Evaluation of the RISC-V Floating Point Extensions

*Niko Zurstraßen, *Lennart M. Reimann,
*Nils Bosbach, †Lukas Jünger, *Rainer Leupers
*RWTH Aachen University, Institute for Communication Technologies and Embedded Systems
†MachineWare GmbH, Aachen, Germany

***Abstract*—Designing an Instruction Set Architecture (ISA) is a challenging task that plays a crucial role in shaping the characteristics of compute systems. However, the process of designing an ISA is often shrouded in secrecy, as many relevant ISAs are proprietary standards developed behind closed doors. In recent years, a disruptive newcomer has emerged in the ISA landscape, known as RISC-V. Unlike other ISAs, RISC-V adopts an open-standard approach, embracing open discussions and decision-making through online forums. The goal of this work is to shed light on the design rationale behind the RISC-V Floating-Point (FP) extensions *F* and *D*. To complement our analysis, we conducted a practical assessment using a profiling RISC-V Virtual Platform (VP) and a comprehensive set of 78 Floating Point (FP) benchmarks and applications. Using this VP, we were able to track and analyze instruction distributions, FP value distributions, and other data of interest. This allows us to draw well-grounded conclusions and gain valuable insights into the characteristics of the RISC-V F/D extensions.**

***Index Terms*—RISC-V, Floating Point, Virtual Platforms, Instruction Set Architecture, Benchmarks**

I. Introduction

Designing an Instruction Set Architecture (ISA) is a demanding and intricate challenge. Wrong decisions may not only have a temporary effect on Key Performance Indicators (KPIs), but can also lead to problems in the longer term due to technical debt. Therefore, it is crucial to make decisions in a reasoned and evidence-based manner. However, especially the latter point is impeded by the availability of publicly accessible data. Due to most market-dominating ISAs being in the hands of private companies, decision rationales are not disclosed to the public. With the emergence of the open standard RISC-V in the last years, the situation of scarce information slowly began to change. Unlike other ISAs, RISC-V embraces an open-standard approach, enabling open discussions and decision-making through online forums. This inclusive approach facilitates the active participation of individuals and organizations in the design process. Nevertheless, especially the design of the RISC-V FP extensions F and D is difficult to comprehend. These extensions were already part of the initial RISC-V ISA [65] in 2011 and thus never part of a public discourse. Only deficiencies and missing specifications were publicly discussed in the following years. Yet, there is no single document summarizing the motivation behind every decision. Rather, information is spread around multiple sources, like Google groups [15], GitHub repositories [16], publications [32], [63], or mailing lists [17]. One of the goals of this work is to summarize the design decisions for the FP extensions F/D and shine light on aspects that were never up to an open debate. To complement our theoretical analysis, we conducted a practical evaluation using a RISC-V Virtual Platform (VP) and an extensive set of 78 FP applications. Through these experiments, we were able to meticulously track and analyze the distribution of instructions, FP values, and other data of interest. Based on this data, we evaluate individual aspects of RISC-V FP extensions and identify possible alternatives and improvements. The collected data should not only help to evaluate decisions retrospectively, but also provide guide rails for future endeavors. Much in the spirit of the RISC-V philosophy, our data is freely accessible [68].

II. Background & History

A. Design & History of the RISC-V F/D Extensions

To cover a wide range of applications, such as embedded systems or high performance computing, the RISC-V ISA provides several so-called *extensions*. Each of these extensions describes a set of properties, like instructions or registers, which can be assembled to larger systems in a modular way. This includes the F/D extensions, which extend RISC-V systems with 32-bit and 64-bit FP arithmetic respectively. The extensions for 16-bit (Zfh) and 128-bit FP arithmetic (Q) are not considered in this work due to their low popularity. Opposed to many other extensions, the F/D extensions were already introduced in the first version of the RISC-V ISA manual [65] in 2011. To a large extent it implements features as mandated in IEEE 754 [63]. In the following, we describe the properties of the F extension, which can be transferred 1-to-1 to D except for the bit width. The F extension adds 32 FP registers, 1 FP Control and Status Register (CSR), and 29 new instructions to RISC-V systems.

B. The Instructions

The heart of any ISA are its instructions. While at the beginning of computer development, there were still significant differences between implementations, today's prevalent FP specifications are similar to a large extent. This is also due to standards such as IEEE 754, which specify the FP formats and instructions to be supported by a conforming ISA. Also RISC-V follows the latest IEEE 754 standard [46] from 2019. Table I highlights the difference between all RISC-V F instructions and their correspondents in x64 and ARMv8. It also reflects the instruction's IEEE 754 2019 status. As shown in the table, the majority of RISC-V instructions are mandated by IEEE 754 and are consequently also prevalent in x64 and ARMv8. Yet there are subtle differences, which are explained in the following. Note that indices in Table I correspond to the subsequent paragraphs.

1) Sign Injection: The three sign injection instructions (FSGNJ, FSGNJN, FSGNJX) were contributed by J. Hauser [66] and are unique to RISC-V [32]. Their main goal is to implement the operations *copy* (FMV in RISC-V), *negate* (FNEG in RISC-V), *abs* (FABS in RISC-V), and *copySign*, which are mandated by the IEEE 2019 standard [46]. This is achieved by transferring the value from rs1 into rd while using a sign based on the following description:

- FSGNJ rd, rs1, rs2: Sign from rs2. Implements *copy* if rs1=rs2.
- FSGNJN rd, rs1, rs2: Negative sign from rs2. Implements *negate* if rs1=rs2.
- FSGNJX rd, rs1, rs2: XORed signs of r1 and r2. Implements *abs* if rs1=rs2.

On x64 systems, the operations *negate* and *abs* are implemented using AND and XOR instructions with a corresponding bitmask.

2) Conversions and Rounding: For every possible conversion from integer to float and vice versa, RISC-V as well as ARM provide the required instructions as mandated by the IEEE 754 standard. The standard also mentions 5 different rounding modes for these instructions. Both ARMv8 and RISC-V allow to directly encode this rounding mode in the instruction. The same approach can be found AVX-512 where it is also possible to encode the rounding mode in the instruction. On x64 systems, the rounding mode has to be set in the FP control and status register. x64 lacks instructions to convert from unsigned 64-bit integer to float and vice versa.

