PREPRINT - ©2023 IEEE - accepted at 2023 IEEE 41st International Conference on Computer Design (ICCD)
DOI: 10.1109/ICCD58817.2023.00090

Efficient RISC-V-on-x64 Floating Point Simulation

*Niko Zurstraßen, *Nils Bosbach, *Jan Moritz Joseph,
†Lukas Jünger, †Jan Henrik Weinstock, *Rainer Leupers

**RWTH Aachen University, Institute for Communication Technologies and Embedded Systems*
†*MachineWare GmbH*

***Abstract*—Fast simulation of Virtual Platforms (VPs) is a cornerstone of modern hardware/software co-development. A particular challenge, especially if target and host Instruction Set Architecture (ISA) are different, is the simulation of Floating Point (FP) instructions. Although FP arithmetic was standardized in 1985 by IEEE 754, extensive revisions and lax definitions have led to a variety of different implementations. Thus, the question we seek to answer in this work is: How can FP instructions be efficiently simulated, if the FP arithmetic provided by the host ISA is semantically different?**

In this paper, we first provide a comprehensive overview of methods used in academia and open-source projects. Subsequently, we propose our own strategy for emulating RISC-V FP instructions on an x64 host. Our idea is to leverage the host's FPU and handle corner cases in software. In contrast to other works, we cover the full spectrum of arithmetic FP instructions and present innovative approaches, especially for the computation of division and square root. Moreover, we show how exception flags and a non-default rounding mode can be handled efficiently. Our approach achieves a $3\times$ speedup in common FP benchmarks compared to purely software-based solutions. When comparing our method against more sophisticated methods, as for example used in QEMU, we achieve a 50% performance gain for non-default rounding modes.

***Index Terms*—RISC-V, x64, Virtual Platforms, Floating Point**

I. Introduction

For several decades now, instructions for Floating Point (FP) arithmetic can be found in most general purpose CPUs. To counteract a jungle of different implementations, a first standardization was formulated in 1985 with IEEE 754 [18]. This standard is followed to a large extent by most popular ISAs like x64, ARM, or RISC-V. Surprisingly, FP instructions on these ISAs yet show different semantics. Two revisions of the IEEE 754 standard in 2008 and 2019, and a number of other factors have led to significant disparities, especially between older and newer ISAs, such as x64 and RISC-V.

For example, the maximum of an sNaN (signaling Not a Number) and an arbitrary FP number (e.g. $max(\text{sNaN}, 5.0)$) yields $sNaN$ on x64, $qNaN$ on RISC-V F 2.0, and 5.0 on RISC-V F 2.2. Despite different results, all implementation adhere to IEEE 754.

The different implementation of seemingly identical instructions poses special challenges for simulation techniques such as Dynamic Binary Translation (DBT). It is one of the fastest simulation techniques for target systems executed on a host with a different ISA. With DBT, instructions of the executed code are translated from the target to the host ISA during runtime. In many cases, instructions can be translated 1-to-1, leading to a near-native execution speed. If this 1-to-1 translation is possible for FP instructions, the approach is also referred to as *hard float*. As shown before, a hard float implementation for RISC-V-on-x64 simulation is unfeasible due to different instruction semantics. Additionally, a possible hard float approach is complicated by further reasons, like a lack of specific rounding modes or NaN boxing.

A solution used in popular open-source projects, such as *gem5* [7], or *Spike* [13], is so-called *soft float*. With soft float, FP instructions are emulated using integer arithmetic, usually as part of a library in a high-level programming language. This provides full control over the implementation, but negatively impacts the simulator's performance.

In this paper, we propose a solution to the crippling performance of soft float and the inflexibilities imposed by hard float. The general idea of our method is to leverage the host Floating Point Unit (FPU) whenever possible, and handle corner cases in software. Compared to other works that take a similar approach, our work covers the full spectrum of arithmetic FP instructions and presents new methods for the calculation of square root and division. In addition, we show how our method can be used to simulate different rounding modes. While the focus of our work is on RISC-V-on-x64 simulation, the presented methods can be applied to other cases as well. To demonstrate the practical benefits of our method, we integrated it into the RISC-V-on-x64 simulator *SIM-V* [20]. Using standard floating point benchmarks, our method achieves performance benefits of $3\times$ compared to soft float approaches. Compared against methods as used in state-of-the-art emulators, like QEMU, our approach executes FP benchmarks up to 44% faster for non-default rounding modes. To summarize, our work contributes:

- New RISC-V-on-x64 FP simulation approaches for quick calculation of rounding and exception flags.
- An extensive overview of methods used in open-source projects and academia.
- A case-study and performance analysis of our approach.

II. Background

In this section, we first introduce mathematical definitions of FP arithmetic. This is followed by the details of FP in the x64 and RISC-V ISA. The emphasis is on the differences between the two architectures, since these represent the main challenge in FP simulation. Finally, the section is concluded with an explanation of methods for overcoming this challenge.

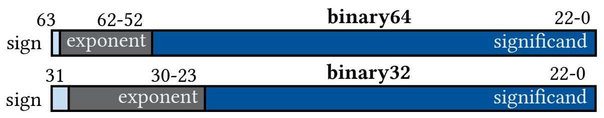

Fig. 1. Bit representation of IEEE 754 binary32 and binary64.

