once set, the flags remain set until reset by a special instruction.

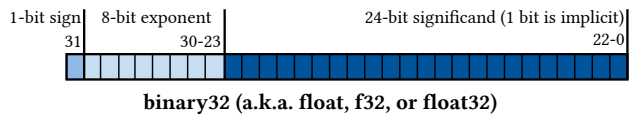

Fig. 1. Bit fields of a 32-bit binary FP value adhering to IEEE 754.

B. Differences Among Architectures

As mentioned in the previous section, the simulation of FP arithmetic is complicated by a plethora of different implementations. In this section, the individual differences and challenges are highlighted in greater detail. Table I provides an overview of popular ISAs and their FP characteristics. Although all of the listed ISAs follow the IEEE 754 standard, there are no ISAs which are exactly alike.

There are three reasons for these differences. Firstly, the IEEE 754 standard leaves room for implementation-defined behavior. That includes, for example, the encoding of a canonical qNaN. This value is generated when an invalid instruction, such as $\infty - \infty$, is executed. However, as the exact encoding is not specified, different implementations are seen in practice. The different qNaN encodings for a 32-bit FP type are given in Table I.

In addition to implementation-defined behavior, the IEEE 754 standard has also been revised twice in its history (2008 [20] and 2019 [21]), whereby some definitions were significantly changed. For example, the maximum and minimum instruction were affected by revisions. These instructions were not included in the 1985 standard, which is why x86 created its own definition with the introduction of SSE in 1999. This situation was rectified with IEEE 754 2008, which was followed by initial versions of the RISC-V ISA. However, the definition included erroneous behavior which negatively impacted the instruction's associativity [18]. This mistake was corrected 11 years later with the 2019 standard. RISC-V then also changed the definition of the maximum/minimum function and thus follows the 2019 standard since version 2.2 of the F and D extension.

In addition to mandatory IEEE 754 features, some ISAs also include non-compliant features. A frequent, but non-compliant feature are Flush-To-Zero (FTZ) modes, which are present in ARM, MIPS64, PowerPC, and x86 SSE. With FTZ enabled, the FPU rounds subnormal numbers to zero. This may increase the FPU's performance, as handling subnormal numbers is costly on many systems.

Besides regarding the characteristics of an ISA as a whole, particular implementations may also introduce bugs, and thus

an individual FP behavior. The most famous case involves wrong results of the division (FDIV) instruction on early Intel Pentium processors in the 1990s. A more recent example are the XuanTie C906 and XuanTie C910 RISC-V processors, which fail to raise the underflow flag in certain corner cases [3].

III. Related Work

FP arithmetic in compute system simulation is a relatively small area of research, with only a handful of publications and open-source projects. The most popular soft float library Berkeley SoftFloat [15] was developed by J. Hauser in 1996 and is still maintained today. It is used in many open-source projects, such as Bochs [22], [24] gem5 [7], Spike [11], RISC-V VP University of Bremen [16], and QEMU [4]. Other soft float libraries include FLIP by C.-P. Jeannerod et al. [6] from 2004, and F. Bellards's SoftFP [5] from 2018. The first academic work addressing the poor performance of soft float libraries is a work by Guo et al. [13] from 2016. As later shown in Section IV, modeling the FP exception flags, especially the inexact flag, is one of the most difficult challenges to overcome when using the host's FPU. However, as defined by the IEEE 754 standard, FP exception flags are sticky. That means once set, the exception flags remain set until cleared. Guo et al. observed that most applications set the inexact exception flag after a few instructions, and never clear it. Consequently, the majority of the FP instructions can be simulated without requiring frequent recalculations of the inexact flag. Only in corner cases, such as non-default rounding modes, their method falls back to a soft float implementation. This idea, with a few modifications, was picked up in a work by Cota et al. [8] in 2019, who also contributed their implementation to QEMU's code base. Also in 2019, a work of You et al. [29] improved on the work by Guo et al. by calculating the inexact exception flags and rounding modes for additions with little overhead. This was achieved by using the Fast2Sum by Dekker [10], which was conceived decades earlier in 1971. A work of Zurstraßen et al. [31] in 2023, showed how algorithms similar to the Fast2Sum algorithm can be used to calculate inexact flags and rounding modes for other arithmetic instructions. Although methods for faster calculation of FP instructions have been published, there are no open-source projects that consolidate them. Thus, FloppyFloat is the first open-source library for fast FP arithmetic in ISSs.