3) Comparisons: While RISC-V provides comparisons, such as equal (FEQ.S) or less than (FLT.S) directly through instructions, ARM and x64 take a different approach. Here, instructions like as FCMP and UCOMISS set flags in status registers which can be used as comparisons in subsequent instructions.

TABLE I
RISC-V INSTRUCTIONS COMPARED TO X64 AND ARMV8.

x64 SSE FMA	ARMv8	RISC-V	IEEE 754 2019
MOVSS	LDR	FLW	mandated
MOVSS	STR	FSW	mandated
VFMADDxxxSS	FMADD	FMADD.S	mandated
VFMSUBxxxSS	FMSUB	FMSUB.S	-
VFNMADDxxxSS	FNMADD	FNMADD.S	-
VFNMSUBxxxSS	FNSUB	FNMSUB.S	-
ADDSS	FADD	FADD.S	mandated
SUBSS	FSUB	FSUB.S	mandated
MULSS	FMUL	FMUL.S	mandated
DIVSS	FDIV	FDIV.S	mandated
SQRTSS	FSQRT	FSQRT.S	mandated
MOVSS[1]	FMOV[1]	FSGNJ.S[1] (FMV.S)	mandated
XORPS[1]	FNEG[1]	FSGNJN.S[1] (FNEG.S)	mandated
ANDPS[1]	FABS[1]	FSGNJX.S[1] (FABS.S)	mandated
MAXSS	FMAX	FMAX	recommended
MINSS	FMIN	FMIN	recommended
CVTSS2SI[2]	FCVT*S[2]	FCVT.W.S[2]	mandated
CVTSS2SI[2]	FCVT*U[2]	FCVT.WU.S[2]	mandated
MOVD	FMOV	FMV.X.W	mandated
UCOMISS[3]	FCMP[3]	FEQ.S[3]	mandated
UCOMISS[3]	FCMPE[3]	FLT.S[3]	mandated
UCOMISS[3]	FCMPE[3]	FLE.S[3]	mandated
-[4]	-[4]	FCLASS.S[4]	mandated
CVTSI2SS	SCVTF	FCVT.S.W	mandated
CVTSI2SS	UCVTF	FCVT.S.WU	mandated
MOVD	FMOV	FMV.W.X	mandated
CVTSS2SI[2]	FCVT*S[2]	FCVT.L.S[2]	mandated
-[2]	FCVT*U[2]	FCVT.LU.S[2]	mandated
CVTSI2SS[2]	SCVTF[2]	FCVT.S.L[2]	mandated
-[2]	UCVTF[2]	FCVT.S.LU[2]	mandated

4) Classification Instruction: An instruction which cannot be found in ARM and x64 is FCLASS. The instruction allows to classify a FP number into several classes as shown in Table II and return the result using a one-hot encoding. This allows to quickly react on the classification by ANDing the result with a bitmask. The instruction is recommended, but not mandated by IEEE 754 1985 [44] and referred to as *Class(x)*. With IEEE 754 2008 [45], it was redeclared as mandatory and renamed to *class(x)*. The classification instruction can be found in other ISAs as well, including Intel i960 (CLASS{R/RL}) [47], LoongArch (FCLASS.{S/D}) [9], IA-64 (FCLASS) [48], and MIPS64 (CLASS.{S/D}, since release 6) [54]. PowerPC implicitly encodes the class for each FP instruction in a register called $\text{FPSCR}_{\text{FPRF}}$ [42]. It is also present in Intel's 80-bit x87 extension as FXAM [49], which is the predecessor of SSE, but Intel decided to remove this instruction from all subsequent extensions. The purpose of the FCLASS instruction is to allow software to react to unusual outputs from other FP instructions with relatively low cycle overhead. In [63] A. Waterman argues that library routines often branch at outputs like NaNs. However, without a designated instruction, this check can take "many more instructions". To what extent cycles are saved is not mentioned. The article also lacks information about how often software uses the FCLASS instruction, and which exact outputs trigger branching. More background data is provided by us in Subsection V-A.

C. The Registers

In addition to the general purpose registers, the RISC-V F extension adds 32 dedicated FP registers with a bit width of FLEN=32 (FLEN=64 for D). During the development of RISC-V, a unified register file was initially considered, but a separate register was ultimately chosen because of the following reasons [63]:

- Some types do not align with the architecture. For example, using the D extension on an RV32 system.
- Allows for recoded formats (internal representation to accelerate handling of subnormal numbers [6]).
- More addressable registers (the instruction implicitly selects a set of registers).
- Natural register file banking simplifying the implementation of superscalar designs.

As explained in [63], a separate register file has the following drawbacks:

- Register pressure increases unless the number of registers is increased.

TABLE II
RETURN TYPES OF THE FCLASS INSTRUCTION.

rd	meaning	rd	meaning	rd	meaning
0	$-\infty$	4	$+0$	8	sNaN
1	-normal	5	+subnormal	9	qNaN
2	-subnormal	6	+normal		
3	-0	7	$+\infty$		

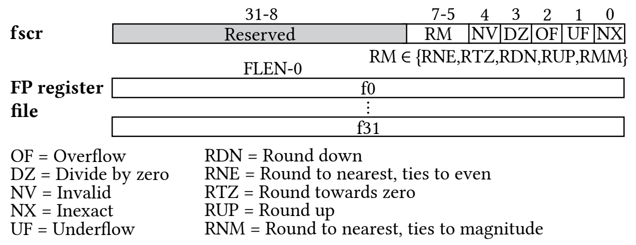

Fig. 1. Register file and FP control and status register of RISC-V.

- Context switching time might increase due to additional register saves. To mitigate this issue, RISC-V introduced dirty flags.

Besides general purpose FP registers, the F extension also adds a CSR to configure rounding modes and indicate FP exceptions (see Fig. 1). The exceptions do not cause traps to facilitate non-speculative out-of-order execution [63].