A. Mathematical Background

When speaking of FP arithmetic, we implicitly assume the most common format according to IEEE 754. In particular, the base-2 formats *binary32* and *binary64* with 32-bit and 64-bit precision, respectively. These are also known as *single* and *double* precision. As shown in Fig. 1, the binary representation of a FP number comprises a sign, a significand, and an exponent. Mathematically, the value of an FP number is calculated as:

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

Since $exponent$ is an unsigned value, an implicit bias is subtracted to be able to represent negative exponents. Exponents with all bits set to 1 or set to 0 are reserved for special values. In case of all bits set to 1, the significand determines if the number is either interpreted as NaN or infinity. If all bits are 0 ($exponent = 0$), the implicit leading 1 is replaced by a 0, and the numbers are referred to as subnormals. While Equation 1 is often used to introduce the concept of IEEE FP numbers, the $p-1$ significand bits with an implicit leading 1 complicate mathematical handling. A representation more suitable for mathematical proofs is:

$$f = M \cdot 2^{e-p+1}, \quad e = exponent - bias \quad (2)$$

Depending on the data type, the precision p, significand M, and the exponent e are constrained by the values given in Table I. Note that the p precision bits include the implicit leading 1. For example, a binary32 value has a precision of 24 bits of which 23 bits are explicitly stored.

Whenever mathematical operations are performed on FP numbers, rounding errors may occur. In literature and this work, rounding is symbolized by the $\circ$ operator. E.g., $\circ_{RNE,32}(a+b)$ corresponds to a 32-bit addition under Round Nearest, Ties to Even (RNE) rounding mode. In the following, if no rounding mode is given, RNE shall be assumed. The errors caused by rounding can be described by the standard error model of FP arithmetic [17]. According to the model, the error of many arithmetic operations $(+, -, /, \cdot, \sqrt{})$, including underflows, can be represented as:

$$\begin{gathered} z = (a \operatorname{op} b) \cdot (1+\epsilon) + \eta = \circ(a \operatorname{op} b) \\ \epsilon\eta = 0, \quad |\epsilon| \le \mathbf{u}, \quad \eta \le |2^{e_{min}}| \cdot \mathbf{u}, \quad \mathbf{u} = 2^{-p} \end{gathered} \quad (3)$$

If no underflow occurs, $\eta = 0$ holds, otherwise $\epsilon = 0$ holds, meaning z is a subnormal number. The relative error ϵ is bounded by the so-called *unit roundoff* error $\mathbf{u}$.

TABLE I
CONSTRAINTS FOR BINARY32 AND BINARY64. WITH $p, M, e \in \mathbb{Z}$.

data type	exponent range	precision bits	significand range
binary32	$-126 \le e_f \le 127$	$p_f = 24$	$\lvert M_f \rvert \le 2^{24} - 1$
binary64	$-1022 \le e_d \le 1023$	$p_d = 53$	$\lvert M_d \rvert \le 2^{53} - 1$

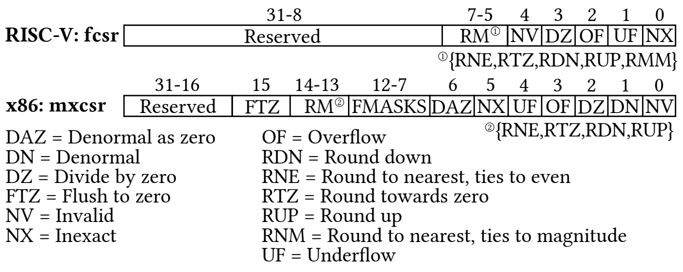

Fig. 2. FP control/status registers of RISC-V (fcsr) and x64 SSE (mxcsr).

B. RISC-V Floating Point Arithmetic

In RISC-V, all functionalities beyond fundamental operations are defined by extensions. The extensions *F* and *D* are considered the standard FP extensions, providing 32-bit and 64-bit FP arithmetic respectively. Both FP extensions adhere to the IEEE 754 2019 standard [19]. Accordingly, there are 5 rounding modes and 5 FP exceptions, which are represented in the `fcsr` register (see Fig. 2). Opposed to other architectures, RISC-V does not trigger traps when encountering FP exceptions. A special peculiarity, not mentioned in the IEEE standard, is RISC-V's hardware-assisted NaN boxing, which saturates the upper bits if an N-bit FP value is written to an M-bit FP register with $M > N$. For example, if a 32-bit FP value resides in a 64-bit register, it is only considered valid if the upper 32 bits are set to 1. This means, instructions working solely on 32-bit FP values must check the upper bits when reading the operands and set them when writing back the result. Since the whole 64-bit value encodes to a negative qNaN, there is no risk of creating valid values by accident. The canonical qNaN created by instructions with invalid operands adheres to the IEEE 754 standard and uses a positive sign. For example, a 32-bit division by zero will result in 0x7fc00000 for 32-bit FP registers. The same 32-bit division for 64-bit FP registers results in a NaN-boxed value of 0xffffffff7fc00000.

C. x64 Floating Point Arithmetic

The following description of the x64 FP arithmetic refers to the SSE and AVX extensions. These were preceded by the x87 extension from 1980, which is still supported by modern x64 CPUs, but widely considered obsolete.

x64 FP instructions follow the IEEE 754 standard [18] from 1985. In addition to the aforementioned functional differences of certain instructions, there are other important subtleties that need to be considered. For example, x64 misses the RNM rounding mode, which was introduced in later standards (see Fig. 2). The five FP exceptions (invalid, underflow, overflow, inexact, divide-by-zero) were already defined in the first standard matching RISC-V in this regard. Yet mapping the FP exceptions from host to target turned out to be one of the most difficult challenges, as shown in the next section. In addition to the five standard FP exceptions, x64 also defines a denormal flag for the detection of subnormal results. One further peculiarity is x64's support for treating subnormal numbers as 0 using the FTZ and DAZ flags. This allows to treat accuracy for performance in underflow-prone applications [12]. In contrast to RISC-V, the x64 ISA also allows to specify which FP exceptions cause a trap. The corresponding masking bits are selected in the FMASK field, as depicted in Fig. 2. Another difference between RISC-V and x64 is the canonical qNaN encoding. On x64 systems, the canonical qNaN uses a negative sign, as opposed to the RISC-V positive sign. That means, a 32-bit qNaN as a result of an invalid operation would be encoded as 0xffc00000.