IV. Methods

In this section we present the methods and C++ implementation of FloppyFloat. Particularly, we highlight our approach for 32-bit FP addition as well as FP-to-integer conversions. For all instructions, FloppyFloat strives to be faster than soft float, while still offering the same flexibility and portability. Hence, FloppyFloat uses standard C++ datatypes, and does not rely on third-party libraries, inline assembly, or compiler-specific features.

A. 32-bit FP Addition

The implementation of a 32-bit FP addition under RoundTiesToAway rounding is presented and discussed in this subsection. This rounding mode was introduced with IEEE 754 2008 and is therefore not available on predating ISAs, such as x86 SSE. The C++ implementation of the corresponding instruction is depicted in Code 1.

The function starts with a simple FP addition in Line 11. This addition is performed under the default C++ rounding mode of RoundTiesToEven. In the subsequent lines, this preliminary result is rectified and FP exception flags are set. This includes checking the result for infinity and NaN with the function `IsInfOrNan`. The function exploits characteristics of infinity and NaN to quickly check for these values in a few instructions. As shown in Line 14 to Line 28, this may result in cases where the invalid or inexact flag has to be set, and where NaN values are generated or propagated. Nevertheless,

```
1  constexpr bool IsInfOrNan(f32 a) {
2    return (a - a) != (a - a);
3  }
4
5  constexpr f32 ClearSignificand(f32 a) {
6    auto u = std::bit_cast<u32>(a);
7    return std::bit_cast<f32>(u & 0xff800000u);
8  }
9
10 f32 FloppyFloat::Add(f32 a, f32 b) {
11   f32 c = a + b;
12
13   if (IsInfOrNan(c)) [[unlikely]] {
14     if (IsInf(c)) {
15       if (!IsInf(a) && !IsInf(b)) {
16         overflow = true;
17         inexact = true;
18       }
19       return c;
20     }
21     if (IsInf(a) && IsInf(b)) {
22       invalid = true;
23       return GetQnan();
24     }
25     if (IsSnan(a) || IsSnan(b))
26       invalid = true;
27     if (IsNan(a) || IsNan(b))
28       return PropagateNan(a, b);
29   }
30
31   if(std::fabs(b) > std::fabs(a))
32     std::swap(a, b);
33   f32 r = (c - a) - b;
34
35   if (r != 0.f32) {
36     inexact = true;
37     f32 cc = ClearSignificand(c);
38     f32 r_scaled = r * 16777216.0f32;
39     if (cc == -r_scaled) [[unlikely]] {
40       u32 cu = std::bit_cast<u32>(c);
41       if (r < 0.f32 && c > 0.f32) {
42         c = std::bit_cast<f32>(cu + 1);
43         overflow = IsInf(c) ? true : overflow;
44       } else if (r > 0.f32 && c < 0.f32) {
45         c = std::bit_cast<f32>(cu - 1);
46         overflow = IsInf(c) ? true : overflow;
47       }
48     }
49   }
50
51   return c;
52 }
```

Code 1. C++ code for an addition under RoundTiesToAway rounding.

these cases are extremely rare in practice, as NaN and infinite values usually indicate poorly designed algorithms. As shown in a data study with 78 FP benchmarks [30], NaN or infinity values appear approximately once in 20,000 instructions on average. In the majority of FP benchmarks, these values do not appear at all. Next, in Line 31-33, the Fast2Sum algorithm [25] is used to calculate the residual/error of the addition. This algorithm states that the exact residual r of an addition can be calculated as (the $\circ$ operator represents rounding under RoundTiesToEven):

$$\begin{aligned} a + b + r = s &= \circ(a + b) \\ r &= \circ(\circ(s - a) - b) \quad with\ |a| > |b| \end{aligned} \tag{3}$$