D. Design of the RISC-V NaN

The FP standard according to IEEE 754 reserves part of the encoding space for so-called Not a Number (NaN). A NaN either represents the result of an invalid operations (quiet Not a Number (qNaN)) or an uninitialized value (signaling Not a Number (sNaN)). According to IEEE 754, a NaN is encoded by a value which has all exponents set to 1, with a non-zero mantissa. The encoding difference between a qNaN and an sNaN was specified in IEEE 754 2008, stating that the MSB in the mantissa functions as a quiet bit. These loosely and changing definitions have led to a variety of NaN encodings among ISAs (see Table III). The initial version of the 2011 RISC-V ISA manual 1.0 [65] states that a canonical NaN has a sign of 0 and all other bits are set. It also specifies that the MSB is the quiet bit with 1 indicating a qNaN, following the IEEE 754 standard in that regard. However, this encoding was changed to ARM's NaN as stated at the 3rd RISC-V Workshop [25] in 2016. It eventually found influence RISC-V ISA manual version 2.1 [64]. According to A. Waterman [63] the new encoding was chosen based on the following arguments:

1) It is the same NaN as used in ARM and Java.
2) Clearing bits has lower hardware cost than setting bits.
3) It is the only qNaN that cannot be generated by quieting an sNaN.

The reason behind the third argument is to distinguish propagated from generated NaNs in case NaN propagation is implemented. Yet, this remains a rather hypothetical argument, as the RISC-V standard does not mandate NaN propagation.

TABLE III
QNANS OF DIFFERENT ISAS.

ISA	sign	significand
SPARC	0	11111111111111111111111
RISC-V $< v2.1$	0	11111111111111111111111
MIPS	0	01111111111111111111111
PA-RISC	0	01000000000000000000000
x64	1	10000000000000000000000
ARM	0	10000000000000000000000
RISC-V $\geq v2.1$	0	10000000000000000000000

E. NaN Boxing

On 2017-03-19, A. Waterman opened a GitHub issue [62], remarking that the undefined of behavior of FP load and store instructions might lead to problems. At that time, storing smaller than FLEN FP values did not have a specified memory layout. For example, if a RISC-V system with D extension loads a 32-bit FP value into register f0, and subsequently stores the register using a 64-bit store, there is no defined memory layout. It is only guaranteed that loading the value from the same address reinstantiates the intended value.

The undefined memory layouts can be problematic in multiple scenarios, as pointed out by A. Bradburry in his RFC [30] on 2017-03-23. For example, when migrating tasks on a heterogeneous System on a Chip (SoC), each core could interpret the FP register file dump differently. To solve this problem, A. Bradburry proposed multiple solutions, which were then discussed in the RISC-V ISA-Dev group [31]. Among the most favored and ISA-compliant approaches were:

- Store 32-bit FP values in the lower half of a 64-bit register. This approach is used by ARM64.
- Cast 32-bit FP values to 64 bit and perform appropriate rounding and masking whenever 32-bit operations are used. Implemented in POWER6 and Alpha.
- Encapsulate 32-bit FP values in a 64-bit FP NaN. Not seen in any architecture before.

After discussing arguments of all approaches, the NaN-boxing scheme was ultimately chosen as the solution and added to the specification on 2017-04-13 [62]. This feature saturates upper bits when working on FP data, which is smaller than the architecture's FP register width FLEN. If the aforementioned RISC-V system loads a 32-bit FP, e.g. 2.5, into register f0, the lower 32 bits of the register represent the FP value, while the upper 32 bits of f0 are set to 1. Hence, the register f0 reads as 0xffffffff40200000. Additionally, a 32-bit value is only considered valid if the upper bits are saturated. Otherwise, the value is interpreted as a negative qNaN. This approach allows for additional debug information, which is not available in other ISAs. As with most ISAs, a FP register file dump does not allow to infer the currently saved data types. However, with NaN boxing, the presence of saturated upper bits allows to determine the data type with high certainty. Because these special NaN values cannot be produced by standard arithmetic instructions, as NaN propagation is not mandated by RISC-V. Yet, there is a risk of confusion with dynamically interpreted languages, which use a software-based NaN boxing for encoding data types.

III. Related Work

Analyzing instruction distributions in applications has been conducted for various ISAs, such as x64 and ARM64. However, to the best of our knowledge, no such analysis has been conducted specifically for RISC-V systems. In literature, two approaches are commonly used to assess instruction distributions. The static analysis approach, as used by [24], [43] simply assesses the instruction occurrences in the binary. However, the results obtained from this method can be misleading, as the number of occurrences does not necessarily indicate how often an instruction is actually executed. Moreover, this approach reaches its limitations for self-modifying code and dynamically interpreted languages. A more accurate and less constrained approach is dynamic analysis, as used in [23], [29], [39]. In dynamic analysis, the instruction distribution is directly obtained from the execution of the benchmark. This can be achieved by counting instructions in a simulator or by utilizing compiler annotations. The latter has the disadvantage of only counting instructions in the application's user mode. Ultimately, the instructions distribution should reflect what is executed on the user's system, including operating system, drivers, and other aspects, which are indirectly related to the executed benchmark. To obtain results that encompass all executed instructions and side effects, a simulator-based approach, as utilized in [29], [39], proves to be one of the few viable methods. This is why we also conducted all experiments on a profiling RISC-V simulator.

IV. Methods

A. The Virtual Platform

To execute all 78 benchmarks, we used MachineWare's RISC-V simulator SIM-V [50]. The simulator was configured to model a rv64imafdc CPU as part of a VP with 4 GiB of main memory. For most benchmarks the VP runs an Ubuntu 22.04 operating system. Some benchmarks run on a minimal buildroot-configured Linux. The simulator was modified to track the number of executed instructions and other data of interest. To not accidentally track boot or non-benchmark related instructions, the simulator was extended by semihosting instructions that allow to reset and dump the statistics. That means, before the execution of a benchmark, the statistics were reset, which was followed by a dump after the execution finished. In contrast to compiler-based annotation, as for example in used in gcov [4], a simulator-based approach allows to track every detail, reaching from instructions in the kernel to closed-source libraries. Although the proprietary simulator SIM-V was used for profiling, other open-source simulators could also be adapted for this use case. Due to its performance, QEMU [26] would be the most suitable. The simulators Spike [34] and gem5 [27] are also conceivable, but their relatively slow, interpreter-based execution requires a significant amount of compute time.