D. Simulation Methods

As already shown, the FP arithmetic of RISC-V and x64 differ in several aspects. This impedes the simulation performance of RISC-V targets on x64 hosts, since instructions cannot be translated in a 1-to-1 fashion. In particular, the following challenges arise:

- Different instruction semantics.
- Different signs for canonical qNaN values.
- x64 missing RNM rounding mode.
- RISC-V NaN boxing.
- Mapping of FP exceptions.

To overcome these challenges, various methods have been developed. In general, they can be categorized as *hard float*, *floppy float*, or *soft float*.

Hard float refers to a simple 1-to-1 target-to-host instruction mapping. This allows for near-native execution speeds, but differences between host and target ISA might impose insurmountable hurdles. However, if bit-level equality of results is not a concern, this approach can be used to trade accuracy for performance. For example, instead of mapping a x64 qNaN to a RISC-V qNaN, the value could be just taken as is. Since both qNaNs are semantically the same, most applications are not affected by different bit-level representations of NaN values. Yet there are applications that expect certain NaN values in order to function properly.

Soft float is a commonly used method to deal with the accuracy flaws of hard float. Instead of using the host's FPU, FP instructions are simulated using integer arithmetic. Usually, each instruction has a corresponding function call, which is part of a software library written in a high-level programming language, such as C. This provides full control of rounding modes, exceptions, and instruction semantics. However, simulating FP instructions using integer arithmetic is associated with low performance. In our experiments, soft float was up to 80× slower than hard float.

The *floppy float* approach combines the flexibility of soft float with the performance of hard float. This is achieved by executing the computational kernel of an instruction on the host FPU and catch corner cases in software. A detailed analysis in the context of other works is given in the next section.

III. Related Work

In the following, an extensive analysis of the simulators *rv8* [8], *QEMU* [4], gem5 [7], and *Spike* [13], is provided. This is complemented by an analysis of academic works [10], [15], [23], [25].

A. rv8

rv8 [8], [9] is an open-source, DBT-based, RISC-V emulator for x64 hosts. Its approach for FP simulation can be characterized as hard float with minor software-based checks. With rv8, the RISC-V target exception flags and rounding mode are mapped 1-to-1 to the x64 host. As explained in Section II, RISC-V and x64 are equivalent with regards to their exception flags. Thus, checking and setting the exceptions flags is simply achieved by accessing the host's `mxcsr` register. Yet mapping the rounding modes poses a problem, as x64 is not endowed with a mode for RNM. rv8 solves this problem by incorrectly mapping RNM to RNE. This trades accuracy for simplicity/performance and leads to rv8 not being conform with the RISC-V standard. Other problems, such as NaN boxing or semantically different instructions, are solved by rectifications in software.

B. gem5, QEMU pre 4.0.0, Spike

The open-source projects gem5 [7], Spike [13], and QEMU [4] pre-v4.0.0 use soft float libraries, with the latter transitioning to floppy float since version 4.0.0. All simulators use(d) the open-source library *Berkley Softfloat* by J. Hauser [16] which is based on the IEEE 754 1985 standard [18]. Soft float libraries that implement the more recent IEEE 754 2008 standard include *SoftFP* by F. Bellard [5], and *FLIP* by C.-P. Jeannerod et al. [6]. Besides generic solutions in programming languages like C, there are also architecture-optimized soft float libraries. For example, *RVfplib* [22] is an optimized library for RISC-V systems that do not include the *F* or *D* extension.

C. QEMU post-v4.0.0

As of version 4.0.0, there is an ongoing effort to speed up the FP performance of QEMU by using the floppy float method from Guo et al. [15]. The initial idea of Guo et al. was to calculate the result of a FP instruction on the host FPU, while the exception flags are emulated in software. This approach showed a significant overhead of the calculation of the inexact exception, leading to no speedup compared to soft float. To avoid the high costs for the inexact calculation, a FP operation is preceded by a quick check, whether the exception must be calculated at all. Because with most ISAs, such as RISC-V or x64, the inexact exception is sticky. I.e., if an instruction has generated an inexact result, the inexact exception remains set, even if subsequent instructions produce exact results. As most applications do not clear the exception and tend to generate an inexact result at some point, it can be assumed, that in most cases the inexact exception is already set.

An example for the square root instruction in QEMU using Guo's method is shown in Code 1. The function starts with a call to `can_use_fpu`. This corresponds to the aforementioned check of the inexact flag. Furthermore, the host FPU can only be used if the host rounding mode corresponds to the target's rounding mode. It is assumed that the default C rounding mode of RNE is used. Hence, a check of the target's rounding mode is sufficient. Since some target architectures

PREPRINT - ©2023 IEEE - accepted at 2023 IEEE 41st International Conference on Computer Design (ICCD)
DOI: 10.1109/ICCD58817.2023.00090

```
 1 static inline bool can_use_fpu(const float_status *s) {
 2   if (QEMU_NO_HARDFLOAT)
 3     return false;
 4
 5   return likely(s->f_excep_flags & f_flag_inexact
 6          && s->f_round_mode == f_round_near_even);
 7 }
 8
 9 float32 float32_sqrt(float32 xa, float_status *s) {
10   if (unlikely(!can_use_fpu(s)))
11     goto soft;
12
13   if (unlikely(!is_zero_or_norm(xa) || is_neg(xa)))
14     goto soft;
15
16   return sqrtf(ua.h);
17   soft: return soft_f32_sqrt(ua.s, s);
18 }
```

Code 1. Code of QEMU's square root instruction (`fpu/softfloat.h`).

like PowerPC do not have a sticky inexact exception, the check can be skipped by defining `QEMU_NO_HARDFLOAT`. To also avoid setting the underflow and invalid exception, a second check is performed in Line 13. The idea of extending Guo's method by checking both underflow and invalid flags was proposed by Cota et al. [10]. If all checks passed, the hard float function `sqrtf` is called, resulting in a `sqrtss` instruction for x64 hosts. In case the checks do not pass, the corresponding soft float function is called.