Although, the Fast2Sum may require branching, it is less prone to overflows compared to alternatives like the 2Sum algorithm. A non-zero residual has two implications. First, it implies that the addition was inexact and the corresponding flag has to be set (Line 36). Furthermore, due to the inexactness, rounding took place. When modeling the RoundTiesToAway rounding mode, discrepancies to the host's RoundTiesToEven may occur whenever the residual lies exactly between two FP values. To identify these tie situations, the significand of the addition's result is cleared in a first step (Line 37). As shown in the function `ClearSignificand`, this simply involves bitmasking the the result. Next, the residual is scaled by 2^p ($2^{24} = 16,777,216$ for 32-bit FP), with p being the number of significand bits (Line 38). Assuming the multiplication does not overflow, the result is guaranteed to be exact as 2^p is a power of two. In Line 39, it is checked whether the result with the cleared significand and scaled residual are equal in magnitude and have opposite signs. If true, the residual lies exactly between two FP numbers. Note that this method may fail to identify ties, whenever the addition's result is a power of two. For example, assume the rounded result of 16-bit FP addition is 1024, while the exact result would be 1023.75. This lies exactly between two FP values, as the next lower FP is 1023.5. Clearing the significand of the result does not change it and yields 1024. Scaling the residual of 0.25 with $-2^p = -2^{11} = -2048$ results in -512, which differs from 1024 both in sign and magnitude. However, for all such cases, the rounding performed by RoundTiesEven is equal to the rounding by RoundTiesToAway as the power of two is always the even result. In the given scenario, both rounding modes would round 1023.75 to 1024. Consequently, there is no rectification needed for these cases.

In case our method identifies a tie (Line 39), the next step is to check whether the result was rounded towards zero. This can be achieved by simply checking the sign bit of the residual and the result. For instance, if the residual is negative and if the result is positive, the result was rounded towards zero (Line 41). To rectify the result, it has to be set to the next greater FP value. As shown in Line 42, calculating the next greater value is a matter of reinterpreting the FP value as an unsigned integer and adding one. This method only works if the value to be increased is guaranteed to be positive and finite. Both assumptions hold valid in the given case. Note that issues with ± 0 are avoided as c can never be ± 0 if there is non-zero residual. This characteristic of FP arithmetic is a consequence of subnormal numbers. Lastly, the rectified result needs to be checked for overflows, which is shown in Line 43. Underflows do not pose a problem, as additions are always guaranteed to not underflow. Again, a characteristic that follows from the availability of subnormal numbers.

B. *f32-to-i32 Conversion*

In this subsection we present a 32-bit FP to 32-bit integer conversion for arbitrary rounding modes. The function, which is given Code 2, starts with checking the FP input for values that cannot be represented in the integer domain. This includes a check for NaN (Line 3) and out of range values (Line 6, Line 9). Since IEEE 754 does not specify how these values have to be mapped, every ISA comes with its own definition. For instance, RISC-V converts a NaN to an integer value of $2^{p-1} - 1$, while x86 SSE converts it to -2^{p-1}. Note that the out-of-range checks apply regardless of the used rounding mode as 32-bit FP numbers can only represent integer values in the given magnitude. For instance, the largest 32-bit signed integer value is $2^{31} - 1$. The first 32-bit FP value exceeding this range is 2^{31}. This is similar for negative values, where the largest negative 32-bit signed integer value is -2^{31}. The first 32-bit FP value exceeding this range is $-2^{31} - 2^8$.

After the initial checks, the conversion operation is performed. Using C++ and its standard library, this operation can be performed under RoundTiesEven (Line 16), RoundTiesTo-

```
1  template <RoundingMode rm>
2  i32 FloppyFloat::F32ToI32(f32 a) {
3    if (IsNan(a)) [[unlikely]] {
4      invalid = true;
5      return LimitNan<i32>();
6    } else if (a >= 2147483648.f32) [[unlikely]] {
7      invalid = true;
8      return LimitMax<i32>();
9    } else if (a < -2147483648.f32) [[unlikely]] {
10     invalid = true;
11     return LimitMax<i32>();
12   }
13
14   i32 ia;
15   if constexpr (rm == RoundTiesToEven) {
16     ia = std::lrint(a);
17   } else if constexpr (rm == RoundTiesToAway) {
18     ia = std::lround(a);
19   } else {
20     ia = static_cast<i32>(a);  // = RoundTowardZero
21   }
22
23   f32 r = static_cast<f32>(ia) - a;
24   if (r != 0.f32)
25     inexact = true;
26
27   if constexpr (rm == RoundTowardNegative) {
28     if (r > 0.f32)
29       --ia;
30   } else if constexpr (rm == RoundTowardPositive) {
31     if (r < 0.f32)
32       ++ia;
33   }
34
35   return ia;
36 }
```

Code 2. C++ implementation for a f32-to-i32 conversion under arbitrary rounding modes.

