An Open-Source RISC-V Vector Math Library

Ping Tak Peter Tang
Riovs Inc., Santa Clara, CA 95954

***Abstract*—RISC-V is an open royalty-free instruction set architecture (ISA) under the governance of RISC-V International. The ISA consists of a base instruction set and many modular extensions among which the vector extension that supports data parallel execution is recently ratified. Supporting the goal of an open ISA to accelerate innovation, we developed an open-source double-precision vector math library as a highly relevant addition to the high-performance computing (HPC) ecosystem. This paper introduces the RISC-V vector ISA, some of those computational instructions we found most useful, and the software structure of, as well as the various numerical techniques employed to make the library accurate and efficient.**

I. Introduction

RISC-V is an open royalty-free instruction set architecture (ISA) that was first developed at the University of California at Berkeley in 2011 [1] (c.f. [2]). One distinguished feature is its being modular and extensible. The base integer instruction set RV32I was released in 2011. This was followed by a number of extensions such as the single-precision and double-precision floating-point extensions. Vendors can implement the base ISA together with whichever additional extensions they deem relevant. Furthermore, RISC-V envisions new ISA extensions to be proposed by practitioners of various specialized areas. In high-performance computing (HPC), SIMD instructions in various architectures have proved successful in boosting performance through data-parallel processing. Consequently, RISC-V approved a vector "V" extension v1.0 [3].

Realizing RISC-V's vision that open technology accelerates innovation in computing requires a diverse and robust ecosystem. To support the HPC ecosystem, we developed an open-source vector math library[1] that richly utilizes the aforementioned "V" vector extension.

There are several purposes of our work. First and most obviously, we aim to provide an accurate and good performing open-source library as a common building block in HPC, enriching the ecosystem and improving the chance of RISC-V's success. Since we target HPC, we focus on IEEE double precision and defer work on other floating-point precisions. Second, this library presents concrete use cases of the RISC-V vector ISA as a validation of the ISA design. To this end, for some of the functions in this library, we implemented multiple versions using different algorithms thus exercising different instructions combinations. Having different versions is also helpful at this stage when different HPC-class hardware implementations are yet to emerge. Last but not least, the code contains extensive comments on the various numerically subtle computations. The goal is to enable fellow travellers in the ecosystem to more easily further contribute to this library as well as constructing other numerical software for RISC-V.

In what follows, Section II discusses some related prior art in math libraries. Section III introduces some key features of the RISC-V vector ISA. Section IV describes our vector math library: its software architecture and some key techniques employed in the functions' implementations. Section V demonstrates concretely through selected functions how we employ these techniques, followed by a summary of accuracy measurements. We reflect on our experience with the RISC-V vector ISA from this work and make some concluding remarks in Section VI.

II. Related Works

The "libm" math library has been part of the infrastructure available to standard programming environments. In the C programming language for example, a program can access these functions by the definitions in `math.h` and linking with `−lm`. Since the first adoption the IEEE-754 standard in 1985 (the latest revision is in [4]), reliable floating-point arithmetic has become the norm as practically all platforms conform to the standard. As a result, libm libraries evolved to handle IEEE special values such as `NaN` or `Inf` and endeavored to deliver functions with quantified accuracy goals. In addition to proprietary software each platform or vendor supplies to its users, open-source libraries also started to emerge. One such early example is fdlibm [5] which was developed and freely distributed by Sun Microsystems, Inc. This library handles IEEE exceptions faithfully. First, it returns appropriate results and exception signals to inputs that warrant them: for example, returning `−Inf` and raising the division-by-zero signal for `log(0.0)`. Second, it carefully avoids raising exceptional signals spuriously. For example, the `log` function of the largest finite input is a very moderate number; but it is conceivable that a program can generate a spurious overflow signal in handling an input number that large. The library states that a few core functions such as `exp` or `log` are provably accurate to within one unit of last place (ulp) [6] but does not maintain this accuracy or makes any explicit claims for most other functions, such as the hyperbolic or inverse trigonometric functions. Glibc [7] contains an open-source libm. The quality is less uniform as it is contains contributions by different developers. For example, within the `ieee754` directory of its math library in Glibc 2.26 includes some correctly rounded functions contributed by IBM, some of the less accurate functions from fdlibm, and some other functions developed by Glibc itself. The OCML [8] library contains complete source codes to a libm supporting LLVM. While