Using the method of Guo et al., the performance of FP instructions can be increased by a factor of more than $2\times$. However, this speedup can only be achieved if the RNE mode is used and if an inexact exception occurs at some point. To tackle the latter issue for additions, Guo et al. developed a quick inexact check which resembles the Fast2Sum algorithm by T.J. Dekker [11].

D. *You et al.*

Picking up the idea of fast inexact checks from Guo et al., You et al. [25] extended it by accounting for rounding modes as well. This is achieved by utilizing the residual r of the Fast2Sum algorithm by T.J. Dekker [11] (see Code 2). According to Dekker, the result of a rounded addition can be described by the sum of its exact value and a residual:

$$\begin{aligned} a + b - r = s = \circ(a+b) \\ r = \circ(b - \circ(s-a)) \quad with\ |a| > |b| \end{aligned} \tag{4}$$

As mathematically proven by Dekker, the residual r holds the exact rounding error of the addition of the variables a and b. Hence, if the residual r is not 0, the FP addition was inexact. Additionally, the value of the residual also determines the rounding direction of the preceding addition $\circ(a+b)$. For $r > 0$, the result of the addition was rounded down; for $r < 0$, the result was rounded up. This can be used, to adjust for any given target rounding mode, as shown in Code 2 Line 7-9. While You et al. managed to develop fast inexact checks and rounding adjustment for additions and conversions, other instructions remain untouched.

E. *Sarrazin et al.*

During the implementation of an Kalray-1-on-x64 simulator, the group of Sarrazin et al. [23] faced the problem

```
 1 float c = a + b; // Result.
 2 float x = fabs(a) > fabs(b) ? a : b;
 3 float y = fabs(a) > fabs(b) ? b : a;
 4 float r = y - (c - x); // Rounding error.
 5 if (r != 0.f) {
 6   inexact = true;
 7   if (r > 0.f) {
 8     c = nextup(c); // Next greater FP value.
 9     overflow = is_inf(c) ? true : overflow;
10   }
11 }
```

Code 2. C++ code of the Fast2Sum algorithm using RUP rounding.

calculating Fused Multiply-Add (FMA) instructions without hardware support. As a solution they combined exact 32-to-64-bit multiplications with the 2Sum algorithm:

$$\begin{aligned} M = \circ_{64}(a \cdot b) \quad S, T = 2Sum(M, \circ_{64}(c)) \\ r = \circ_{32}(S) \quad E = ||S - r|| \\ with\ \circ_{32}(a) = a, \quad \circ_{32}(b) = b, \quad \circ_{32}(c) = c \end{aligned} \tag{5}$$

Note that the result of the 2Sum algorithm is identical to the Fast2Sum algorithm, which was presented in the previous subsection. A detailed discussion about the differences and performance implications is provided in Subsection IV-A. The residual T determines if the addition of c and $a \cdot b$ was inexact. This can have an impact on the rounding if E is in the middle of two 32-bit FP numbers ($E = 2^{e_r - p}$). Hence, if E is equal to $2^{e_r - p}$, S and T must be used to adapt r accordingly.

While the approach of Sarrazin et al. is about calculating FMA by means of multiplcation and addition, the general idea can be used to determine the inexactness and perform directed rounding. This is shown in Subsection IV-A5.

One major disadvantage of the method by Sarrazin et al. is the dependence on larger data types. If the residual of a 32-bit FMA instruction is computed, at least 64-bit FP precision is required. Or more precisely, the larger data type needs at least $2p$ significand bits. Consequently, this algorithm does not work for double precision values on x64 systems.

F. *FPU Guards*

In the previous subsections, we showed several approaches of emulating a specific FP arithmetic. The focus of the floppy float approaches [10], [15], [23], [25] was on a fast calculation of the exception flags and the simulation of rounding modes. At first glance, this may seem superfluous and too complicated for RISC-V-x64 simulation. As already shown for *rv8*, at least for the exception flags, the `mxcsr` registers of the host can be used. Also for the rounding modes, 4 out of 5 modes can be set using the `mxcsr`. Hence, a possible method for fast simulation, which we call FPU guards, could look as follows:

1) Save host FPU state
2) Load target FPU state
3) Execute target FP instruction
4) Save target FPU state
5) Load host FPU state

In fact, this simple method was also considered in the works of Guo et al. [15], You et al. [25], Sarrazin et al. [23], and Cota et al. [10]. However, in three of the works, it was ultimately discarded due to poor performance. According to Guo et al. [15], the FPU guards are even outperformed by soft float implementations. Also we could we could confirm the results with our experiments. An investigation revealed that the poor performance can be attributed to the extremely high-latency x64 instructions to access the FP status register.

IV. Methods

A. Quick Inexact/Rounding Calculations

The main contribution of our work is presented in this section. Thereby, the focus is especially on fast calculation of exception flags and simulation of rounding modes. By introducing methods for addition/subtraction, multiplication, division, square root, and FMA, we cover all relevant FP instructions. Besides using mathematical means to check the validity of our approach, all instructions were verified using the *RISC-V Architecture Test* [14].

1) Fast Addition: As explained in Section III-D, the work of You et. al [25] uses the Fast2Sum algorithm for the calculation of the residual r. This requires two arithmetic operations, but the operands must be sorted by absolute value. Consequently, branching instructions might be needed, which can lead to performance penalties. As an alternative without sorted operands, O. Møller [21] proposed the *2Sum* algorithm. It does not require branching instructions, but involves more arithmetic instructions (see Code 3). In our experiments, the 2Sum algorithm was about 10% faster than the Fast2Sum algorithm when working on randomized data. If the input data is predictable, thus favorable to the branch predictor, both algorithms achieve the same performance.

```
float c = a + b; // Result.
float da = (c - b) - a;
float db = (c - a) - b;
float r = da + db; // Residual.
```

Code 3. C++ code of the 2Sum algorithm.