Away (Line 18), and RoundTowardZero (Line 20). We will later show how the results of these rounding modes can be used to model the missing rounding modes RoundTowardNegative and RoundTowardPositive.

Next, the inexactness of the conversion is determined by using a similar residual approach as in the 32-bit FP addition (Line 23). The residual is calculated by first casting the integer result back to FP. Note that this operation is always exact as the integer result was derived from a FP number. Subsequently, the input value a is subtracted, leaving only the exact residual of the initial f32-to-i32 conversion. This residual is exact due to Sterbenz' Lemma [27], which states that:

$$a - b = \circ(a - b) \quad \text{if} \quad b/2 \leq a \leq 2b \tag{4}$$

Except for two corner cases, the condition $(b/2 \leq a \leq 2b)$ always holds true. The first corner case is $ia = 0$ with $a \neq 0$. However, as one operand is zero, the subtraction is guaranteed to be exact. The second corner case is $|ia| = 1$ with $|a| < 0.5$. However, this situation cannot occur with the used rounding modes. Ultimately, if the residual is unequal to zero, the inexact exception flag set is set.

As a last step, the result has to be rectified in case RoundTowardNegative or RoundTowardPositive rounding is used. Since the intermediate result was calculated using RoundTowardZero, it suffices to check the sign of the residual and in-/decrement it by one. For instance, if the residual was positive, the conversion operation rounded to FP towards positive infinity. However, if RoundTowardNegative shall be modeled the intermediate result has to be decremented.

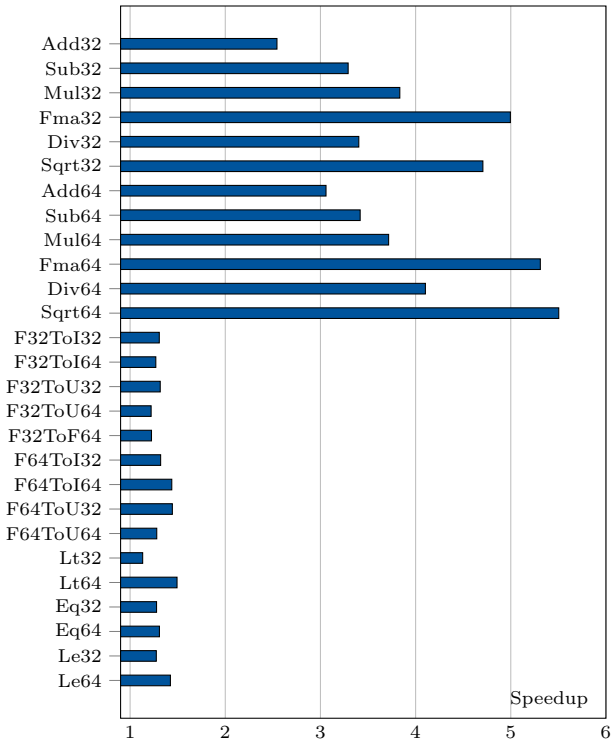

Fig. 2. Speedup of FloppyFloat. Berkeley SoftFloat was used as reference.

V. Results and Discussion

In the following two subsections, the performance characteristics of FloppyFloat are presented and discussed. Subsection V-A shows a direct comparison of FloppyFloat against Berkeley SoftFloat for individual instructions. To see how individual speedups translate into faster simulations, Subsection V-B shows the performance results of replacing Berkeley SoftFloat in the Spike simulator by FloppyFloat. All of the following experiments were conducted on an AMD 3990X x86 host system.