[1]`https://github.com/rivosinc/veclibm`

there are no accuracy claims, the double-precision library uses extra-precision double-double [9] simulations generously and is expected to be adequately accurate. This library handles exceptions via post-processing: It first carries out computations for all input arguments, and checks afterwards for exceptional or extreme input values, revising when needed the previously computed results. Post-processing can raise spurious exceptions. For example, $\texttt{exp}(+\texttt{Inf})$ should return $+\texttt{Inf}$ without exceptions. The general computation however usually computes $r = x - \texttt{round\_int}(x/log2)\log(2)$ in $\texttt{exp(x)}$. If `round_int` produces $+\texttt{Inf}$, then a spurious invalid operation signal will be generated subtracting two $+\texttt{Inf}$s. The libraries discussed so far consist of scalar functions that generally take and return a single floating-point number.

On computing platforms that support a SIMD ISA, vector math libraries are often provided that take an array of floating-point input arguments and return an array containing the corresponding function values. Intel Vector Math Library [10] is Intel SIMD specific (e.g. SSE, AVX256, AVX512) and closed source, though parts of it are contributed to Glibc. The Arm library in [11] is open source. It has a number of scalar functions and also four double-precision vector functions: exp, log, sin and cos, excluding the vector pow function as it calls the scalar version within a loop. Vector Libm [12] contains 9 double-precision functions coded in a scalar fashion intended to be automatically vectorized by compilers. This approach trades ISA agnosticism with performance. Some of these libraries have explicit accuracy claims and most are around 4 ulps. Comments within the source codes in [11] suggest their accuracies are within 2 ulps. The open-source vector function library SLEEF [13] is an excellent contribution. It contains a rather complete collection of double-precision elementary functions: the `exp` and `log` functions in three bases, `expm1` and `log1p` and the power function `pow`; the three trigonometric functions and their inverses and the `atan2` function; the `sin` and `cos` functions and their inverses have the "pi" versions in additional to the standard radian version; the three hyperbolic functions and their inverses; the cube root function. Most of these functions have a more accurate 1 ulp and a relaxed 3.5 ulps versions. The stated accuracies were assessed empirically. The accuracy level of the single-version functions are unspecified. Similar to OCML, exceptional/extreme inputs are post-processed and thus spurious exception signals can be raised at times. SLEEF uses a set of library defined intrinsic functions that serve as an abstraction layer. This allows it to support different SIMD ISAs via appropriate implementations of these intrinsic functions. This portability is highly valuable. Nevertheless, one would expect some performance trade off as it is difficult for a general abstraction to fully exploit every single specific ISA. Indeed, SiFive reported this exact experience when they ported the SLEEF library to their platform [14]; and their platform specific library can better utilize their hardware. That paper does not fully discuss the status or availability of their library.

Our vector math library is RISC-V specific. The library currently contains all the elementary functions included in SLEEF, and additionally the "pi" version for tangent and its inverse. Some functions have multiple algorithmic versions, but are all assessed empirically to be within 1 ulp accurate. We plan to apply rigorous error analysis in the future. Some common special functions such as gamma, error, Gaussian distribution are under development. It handles exceptional and extreme inputs faithfully without generating spurious exceptions. We expect the computing industry to embrace the RISC-V open ISA and thus notwithstanding our library's single mindedness, it will serve many compute platforms.

III. RISC-V Vector ISA Basics

A vector ISA specifies instructions that operate on vector registers, each of which holds multiple elements. A single issue of an instruction thus executes the same operation on each of these multiple sets of operands, gaining the name of single-instruction-multiple-data (SIMD) instructions. Vector operations work perfectly for computational kernels that can exploit data parallelism; adding two long vectors in computational linear algebra is the "Hello, World!" example of this.

Traditional SIMD ISA such as Intel's AVX requires the program to know explicitly the number L of elements that a vector register holds. A SIMD implementation for an N-length vector addition typically "strips" L elements off this vector per iteration. Strip mining here typically requires two loops. The first processes L elements per iteration. The second (often called an epilogue) processes the remaining $N\%L$ elements in a scalar fashion.