2) Fast 32-bit Multiplication: To enable fast calculation of multiplications, we developed the *UpMul* algorithm (see Code 4). Similar to addition, the value of the residual is calculated, values are rounded, and exceptions are set. For *UpMul* to work, the operands a and b must be available as 32-bit FP values. In a first step, these are upcasted to 64-bit values and then multiplied. Since the number of significands more than doubles from 32-bit FP to 64-bit FP, the result of the multiplication can be represented exactly. If the exact value is subtracted from the erroneous value, the residual remains (see Equation 6).

$$\begin{aligned} c_{exact} + r = c = \circ_{32}(a \cdot b) \\ r = \circ_{64}(\circ_{32}(a \cdot b) - \circ_{64}(a \cdot b)) \end{aligned} \tag{6}$$

The formula can be derived by first showing that the multiplication of the 32-bit values as 64-bit values is exact. Using Equation 2, the multiplication can be expressed as:

$$a \cdot b = M_a \cdot M_b \cdot 2^{e_a + e_b - 2p_f + 2} = c = M_c \cdot 2^{e_c - p_d + 1} \tag{7}$$

Thus, we can derive the ranges of M_c and e_c:

$$\begin{aligned} |M_c| = |M_a \cdot M_b| \leq (2^{24} - 1)^2 \leq 2^{48} - 1 \leq 2^{p_d} - 1 \\ |e_c| = |e_a + e_b - 2p_f + p_d + 1| \leq 260 \leq |e_{d,min}| \end{aligned} \tag{8}$$

Since both M_c and e_c fit into the range of a double-precision value, the result of the multiplication is exact. Furthermore, the 64-bit multiplication of two 32-bit values can never lead to an underflow, which is why we do not consider this corner case. From Equation 8 we can also see why $2p$ significand bits are required to represent a multiplication exactly. Next the exactness of the subtraction can be shown by using Sterbenz' Lemma [24]. Since the values of $\circ_{64}(a \cdot b)$ and $\circ_{32}(a \cdot b)$ differ by not more than a unit roundoff, their subtraction is exact.

A C++ implementation for the RUP rounding mode can be found in Code 4. As shown in the code, an inexact calculation has occurred if $r \neq 0$ (Line 3). Subsequently, the result is rectified in case the host hardware rounded it down (Line 5-6). This could lead to an overflow, hence the result is checked for infinity (Line 7). According to the RISC-V ISA, tininess is detected after rounding, requiring an underflow check after the rectification (Line 9).

```
1  float c = a * b;
2  double r = (double)c - (double)a * (double)b;
3  if (r != 0.) {
4    inexact = true;
5    if (r < 0.) {
6      c = nextup(c); // Next greater FP value.
7      overflow = is_inf(c) ? true : overflow;
8    }
9    underflow = (is_subnormal(c) || is_zero(c))
10                  ? true : underflow;
11 }
```

Code 4. C++ code of the UpMul algorithm for a RUP case.

3) Fast 32-bit Division: For fast division, we developed a new method called *UpDiv* (see Code 5), which was not seen in any other work before. Similar to the *UpMul* algorithm, both operands must be 32-bit FP values, and the goal is to compute the residual r. However, in this case, the exact determination of the residual of a division is overambitious, as certain rational numbers cannot be represented with a finite number of significand bits. Nevertheless, the exact value of the residual is not crucial for our endeavor. Rather, we want to know whether there was a rounding error, and if it is positive or negative. In mathematical terms, an approximation of the residual $\tilde{r}$ is sought, for which $sgn(\tilde{r}) = sgn(r)$ is satisfied. Such an approximation is obtained by:

$$\begin{aligned} c_{exact} + r = c = \circ_{32}(a/b) \\ \tilde{r} = \circ_{64}(\circ_{64}(\circ_{32}(a/b) \cdot b) - a) \cdot sgn(b) \end{aligned} \tag{9}$$

The equation can be derived by using the standard model of FP arithmetic extended for subnormals (see Equation 3). According to the model, the error of the FP division, including underflow and overflow, can be represented as follows:

$$\frac{a}{b} \cdot (1 + \epsilon_1) + \eta_1 = \circ_{32}(a/b) \tag{10}$$

If the result of the division is upcasted to 64-bit and multiplied by the value of b, which is also upcasted to 64-bit, the result must be exact (see previous section). This allows to calculate the approximation $\tilde{a}$ as follows:

$$\tilde{a} = a + a\epsilon_1 + b\eta_1 = \circ_{64}(b \cdot \circ_{32}(a/b)) \tag{11}$$

Subtracting a from $\tilde{a}$ yields Equation 12.

$$z = \circ_{64}(\circ_{64}(b \cdot \circ_{32}(a/b)) - a) = (a\epsilon_1 + b\eta_1)(1+\epsilon_2)$$
$$z = \begin{cases} b\eta_1(1+\epsilon_2) & subn. \\ a\epsilon_1(1+\epsilon_2) & else \end{cases} \tag{12}$$

Although this addition can be inexact, which is described by ϵ_2, the result 0 can only be obtained if the preceding division was exact ($\epsilon_1 = \eta_1 = 0$). Otherwise, the sign of z is directly determined by $a\epsilon_1$ or $b\eta_1$. Next, Equation 12 is rearranged to:

$$\epsilon_1 = \frac{z}{a \cdot (1+\epsilon_2)} \qquad \eta_1 = \frac{z}{b \cdot (1+\epsilon_2)} \tag{13}$$