B. The Benchmarks

The main criterion for the selection of the benchmarks was the use of FP instructions. That is, all benchmarks listed in Fig. 2 execute at least one FP instruction. Another concern was that benchmarks should cover a variety of scenarios. From high-performance computing (NPB [11]) over machine learning (OpenNN [14]) to graphics computation (glmark2 [5]); a large spectrum of different applications is reflected in chosen benchmarks. Additionally, attention was paid to the presence of different programming languages. Depending on the language, different peculiarities in the FP arithmetic can arise. Therefore, we chose benchmarks in Fortran (NPB [11]), Javascript (Octane 2.0 [13]), C++ (FinanceBench [3]), Python (NumPy [40]), and other programming languages. The complete list of benchmarks can be found in Fig. 2. In total, the 78 benchmarks executed 80,653,539,756,271 instructions of which 16,824,921,642,417 were part of the F/D extensions.

SPEC CPU 2017 [20]
(1) 503.bwaves
(2) 507.cactuBSSN
(3) 508.namd
(4) 510.parest
(5) 511.povray
(6) 519.lbm
(7) 527.cam4
(8) 538.imagick
(9) 544.nab
(10) 549.fotonik3d
(11) 554.roms

FinanceBench [3]
(43) Black Scholes
(44) Bonds
(45) Monte Carlo

mibench [39]
(60) basicmath
(61) susan
(62) qsort
(63) lame
(64) rsynth
(65) fft

glmark2 [5]
(18) buffer
(19) build
(20) bump
(21) clear
(22) conditionals
(23) desktop
(24) effect2d
(25) function
(26) ideas
(27) jellyfish
(28) loop
(29) pulsar
(30) refract
(31) shading
(32) shadow
(33) terrain
(34) texture

NumPy [40]
(48) linalg
(49) scalar

Octane 2.0 [13]
(46) raytrace
(47) navierstoke

NPB [11]
(12) NPB.bt.A
(13) NPB.cg.A
(14) NPB.ep.A
(15) NPB.ft.A
(16) NPB.mg.A
(17) NPB.sp.A

Other
(66) fbench [60]
(67) ffbench [61]
(68) linpack32 [8]
(69) linpack64 [8]
(70) whetstone [22]
(71) stream [21]
(72) lenet-infer
(73) alexnet-train
(74) cray [1]
(75) aobench [35]
(76) glxgears
(77) himeno [7]
(78) SciMark 2.0 [18]

OpenNN [14]
(35) iris_plant
(36) breast_cancer
(37) simple_approx
(38) simple_class
(39) logical_operations
(40) airfoil
(41) mnist
(42) outlier_detection

CoreMark-PRO [2]
(50) loops-all-mid-10k
(51) linear_alg-mid-100x100
(52) nnet_test
(53) radix2-big-64k

smallpt [19]
(54) smallpt-c
(55) smallpt-cpp
(56) smallpt-java
(57) smallpt-erlang
(58) smallpt-numpy
(59) smallpt-python3

Fig. 2. All 78 benchmarks and their corresponding index for this work.

V. Results & Discussion

A. Instruction Distributions

A heatmap depicting the relative distribution of FP instructions per benchmark can be found in Fig. 5. Note that this and other figures treat the F and D extension as a single entity. For example, FLX refers to the sum of FLS (32 bit) and FLD (64 bit). It can be seen that FP store and load instructions are the most prevalent instruction in nearly every benchmark. This stands in contrast to instructions such as FCLASS, FMIN, or FMAX, which are often not executed once. To get an overview of frequently and rarely seen instructions, we accumulated the instruction distributions for all benchmarks (see Fig. 3). The relative contribution of each benchmark can be inferred from Fig. 4. Fig. 3 shows that RISC-V FP instructions seem to follow an exponential distribution. The instructions FLX (32%), and FSX (17%), sum up to nearly 50% of all executed FP instructions. On the other side of the spectrum, the FCLASS instruction only occurs once every 13,812 instructions. On top of that, only 12 out of 78 benchmarks used this instruction at all. This raises the question whether such an instruction should be part of a RISC ISA. In the following, we provide answers by analyzing the contexts of this instruction, considering possible alternatives, and evaluating its hardware cost.

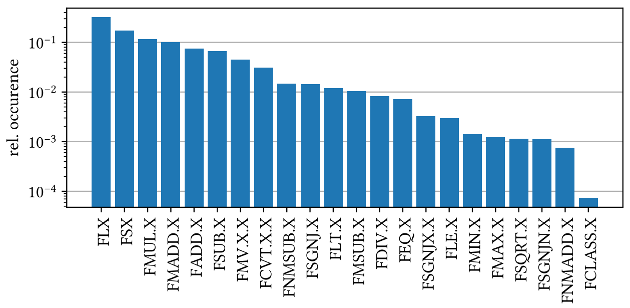

Fig. 3. Distribution of FP instructions accumulated over all benchmarks.