In contrast, RISC-V vector ISA is register-length agnostic and has a feature that renders strip-mining epilogue unnecessary. RISC-V vector ISA defines 32 architectural vector registers but the size `VLEN` of these registers can be implementation chosen as 2^k bits provided that $\texttt{VLEN} \leq 2^{16}$. `VLEN` needs to be at least as big as `ELEN`, the maximum size in bits of a vector element that any operation can produce or consume. Moreover, some tool chain may impose other lower bounds: LLVM requires that $\texttt{VLEN} \geq 128$. As an example, if `VLEN` is 512, then each physical vector register can hold 8 IEEE double-precision elements.

The vector registers are general purpose, used for integer, floating-point and logical (mask/predicate) variables. The size of the elements together with the instruction unambiguously determine how the bits of the elements inside the general-purpose registers are to be interpreted. All vector instructions can operate under masks. Going beyond the floating-point vector instructions, there are vector instructions that operate on fixed-point values which our work makes good use of (c.f. Section IV-C4).

In addition to the physical registers, one can group multiple (or even fractional) physical vector registers to form one logical register. This is specified by the parameter `LMUL`. For our work here, we only consider non-fractional $\texttt{LUML} = 1, 2, 4$ or 8. If `VLEN` is 512 and `LMUL` is 2, then each logical vector register can hold 16 double-precision elements and thus one single vector floating-point add instruction executes 16 additions. A same program can optimize its performance

```
// Adding two N-length vector X and Y
// Using C intrinsic with comments
n_remain = N;
for (; n_remain>0; n_remain-=v_ell) {
 v_ell = __riscv_vsetvl_e64m2(n_remain);
 // C intrinsic for the vsetvl instruction
 // e64 means 64-bit element (FP64 e.g.)
 // m2 means LMUL=2
 // vsetvl sets the vl register to # active
 // elements in the logical vector register
 // and assigns that value to v_ell
 vx = __riscv_vle64_v_f64m2(X,v_ell);
 vy = __riscv_vle64_v_f64m2(Y,v_ell);
 // load v_ell arguments into registers vx, vy
 // starting from address X and Y.
 // Note that the intrinsic function takes v_ell
 // as an input and "resets" the vl register
 // Compilers are expected to exploit knowledge
 // of the vl register's content and skip the reset
 // whenever possible
   ... process, store result, etc. ...
 X += v_ell; Y += v_ell;
 // v_ell is usually the number of elements that
 // the logical register can hold
 // at the last iteration v_ell becomes N%L
 // if N is not a multiple of L
}
```

Listing 1. One loop for strip mining

on different hardware implementations by changing `LMUL`. Note however, the number of logical registers available will naturally be just $32/\texttt{LMUL}$ whenever $\texttt{LMUL} > 1$.

Two special registers are used to dynamically configure the vector registers that greatly eases programming efforts: (1) the `vtype` register specifies the bit width of each element (e.g. 32 for IEEE single-precision data or 32-bit integers) and the register grouping parameter `LMUL` and (2) the `vl` register that specifies the number of elements (starting at element 0) inside the logical register to be operated on. Incidentally, the Cray-1 vector architecture also had a similar `vl` register. The ability to operate on a top portion of elements eliminates the need for strip-mining epilogues, as Listing 1 illustrates.

IV. RISC-V Vector Math Library

We wrote our library in the C programming language. The RISC-V vector instructions can be accessed in several ways. At the lowest level, one can use inline assembly; at the highest level, one can write standard scalar C code and rely on vectorizing compilers. Clang (and GCC will soon) provides access to the RISC-V Vector ISA through specific and also overloaded intrinsic functions. We chose to use the latter as they offer both direct access to the ISA as well as ease of re-configuration. For example, the overloaded intrinsic

```
vsum = __riscv_vfadd(vx, vy, v_ell);
```

works for either IEEE binary16, binary32 or binary64 and for any `LMUL` values. By using types defined in one header file, we can easily use different `LMUL` settings for different functions. All of these intrinsics start with the prefix `__riscv_` and requires the active vector length parameter (the `v_ell` after `vx` and `vy`). For the rest of the paper, we will omit both so to simplify our presentation.

```
#include "rvvlm.h"
#include FUNC_PRECISION_N_LMUL
#define FUNC_PARAM_1 func_value_1
#define FUNC_PARAM_2 func_value_2
    ......................
#include function.inc.h
```