Inserting Equation 13 into Equation 10 yields for both cases the following residual:

$$r = z/(b \cdot (1+\epsilon_2)) \tag{14}$$

Therefore, the residual can only be 0 if z is 0 as well. Likewise, the sign of r is directly determined by z and b. Consequently, we conclude $sgn(\tilde{r}) = sgn(r)$.

```cpp
float c = a / b;
double r = (double)c * (double)b - a;
r = signbit(b) ? -r : r;
if (r != 0.) {
  inexact = true;
  if (r < 0.) {
    c = nextup(c); // Next greater FP value.
    overflow = is_inf(c) ? true : overflow;
  }
  underflow = (is_subnormal(c) || is_zero(c))
                ? true : underflow;
}
```

Code 5. C++ code of the UpDiv algorithm for a RUP case.

4) Fast Square Root: The calculation of a fast square root and its residual follows the same principle as the UpMul and the UpDiv algorithm. We exploit that multiplication is the inverse operation of square root, and that multiplication with larger data types is exact. The residual results according to Equation 15.

$$b_{exact} + r = b = \circ_{32}(\sqrt{a})$$
$$\tilde{r} = \circ_{64}(\circ_{64}(\circ_{32}(\sqrt{a})^2) - a) \tag{15}$$

The proof of the *UpSqrt* algorithm is equivalent to the proof of the *UpDiv* algorithm. Again, an approximation $\tilde{r}$ for the residual r with $sgn(r) = sgn(\tilde{r})$ is sought. Similar to the UpDiv proof, we exploit exact multiplications by upcasting. Moreover, the multiplication can be used as an inverse function of the actual operation. The final result is the following expression:

$$r = \sqrt{\frac{\tilde{r}}{1+\epsilon_2} + a} - \sqrt{a} \tag{16}$$

Since the sign of r is only dependent on $\tilde{r}$, $sgn(r) = sgn(\tilde{r})$ holds.

5) Fast 32-bit Fused Multiply-Add: For fast FMA simulation we repurposed and extended the method of Sarrazin et al. [23]. The idea is to first calculate the exact multiplication of a and b using a larger data type. Subsequently, the residual of the summation of $a \cdot b$ and c is calculated using the 2Sum algorithm. But even if this summation was exact ($r_1 = 0$), the final result might not be representable as 32-bit FP value. Hence, another residual r_2 is calculated to determine the 64-bit to 32-bit rounding error. Note that r_2 is exact due to Sterbenz' Lemma [24].

$$d_{exact} + r = d = \circ_{32}(a \cdot b + c)$$
$$r_1 = 2Sum(\circ_{64}(a \cdot b), c) \tag{17}$$
$$r_2 = \circ_{64}(\circ_{64}(\circ_{64}(a \cdot b) + c) - d)$$

Finally, an approximation of the rounding error $\tilde{r}$ can be calculated, as shown in Equation 18.

$$\tilde{r} = r_1 + r_2 \tag{18}$$

Although the addition of r_1 and r_2 is not exact per se, it satisfies $sgn(\tilde{r}) = sgn(r)$.

6) Fast 64-bit Operations: Our previous upcast algorithms *UpMul*, *UpDiv*, *UpSqrt*, and also the FMA algorithm according to Sarrazin et al. [23], are all based on the availability of a larger data type that can perform multiplications exactly. As mentioned earlier, these algorithms reach their limitations for 64-bit values on x64 systems. To circumvent these limitations, the fused-multiply-add (FMA) of the x64 ISA can be used. This instruction is defined in the FMA3/FMA4 instruction set extensions and is part of all modern x64 processors. For example, using FMA, the residual of the *UpMul* algorithm can be calculated as follows:

$$r' = \circ_{64}(\circ_{64}(a \cdot b) - a \cdot b) = \circ_{64}(c - c_{exact}) \tag{19}$$

However, the rounding step at the end of each FMA instruction poses a problem. Although an FMA instruction calculates all intermediate results with infinite precision, the result is eventually rounded. In the example shown, it is possible that r' is not representable with a 64-bit precision. One could therefore wrongly assume a value of 0, although the value is actually different from 0. Hence, $r' = r$ does not hold in all cases.

Consequently, bounds must be determined for which r' is no longer representable. Since r' is the direct result of the subtraction of c and c', we have to determine the smallest distance between these numbers, excluding 0. This distance is $|d| \geq 2^{e_c - 2p_d}$. The number of double significand bits $2p_d$ follows from the exact intermediate results of the FMA instruction. As explained previously, $2p$ significand bits are needed for the exact representation of a p-bit multiplication. In order to represent r' as a 64-bit FP value, $e_c - 2p_d \geq e_{d,min} - p_d + 1$ must hold. A simple rearrangement leads to the following inequality:

$$e_c \geq e_{d,min} + p_d + 1 = -1022 + 53 + 1 = -968 \tag{20}$$

If $|c|$ is less than 2^{-968}, our method cannot be used, and the instruction has to be calculated using soft float. However, the range below 2^{-968} represents less than 3% of all 64-bit FP values. An example for the division is given Code 6.

```
if (abs(a) < 4.008336720017946e-292)
    return soft_fp::div(a, b);

double r = std::fma(c, b, -a);
if (r != 0.) {
    inexact = true;
    underflow = (is_subnormal(c) || is_zero(c))
                  ? true : underflow;
}
```

Code 6. C++ code for a 64-bit UpDiv algorithm using a FMA instruction.

V. Results and Discussion

In the following subsections, the performance results of our floppy float approach are presented. All experiments were conducted on a *AMD Ryzen Threadripper 3990x* running an Ubuntu 21.10 operating system. The machine can achieve a boost clock of 4.3GHz.