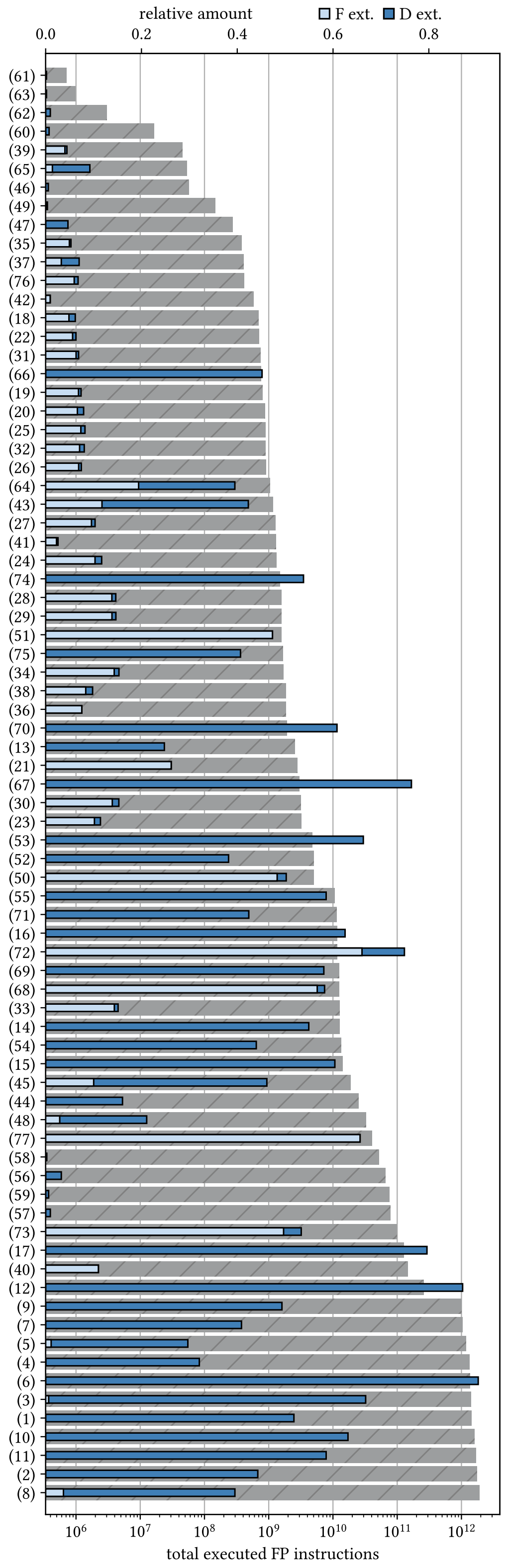

Fig. 4. Relative distribution of F (light blue) and D (blue) instructions and total number of executed instructions (grey).

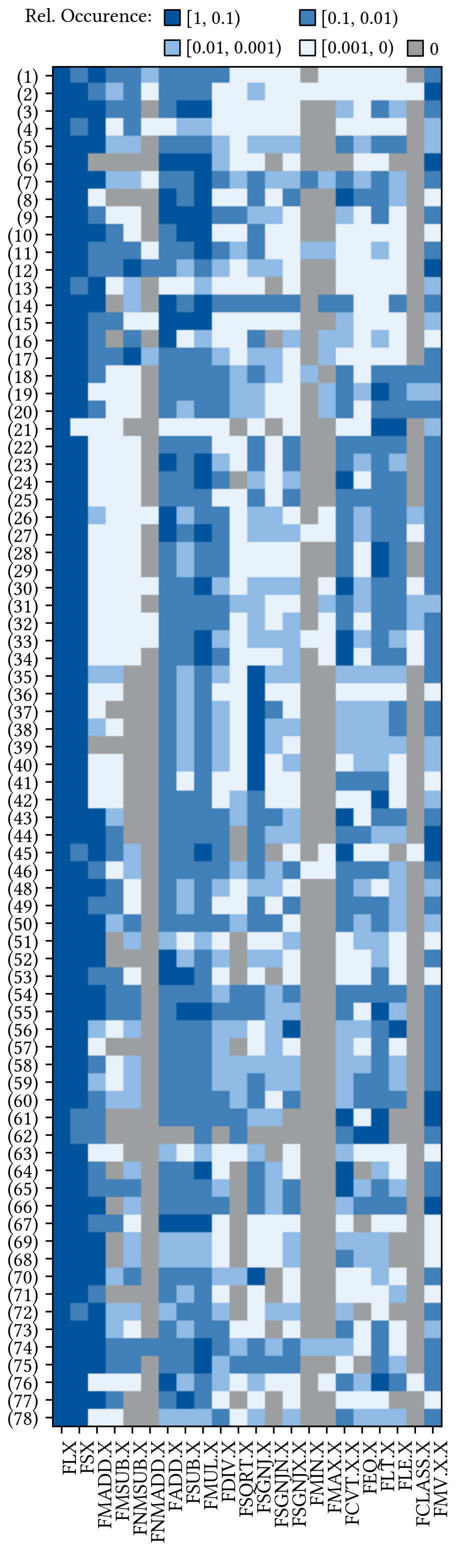

Fig. 5. Relative distribution of FP instructions per benchmark.

```
float __fmaxf(float x, float y) {
  float r;
  if ((_FCLASS (x) | _FCLASS (y)) & _FCLASS_SNAN)
    return x + y;

  asm ("fmax.s␣%0,␣%1,␣%2" : "=f"(r) : "f"(x), "f"(y));
  return r;
}
```

Code 1. glibc implementation of std::fmax for RISC-V.

1) Abundance of FCLASS: As shown before, the FCLASS instruction occurs infrequently, with many benchmarks not only using it once. The benchmark glmark2-bump attains the highest relative value, with 0.091% of all instructions being FCLASS. Besides being present in all glmark benchmarks, it also occurs in FinanceBench and SPEC 507.cactuBSSN. Since FCLASS can appear in different contexts, we investigated the reasons for its use in the benchmarks.

For all benchmarks, we could track down all usages of the FCLASS instruction to glibc's *fmax/fmin* function. The corresponding C implementation for 32-bit FP is depicted in Code 1. Here, one would intuitively expect only a RISC-V *fmax* instruction, yet there are additional checks for sNaNs. This is due to RISC-V adhering to the IEEE 754 standard from 2019, where the maximum of an sNaN and numerical value must return the latter. In glibc, however, this operation has to return a qNaN, making it compliant with older IEEE 754 standards. To rectify this mismatch, additional checks and treatments for sNaN are needed. As explained by D. G. Hough [41], converting sNaN to qNaN in minimum/-maximum functions, as in glibc and older IEEE 754 standards, was a bug in the specification and entails awkward mathematical properties. The fix from IEEE 754 2019 is not yet present in glibc. Other C standard libraries, such as musl [10] or NewLib [12] directly map *fmax* and *fmin* to the underlying ISA implementations inheriting their NaN-handling characteristics. That means, if we link the benchmarks against musl or NewLib instead of glibc, the number of executed FCLASS instructions can be reduced to 0. Or in other words, using this approach, FCLASS does not occur once in 78 benchmarks executing trillions of instructions. Even though the IEEE 754 2019 standard mandates this instruction, we recommend reconsidering its usage in the RISC-V ISA. We also believe Intel came to the same conclusion, which is why extensions after x87 do not include this instruction. Yet, excluding the FCLASS instruction is associated with only little benefits in terms of hardware cost. For a Synopsys ASIP Designer RISC-V processor synthesized with Synopsys Design Compiler and a 28/32nm library, the FCLASS instruction accounted for 0̃.25% of the FPU's area excluding register file.