A. FloppyFloat vs. SoftFloat

In this subsection, we compare the performance of individual instructions in FloppyFloat and Berkeley SoftFloat. The performance results for RISC-V-on-x86 are shown in Figure 2. For the experiments, the individual instructions of the libraries were repeatedly executed, with their total execution times being measured. As the performance of both libraries significantly depends on the input data, the inputs were designed to always favor the fastest and also most common path. That means values, such as infinity or NaN are avoided, and results never underflow. Moreover, all exception flags were set to true at the start of execution, and instructions were executed under the most common rounding modes. That means arithmetic instructions always use the RoundTiesToEven rounding mode, while float-to-integer instructions always use the RoundTowardZero rounding mode. As can be seen in the Figure, FloppyFloat is faster than Berkeley SoftFloat in all cases. Especially for complex arithmetic instructions, such as a 64-bit square root instruction, FloppyFloat is more than $5\times$ faster than Berkeley SoftFloat. For lightweight datatype conversion instructions, speedups vary between $1.2\times$ and $1.5\times$. Note that these speedups are mostly independent of the modeled target systems, as the most frequently executed path is the same for all ISAs. Much more decisive are the microarchitectural differences between host CPU and host FPU. To illustrate this, we also ran the experiment shown in Figure 2 on a VisionFive2 (RISC-V) and a Mac mini M2 (ARM) host system. Again, FloppyFloat was faster in all cases, but there were significant differences in the individual speedups. While the Sqrt64 operation on the AMD 3990X host achieved a speedup of $5.5\times$, it was $2.57\times$ on the Max mini M2 and only $1.3\times$ on the VisionFive2.

B. FloppyFloat in Spike

To assess the actual performance benefits of FloppyFloat for FSS users, we integrated our library into the official RISC-V golden reference simulator Spike [11], and executed a comprehensive set of FP benchmarks. This includes aobench [1], fbench [28], FinanceBench [12], Himeno [17], mibench [14], smallpt [26], SPEC CPU 2017 [2], STREAM [23], and whetstone [9]. The benchmarks were statically compiled for 64-bit RISC-V with extensions I, M, A, F, D, and C. Note that some benchmarks are not included, as they faced problems with Spike's proxy kernel. Since FloppyFloat is designed as a drop-in replacement for soft float libraries, integrating FloppyFloat

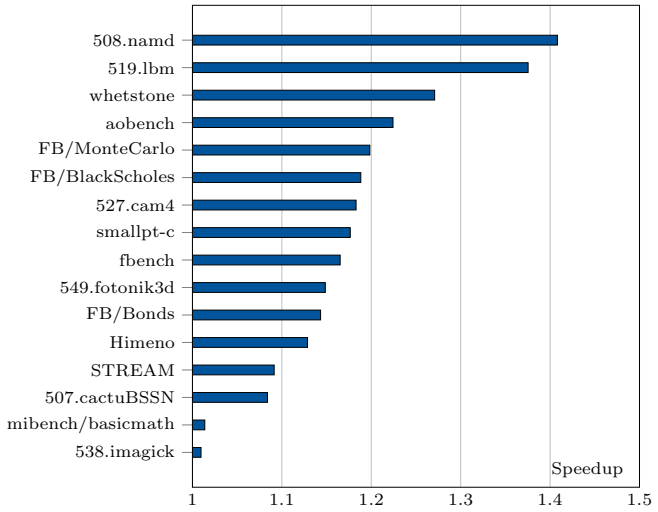

Fig. 3. Speedup of Spike with FloppyFloat. Spike with its default Berkeley SoftFloat was used as reference.

into Spike is matter of hours at most. The performance results of the benchmark executions are depicted in Figure 3. Overall, a heterogeneous performance picture emerges, as the attainable speedups highly depend on executed instructions in a given benchmark. For instance, mibench's basicmath attains a speedup of $1.01\times$ with 0.7% of all executed instructions being FP instructions. 508.namd from SPEC CPU 2007 reaches a speedup of $1.41\times$ with 66.8% of all instructions being part of the FP extensions. Regardless of their FP instruction share, all executed FP benchmarks benefited from the use of FloppyFloat.

VI. Conclusion & Outlook

In this work, we presented *FloppyFloat* - the first open-source Floating Point (FP) library specifically designed for Instruction Set Simulators (ISSs). Opposed to frequently used soft float libraries, FloppyFloat uses the Floating Point Unit (FPU) of the host system, which is present on virtually all modern systems. To model the FP characteristics of a specific target Instruction Set Architecture (ISA), FloppyFloat first calculates an initial result on the host FPU and then conducts rectifications in software. Thus, FloppyFloat can model many FP features, from exceptions flags to rounding modes, without relying on the host FPU to provide these features. Currently, FloppyFloat provides predefined and tested setups for FP characteristics of ARM64, RISC-V, and x86 SSE targets. Compared to the popular Berkeley SoftFloat, individual instructions see a performance increase of up to $5.5\times$. Replacing said soft float library in the RISC-V simulator Spike [11] attains speedups of up to $1.41\times$ when executing FP benchmarks.