A. Instruction Benchmark Results

In the previous sections, we made qualitative statements about the performance of FP simulation approaches. We claimed that hard float is faster than floppy float, and that floppy float is faster than soft float. To support these statements with evidence, we developed a benchmark to assess the performance of the respective approaches. The benchmark is designed to determine the maximum performance of each individual instruction. That means, there is no DBT overhead, inputs and outputs are never subnormal, and there are no data dependencies between the instructions. Also, the input data was designed to work optimally with the soft float method. It should be noted that soft float is sensitive to input data due to its control-flow-heavy calculations. The results of our benchmark are shown in Fig. 3. Hard float achieves up to 8500 MIPS for instructions that can be executed in one cycle (max, min, add, sub, etc.). This is explained by the FP pipeline of the host processor. Most of the instructions can use 2 of 4 FP pipes provided by the Zen 2 microarchitecture, leading to $8500 MIPS \approx 2 \cdot 4.3GHz$. Some instructions, such as division, square root, or 64-bit multiplication, require multiple cycles, which results in lower performance. Nevertheless, hard float is faster than soft and floppy float in all cases. The performance of the floppy float approach is in the range of 300-600 MIPS, and is faster than soft float by up to $5\times$ in some operations, such as square root. For lightweight operations, such as min or max, there is no significant difference between soft- and floppy float.

B. System Benchmark Results

Since our approach is intended to accelerate FP performance in DBT simulators, a practical performance assessment is indispensable. For this purpose, we integrated our approach, the method by Cota et al. [10] (QEMU's method), and Bellard's *SoftFP*, into the DBT-based RISC-V simulator SIM-V [20]. This simulator was then used to conduct a performance analysis using well-known FP benchmarks, such as linpack [1], NPB [2], SPEC CPU 2017 [3], and other representative workloads. The results can be found in Fig. 4. In the graph, the speedups of the individual benchmarks are shown, whereby the soft float method was used as a reference baseline. All benchmarks in Subplot a) were executed with the default RNE rounding, while Subplot b) represents the same benchmarks under RUP rounding.

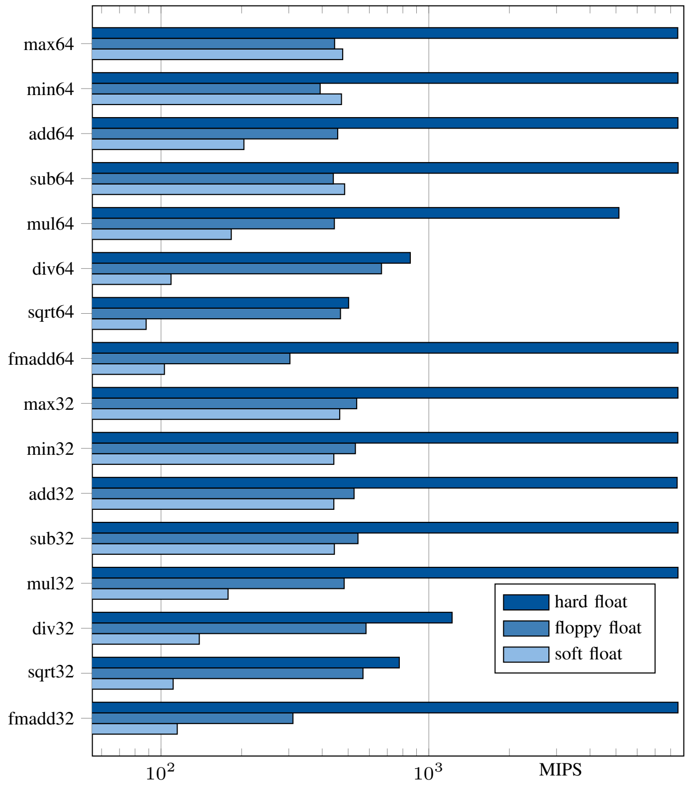

Fig. 3. Measuring the performance of soft, floppy, and hard float. Floppy float refers to the approach presented in this paper, while soft float is based on SoftFP [5].

As can be seen in the graph, QEMU's method and our approach achieve a speedup of $3\times$ in a best case scenario (see Subplot a), NPB/ft.A and 508.namd). Also, in most cases, the performance of our approach is equal to the performance of QEMU's approach when RNE rounding is used. As explained in Section IV, our approach is only faster when underflows occur and no inexact flags are set, or when a non-default rounding mode is not used. Since most applications already set an inexact flag after a few executed instructions, the speedup gained from an accelerated inexact calculation is marginal. Also, underflows are seldom, as we could confirm with an instruction and data study. For example, in the case of the NPB/ft.A benchmark, not a single underflow occurred in a total of 3,875,127,289 executed fmadd instructions.

To demonstrate the advantages of our methods, we ran all benchmarks again under RUP rounding, which is depicted in Fig. 4 b). Here we can see that QEMU is slower than soft float in all cases. This can be attributed to the fact that QEMU first checks the rounding mode before resorting to soft float (see Code 1). Our method, however, can rectify the result for