Before removing an instruction from an ISA, it should be investigated, which combination of other instructions achieves the same semantics. However, in the case of FCLASS, it is not necessary to aim for a bit-exact reproduction. As mentioned by A. Waterman [63], the purpose of FCLASS is to branch if exceptional values such as NaN are encountered. Fig. 6 shows both a typical assembly context for sNaN and alternatives without FCLASS. As can be seen, a check for a certain FP type using FCLASS requires 3 instructions. First, FCLASS returns the value type in a one-hot encoding, then the type of interest is extracted by bitmasking, and finally a branch is taken depending on the previous result. If FCLASS is not used, the same goals can be achieved with even less instructions (see positive zero, or general NaN). For example, we can exploit that comparisons with NaN values always return false, allowing us to check for their presence in only one instruction. Similar to FCLASS, all instructions used in Fig. 6 are also lightweight and do not require any data memory accesses.

```
// fclass sNaN example
fclass.s x1, f0
andi x1, x1, 0x100
bnez x1, lbl-is-snan

// zero
fmv.w.x f1, x0
feq.s x1, f1, f0
bnez x1, lbl-is-zero

// positive zero
fmv.x.w x1, f0
bez x1, lbl-is-p-zero

// negative zero
fneg f0, f0
fmv.x.w x1, f0
bez x1, lbl-is-n-zero
// generic NaN
feq.s x1, f0, f0
beqz x1, lbl-is-nan

// quiet NaN
fmv.x.w x1, f0
lui x2, 0x7fc00
and x1, x1, x2
beq x1, x2 lbl-is-qnan

// signaling NaN
feq.s x1, f0, f0
fmv.x.w x2, f0
bexti x2, x2, 22
or x1, x1, x2
beqz x1, lbl-is-snan
```

Fig. 6. Typical FCLASS use case and FCLASS-less alternatives.

B. Value Distribution

1) Subnormal Numbers & Underflows: Subnormal numbers and gradual underflow are one of the most controversial features of the IEEE 754 standard [52]. On the one hand, subnormal numbers bring numerically advantageous properties like Sterbenz' lemma [58], on the other hand, they increase hardware cost, and their implementation is considered the most challenging task in Floating Point Unit (FPU) design [57]. As shown by numerous works, handling subnormal numbers can reduce a CPU's attainable throughput by more than 100× [28], [33], [67]. Due to this possible performance degradation, Intel introduced the so-called Flush To Zero (FTZ) mode with the release of Streaming SIMD Extensions (SSE) in 1999 [59]. This mode allows to flush subnormal numbers to zero, increasing the performance of applications with non-critical accuracy requirements like real-time 3D applications. However, how often subnormal numbers and underflows occur in practice is not stated in any of the aforementioned works. Also other works only provide anecdotal evidence and statements like "gradual underflows are uncommon" [51].

Our results confirm that underflows and subnormals are rather an exception. Out of 78 benchmarks, 60 did not raise an underflow exception or have a single subnormal input operand. The highest share of underflows occurs in MiBench susan with 0.48% of all arithmetic instructions underflowing. The relative number of underflows per benchmark is depicted in Fig. 9. Accumulated over all benchmarks, underflows occurred once every 7,992 instructions with subnormal inputs occurring once every 3,875 operands. In summary, only a fraction of FP applications would benefit from a FTZ mode. To what

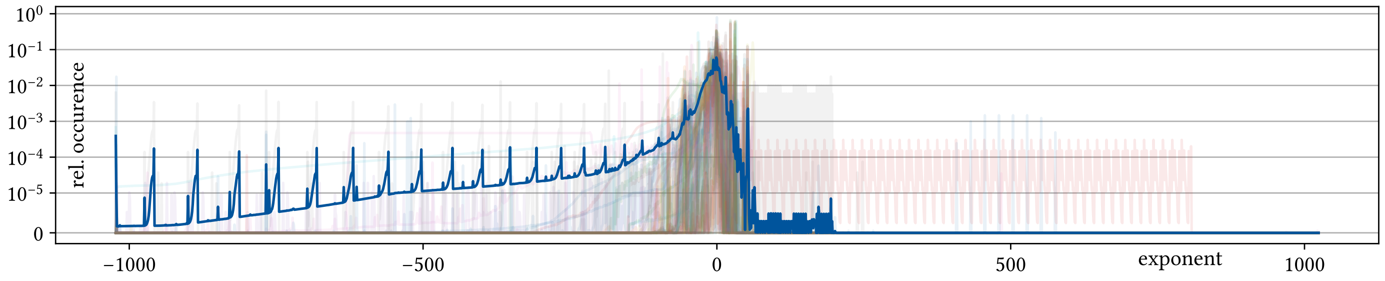

Fig. 7. Relative distributions of FP in and out values for 64-bit arithmetic instructions in the individual benchmarks. The blue line represents the average.

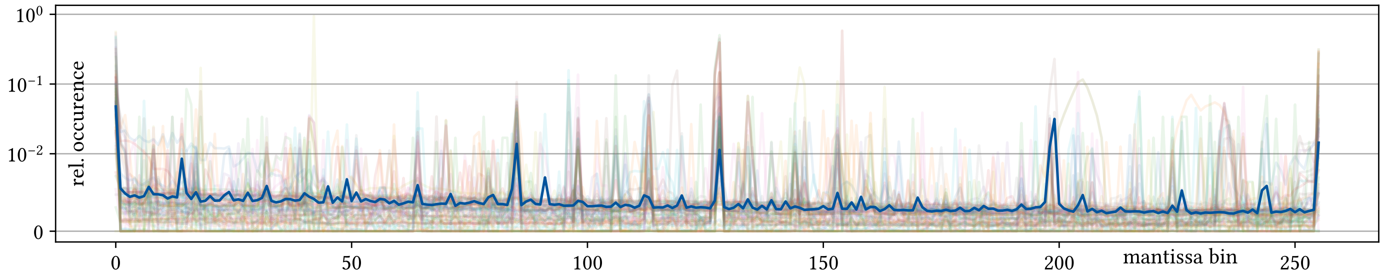

Fig. 8. Distribution of mantissas for arithmetic 64-bit instructions. The values were distributed into 256 different bins. The average is given by the blue line.

extent performance can be increased, ultimately depends on the hardware implementation. Since RISC-V has a separate FP register file, techniques such as recoded formats [6] can be used to process subnormal numbers efficiently. Running the subnormal arithmetic evaluation benchmark by Dooley et al. [33] on the RISC-V-based StarFive VisionFive 2 shows no performance degradation due to subnormal arithmetic. Thus, the decision not to endow RISC-V with a FTZ mode, as in ARMv8 or x64, is reasonable in our opinion.