References

[1] "aobench," https://github.com/syoyo/aobench, accessed: 2024-08-14.
[2] "SPEC CPU 2017," www.spec.org/cpu2017, accessed: 2024-08-14.
[3] "XuanTie Missing Underflow Issue," https://github.com/revyos/revyos/issues/17, accessed: 2024-08-14.
[4] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.
[5] ——, "SoftFP," https://bellard.org/softfp/, 2018, accessed: 2023-09-15.
[6] C. Bertin, N. Brisebarre, B. Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. Raina, and A. Tisserand, "A floating-point library for integer processors," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 5559, 10 2004.
[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, aug 2011.
[8] E. G. Cota and L. P. Carloni, "Cross-ISA Machine Instrumentation Using Fast and Scalable Dynamic Binary Translation," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019.
[9] H. Curnow and B. Wichmann, "A Synthetic Benchmark." *The Computer Journal*, vol. 19, pp. 43–49, 02 1976.
[10] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.
[11] R.-V. Foundation, "Spike RISC-V ISA Simulator," https://github.com/riscv-software-src/riscv-isa-sim, accessed: 2024-08-14.
[12] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–136. [Online]. Available: https://doi.org/10.1145/2458523.2458536
[13] Y.-C. Guo, W. Yang, J.-Y. Chen, and J.-K. Lee, "Translating the ARM Neon and VFP Instructions in a Binary Translator," *Softw. Pract. Exper.*, vol. 46, no. 12, dec 2016.
[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization.*, 2001.
[15] J. R. Hauser, "Berkley SoftFloat," https://github.com/ucb-bar/berkeley-softfloat-3, 1996, accessed: 2024-08-14.
[16] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120300503
[17] R. Himeno, "Himeno Benchmark," https://github.com/kowsalyaChidambaram/Himeno-Benchmark, accessed: 2024-08-14.
[18] D. G. Hough, "The IEEE Standard 754: One for the History Books," *Computer*, vol. 52, no. 12, pp. 109–112, 2019.
[19] "IEEE Standard for Binary Floating-Point Arithmetic," IEEE, 1985.
[20] "IEEE Standard for Floating-Point Arithmetic," IEEE, 2008.
[21] "IEEE Standard for Floating-Point Arithmetic," IEEE, 2019.
[22] K. Lawton, "Bochs," https://bochs.sourceforge.io/, accessed: 2024-08-14.
[23] J. D. McCalpin, "STREAM benchmark," https://www.cs.virginia.edu/stream/, accessed: 2024-08-14.
[24] D. Mihocka and S. Shwartsman, "Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure," in *Proceedings of the Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2008, pp. 55–70.
[25] O. Møller, "Quasi Double-Precision in Floating Point Addition," *BIT*, vol. 5, no. 1, pp. 37–50, mar 1965.
[26] M. Moulin, "smallpt," https://github.com/matt77hias/smallpt, accessed: 2024-08-14.
[27] P. Sterbenz, "Floating Point Computation," 1974.
[28] J. Walker, "fbench," https://www.fourmilab.ch/fbench/fbench.html, accessed: 2024-08-14.
[29] Y.-P. You, T.-C. Lin, and W. Yang, "Translating AArch64 Floating-Point Instruction Set to the X86-64 Platform," in *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, 2019.
[30] N. Zurstrassen, L. M. Reimann, N. Bosbach, L. Juenger, and R. Leupers, "Evaluation of the RISC-V Floating Point Extensions," in *DVCon Europe 2023; Design and Verification Conference and Exhibition Europe*. VDE, 2023, pp. 57–64.
[31] N. Zurstraßen, N. Bosbach, J. M. Joseph, L. Jünger, J. H. Weinstock, and R. Leupers, "Efficient RISC-V-on-x64 Floating Point Simulation," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, 2023, pp. 1–6.