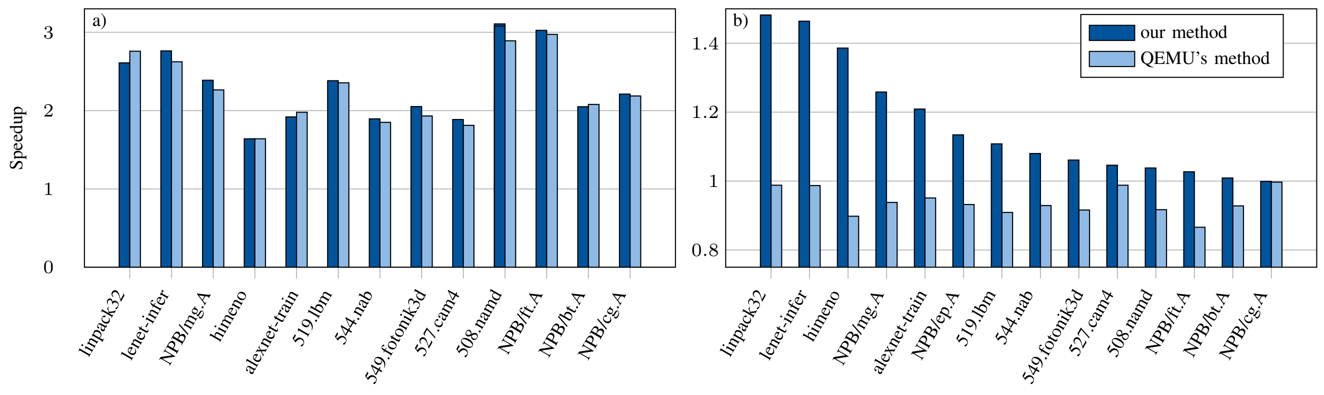

Fig. 4. System benchmark performance of our and QEMU's method. The performance of soft float is used as the speedup's baseline. Subplot a) represents benchmarks executed with the default RNE rounding mode while benchmarks in b) were executed with RUP.

most instructions and set the exception flags without using soft float. Thus, we achieve speedups of 50% over QEMU for benchmarks like linpack32. Since the speedup of our method depends on the executed instructions, we observe a heterogeneous picture of results. Moreover, the speedups under RNE cannot be used to infer the speedups under RUP. As described in Section IV, we do not have a method for 64-bit FMA instructions, and all presented approaches require less checks when working on 32-bit data. Hence, single precision benchmarks, such as linpack32 or machine learning applications (lenet, alexnet), achieve higher speedups in non-default rounding modes. Applications that comprise many 64-bit FMA instructions achieve low to no speedup (see NPB/bt.A and NPB/cg.A).

VI. Conclusion & Outlook

In this work, we have shown how FP instructions can be efficiently simulated if target and host ISA are different. As a first step, we conducted a comprehensive analysis of existing work to create the most complete picture of this topic to date. We then summarized our findings and complemented them with our own approaches. The focus of our work was on the fast computation of inexact/underflow exceptions and different rounding modes. To emphasize the practical relevance of our approach, we integrated our method into the RISC-V simulator SIM-V and compared it with other approaches. This was done by comparing the achieved performance of well-known FP benchmarks, such as SPEC 2007, NPB, and linpack. As shown in the results, the advantage of our approach becomes evident when a non-default rounding mode is used. In these cases, our approach is 50% faster than the method used by QEMU. For default rounding modes, we are on par with QEMU's method achieving speedups of up to 3x compared to soft float. Since the speedup is based on a fast calculation of exception flags and rounding errors, architectures with non-sticky bits, such as PowerPC, also benefit for default rounding modes. As shown in our work, efficient algorithms for 64-bit FMA instructions were still untouched and remain subject to future investigations. Also, the efficient simulation of narrower data types, such as float16, or vector instructions, might be relevant in future due to the growing interest in machine learning.

References

[1] "LINPACK C benchmark," https://netlib.org/benchmark/linpackc.new, accessed: 2023-09-15.
[2] "NAS Parallel Benchmarks," https://www.nas.nasa.gov/software/npb.html, accessed: 2023-09-15.
[3] "SPEC CPU 2017," www.spec.org/cpu2017, accessed: 2023-09-15.
[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 *et al.*, "A floating-point library for integer processors," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 5559, 10 2004.
[7] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, aug 2011.
[8] M. Clark *et al.*, "rv8 - RISC-V simulator for x86-64," https://github.com/michaeljclark/rv8, accessed: 2023-09-15.
[9] ——, "rv8: a high performance RISC-V to x86 binary translator," *CARRV*, 10 2017.
[10] E. G. Cota *et al.*, "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.
[11] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.
[12] I. Dooley *et al.*, "Quantifying the interference caused by subnormal floating-point values," 01 2006.
[13] R.-V. Foundation, "Spike RISC-V ISA Simulator," https://github.com/riscv-software-src/riscv-isa-sim, accessed: 2023-09-15.
[14] Gala, N. and Karasek, M., "RISC-V Architecture Test," https://github.com/riscv-non-isa/riscv-arch-test, accessed: 2023-09-15.
[15] Y.-C. Guo *et al.*, "Translating the ARM Neon and VFP Instructions in a Binary Translator," *Softw. Pract. Exper.*, vol. 46, no. 12, dec 2016.
[16] J. R. Hauser, "Berkley SoftFloat," https://github.com/ucb-bar/berkeley-softfloat-3, 1996, accessed: 2023-09-15.
[17] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2002.
[18] "IEEE Standard for Binary Floating-Point Arithmetic," IEEE, 1985.
[19] "IEEE Standard for Floating-Point Arithmetic," IEEE, 2019.
[20] L. Jünger *et al.*, "SIM-V: Fast, Parallel RISC-V Simulation for Rapid Software Verification," *DVCON Europe 2022*, 2022.
[21] O. Møller, "Quasi Double-Precision in Floating Point Addition," *BIT*, vol. 5, no. 1, pp. 37–50, mar 1965.
[22] M. Perotti *et al.*, "RVfplib: A Fast and Compact Open-Source Floating-Point Emulation Library for Tiny RISC-V Processors," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu *et al.*, Eds., 2022.
[23] G. Sarrazin *et al.*, "Virtual Prototyping of Floating Point Units," ser. RAPIDO '16, 2016.
[24] P. Sterbenz, "Floating Point Computation," 1974.
[25] Y.-P. You *et al.*, "Translating AArch64 Floating-Point Instruction Set to the X86-64 Platform," in *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, 2019.