2) Exponent Distribution: Although IEEE 754 FP is the most widespread approximation of real numbers in computing, other formats can be considered as well. A frequent criticism of IEEE 754 FP is its nearly uniformly distributed accuracy. According to many works [36]–[38], [55], most values in practical applications are centered around 1, which is why formats with tapered accuracy, like Gustafson's posit [38], accumulate less error in benchmarks. See Fig. 10, which qualitatively depicts the accuracy of IEEE FP and posit. Since our simulator captures the value ranges of all FP instructions, we can subject this statement to thorough testing.

In Fig. 7 we present the exponent distribution of all in- and outputs of 64-bit arithmetic instructions. As can be seen, the values of most benchmarks are indeed centered near a magnitude of $2^0 = 1$. This provides initial evidence that formats with a tapered accuracy, like the proposed RISC-V extension *Xposit* from [53], can reduce rounding errors for a wide range of applications.

3) Mantissa Distribution: As shown in [56], some theoretical considerations, like finding the optimal radix for a number format, require assumptions about the average distribution of the mantissa. One reasonable choice is a logarithmical distribution of mantissa values. As shown by R. W. Hamming, arithmetic operations transform various input distributions to a logarithmic distribution. Excluding several hotspots, also the mantissa distributions of all executed benchmarks in Fig. 8 seem to follow this trend.

VI. Conclusion & Outlook

In this work we showed how a modified RISC-V simulator can be used to analyze the characteristics of the RISC-V Floating Point (FP) extensions F and D. In total our simulator executed more than 16 trillion FP instructions of 78 applications, precisely tracking the distribution of FP of instructions, FP mantissa, FP exponent, and frequency of underflows. As our most important finding, we saw a significant underutilization of RISC-V's FCLASS instruction. Moreover, we could pinpoint all FCLASS usages to glibc's questionable implementation of the maximum and minimum functions. If other C standard libraries than glibc are used, the number of executed FCLASS instructions can be reduced to 0. We also presented lightweight alternatives to the FCLASS instruction by emulating common uses cases with a handful of other RISC-V instructions. Consequently, we recommend to reconsider the role of FCLASS in the RISC-V FP extensions.

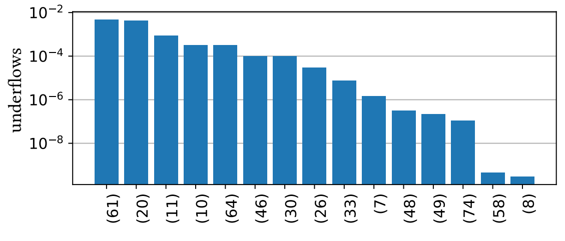

Fig. 9. Benchmarks with the highest relative share of underflows.

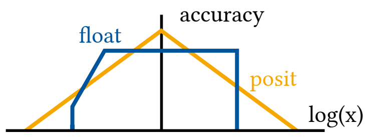

Fig. 10. Qualitative depiction of floating point and posit accuracy.

REFERENCES

[1] “c-ray,” https://github.com/jtsiomb/c-ray, accessed: 2023-07-13.
[2] “CoreMark®-PRO,” https://github.com/eembc/coremark-pro, accessed: 2023-07-13.
[3] “FinanceBench,” https://github.com/cavazos-lab/FinanceBench, accessed: 2023-07-13.
[4] “gcov,” https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, accessed: 2023-07-13.
[5] “glmark2,” https://github.com/glmark2/glmark2, accessed: 2023-07-13.
[6] “HardFloat Recoding,” www.jhauser.us/arithmetic/HardFloat-1/doc/HardFloat-Verilog.html, accessed: 2023-07-13.
[7] “Himeno Benchmark,” https://github.com/kowsalyaChidambaram/Himeno-Benchmark, accessed: 2023-07-13.
[8] “linpack,” https://www.netlib.org/linpack/, accessed: 2023-07-13.
[9] “LoongArch Reference Manual Volume 1: Basic Architecture.”
[10] “musl,” https://musl.libc.org/, accessed: 2023-07-13.
[11] “NAS Parallel Benchmarks,” https://www.nas.nasa.gov/software/npb.html, accessed: 2023-07-13.
[12] “Newlib,” https://sourceware.org/newlib/, accessed: 2023-07-13.
[13] “Octane 2.0,” https://github.com/chromium/octane, accessed: 2023-07-13.
[14] “OpenNN Examples,” https://github.com/Artelnics/opennn/tree/master/examples, accessed: 2023-07-13.
[15] “RISC-V ISA Dev Google Group,” https://groups.google.com/a/groups.riscv.org/g/isa-dev, accessed: 2023-07-13.
[16] “RISC-V ISA Manual Github Repository,” https://github.com/riscv/riscv-isa-manual, accessed: 2023-07-13.
[17] “RISC-V Working Groups Mailing List,” https://lists.riscv.org/g/main, accessed: 2023-07-13.
[18] “SciMark 2.0,” https://math.nist.gov/scimark2/, accessed: 2023-07-13.
[19] “smallpt,” https://github.com/matt77hias/smallpt, accessed: 2023-07-13.
[20] “SPEC CPU 2017,” https://spec.org/cpu2017/, accessed: 2023-07-13.
[21] “STREAM benchmark,” https://www.cs.virginia.edu/stream/, accessed: 2023-07-13.
[22] “whetstone,” https://netlib.org/benchmark/whetstone.c, accessed: 2023-07-13.
[23] “Analysis of x86 instruction set usage for dos/windows applications and its implication on superscalar design,” in *Proceedings of the International Conference on Computer Design*, ser. ICCD ’98. USA: IEEE Computer Society, 1998, p. 566.
[24] A. Akshintala, B. Jain *et al.*, “X86-64 instruction usage among c/c++ applications,” in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 68–79. [Online]. Available: https://doi.org/10.1145/3319647.3325833
[25] K. Asanovic, “3rd RISC-V Workshop: RISC-V Updates,” https://riscv.org/wp-content/uploads/2016/01/Tues1000-RISCV-20160105-Updates.pdf, 01 2016, accessed: 2023-07-13.
[26] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator.” 01 2005, pp. 41–46.
[27] N. Binkert, B. Beckmann *et al.*, “The Gem5 Simulator,” *SIGARCH Comput. Archit. News*, 2011.
[28] J. Bjørndalen and O. Anshus, “Trusting floating point benchmarks - are your benchmarks really data independent?” 06 2006, pp. 178–188.
[29] N. Bosbach, L. Jünger *et al.*, “Entropy-based analysis of benchmarks for instruction set simulators,” in *RAPIDO2023: Proceedings of the DroneSE and RAPIDO: System Engineering for constrained embedded systems*, ser. RAPIDO2023. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 54–59, toulouse, France.
[30] A. Bradbury, “NaN Boxing RFC,” https://gist.github.com/asb/a3a54c57281447fc7eac1eec3a0763fa, accessed: 2023-07-13.
[31] ——, “NaN Boxing ISA-Dev Group,” https://groups.google.com/a/groups.riscv.org/g/isa-dev/c/_r7hBlzsEd8/m/z1rjr2BaAwAJ, 03 2017, accessed: 2023-07-13.
[32] T. Chen and D. A. Patterson, “RISC-V Geneology,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-6*, 2016.
[33] I. Dooley and L. Kale, “Quantifying the interference caused by subnormal floating-point values,” 01 2006.
[34] R.-V. Foundation, “Spike RISC-V ISA Simulator,” https://github.com/riscv-software-src/riscv-isa-sim, accessed: 2023-09-15.
[35] S. Fujita, “aobench,” https://github.com/syoyo/aobench, accessed: 2023-07-13.
[36] A. Guntoro, C. De La Parra *et al.*, “Next generation arithmetic for edge computing,” in *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2020, pp. 1357–1365.
[37] J. Gustafson, “Posit arithmetic,” *Mathematica Notebook describing the posit number system*, 2017.
[38] J. Gustafson and I. Yonemoto, “Beating floating point at its own game: Posit arithmetic,” *Supercomputing Frontiers and Innovations*, vol. 4, pp. 71–86, 06 2017.
[39] M. Guthaus, J. Ringenberg *et al.*, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.
[40] C. R. Harris, K. J. Millman *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2
[41] D. G. Hough, “The ieee standard 754: One for the history books,” *Computer*, vol. 52, no. 12, pp. 109–112, 2019.
[42] IBM, “PowerPC User Instruction Set Architecture Book I Version 2.01,” 2003.
[43] A. H. Ibrahim, M. B. Abdelhalim *et al.*, “Analysis of x86 instruction set usage for windows 7 applications,” in *2010 2nd International Conference on Computer Technology and Development*, 2010, pp. 511–516.
[44] “IEEE Standard for Binary Floating-Point Arithmetic,” IEEE, 1985.
[45] “IEEE Standard for Floating-Point Arithmetic,” IEEE, 2008.
[46] “IEEE Standard for Floating-Point Arithmetic,” IEEE, 2019.
[47] Intel, “80960KB Programmer’s Reference Manual.”
[48] ——, “Intel® IA-64 Architecture Software Developer’s Manual Volume 3: Instruction Set Reference,” 2000.
[49] ——, “Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture,” 2016.
[50] L. Jünger, J. H. Weinstock *et al.*, “SIM-V: Fast, Parallel RISC-V Simulation for Rapid Software Verification,” *DVCON Europe 2022*.
[51] W. M. Kahan, “Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic,” https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF, accessed: 2023-07-13.
[52] W. M. Kahan and C. Severance, “An Interview with the Old Man of Floating-Point,” https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html, accessed: 2023-07-13.
[53] D. Mallasén, R. Murillo *et al.*, “Percival: Open-source posit risc-v core with quire capability,” *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1241–1252, 2022.
[54] MIPS, “MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual Revision 6.05,” 2016.
[55] R. Morris, “Tapered floating point: A new floating-point representation,” *IEEE Transactions on Computers*, no. 12, pp. 1578–1579, 1971.
[56] J.-M. Muller, N. Brisebarre *et al.*, *Handbook of Floating-Point Arithmetic*, 01 2010.
[57] E. Schwarz, M. Schmookler *et al.*, “FPU implementations with denormalized numbers,” *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 825–836, 2005.
[58] P. H. Sterbenz, “Floating-point computation,” 1973.
[59] S. Thakkur and T. Huff, “Internet Streaming SIMD Extensions,” *Computer*, vol. 32, no. 12, pp. 26–34, 1999.
[60] J. Walker, “fbench,” https://www.fourmilab.ch/fbench/fbench.html, accessed: 2023-07-13.
[61] ——, “ffbench,” https://www.fourmilab.ch/fbench/ffbench.html, accessed: 2023-07-13.
[62] A. Waterman, “NaN Boxing Github Issue,” https://github.com/riscv/riscv-isa-manual/issues/30, accessed: 2023-07-13.
[63] ——, “Design of the RISC-V Instruction Set Architecture,” 2016.
[64] A. Waterman, Y. Lee *et al.*, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1,” California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, Tech. Rep., 2016.
[65] ——, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA, Version 1,” *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.
[66] ——, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2,” 2017.
[67] M. Wittmann, T. Zeiser *et al.*, “Short note on costs of floating point operations on current x86-64 architectures: Denormals, overflow, underflow, and division by zero,” 06 2015.
[68] N. Zurstraßen, “Instruction and FPU Traces,” https://chciken.com/assets/risc-v_floating_point/traces.zip, accessed: 2023-07-13.