Listing 2. Organization of a function implementation

In this style, the most used intrinsics are the usual suspects: `vfclass`, `vfadd`, `vfsub`, `vfmadd`, `vfmacc`, `vfsgnj`. The `vfclass` instruction identities a floating-point as one of ten classes and is used extensively to enable exception handling detailed in Section IV-B. The multiply-add `vfmadd` and multiply-accumulate `vfmacc` instructions have variants that either negate the product, summand or the resulting expression. The sign injection `vfsgnj` instruction has variants of injecting a negated sign or `xor`-ing the operands' sign bits. For some functions, the instructions `vfdiv`, `vfsqrt`, `vfrec7` are needed. The approximate reciprocal instruction `vfrec7` provides a value that approximates $1/\texttt{x}$ to about 7 significant bits and proves to be valuable (see details later). Finally, the logical manipulations of mask (predicates) are also frequently used. They include `vmand`, `vmor`, `vmandn`, `vcpop` which are `a and b`, `a or b`, `a and` (`negate b`), and counting the number of `trues` in a mask register. respectively. Finally, we note that whenever one of the operands to a binary operator happens to be a scalar and not a vector, it has to be the second one. Thus for non-commutative operators such as subtraction and division, there are the "reverse" forms `vfrsub`, `vfrdiv`.

Listing 2 shows the typical organization of a function. The top-level include file `rvvlm.h` defines all the libm functions. It also defines the choice of precision and `LMUL` value via defining `FUNC_PRECISION_N_LMUL` to be the name of a specific header file. That header file defines the appropriate `typedef` for the function in question. At the bottom level, the header file `function.inc.h` (e.g. `expD.inc.h`) implements the function in question, but often with some conditional compilation as it is common that a number of functions are implemented very similarly except for some simple changes. Examples include `exp` and `exp2`, or `sin` and `sinpi`. The conditional compilation flags are set by those `FUNC_PARAM_n` prior to the lowest level include file. These function specific parameters also control which of the two APIs will be used for the function: the unit stride API

```
void func(size_t n, const double *x, double *y);
```

or the general stride API

```
void func(size_t n, const double *x, size_t inc_x,
    double *y, size_t inc_y);
```

Listing 3 shows the common structure of the implemented functions. Steps 1 and 5 are straightforward load/store between registers and memory. The strip mining is the same as described earlier. We elaborate on the other steps.

```
ENSURE_RNE;
// macro; ensure round-to-nearest-even in effect
for (; n>0; n-=v_ell) { // strip mining
  //1. Load input into vector register
  //2. Filter and handle exceptions
  //3. Compute for normal inputs
  //4. Merge in exceptional results
  //5. Store vector register to output
}
RESTORE_IF_NEEDED;
// macro; ensure original rounding mode in effect
```

Listing 3. Structure of a function implementation

A. Prevalent Rounding Mode

The rounding mode setting during function computation is important. Many existing function libraries assume the floating-point environment has the round-to-nearest-even mode set. While this assumption is very likely valid, it is not ensured. Unfortunately, for most if not all elementary function implementations, computing in a different rounding mode can result in losing almost all precision, not just a couple of bits. This is because most implementations entail an argument reduction process that typically look like

$$r = x - \texttt{nearest\_int}(x/p) \times p$$

and expects the result r to fall into the interval $[-p/2, p/2]$. For example, $p = \log(2)$ for the exponential function, and $p = \pi$ (or $\pi/2$) for the trigonometric functions. An approximating polynomial that specializes on this interval is hard coded into the implementation. The "`nearest_int`" is typically a round-to-integer instruction whose rounding mode is controlled by the environment. All rounding modes except round-to-nearest can lead to an r as large as p in magnitude, rendering the approximating polynomial's accuracy woefully inadequate. We do not take a round-to-nearest mode for granted. Changing the rounding mode setting in a floating-point environment may lead to draining of the processing pipeline; but the cost of reading the current setting is likely to be moderate. Our macro `ENSURE_RNE` reads the current rounding mode and only change it if necessary. The macro to restore the rounding mode does the reverse – and in general involves neither reading nor writing the rounding mode setting. The usual cost involves reading the current rounding mode and two non-taken actions, which is amortized over the length of the input array.

B. Exceptions Handling

Consider Step 2 in Listing 3. Elementary functions that are defined as mappings from floating-point numbers into floating-point numbers need to handle many special situations that the classical mathematical functions do not. Similar to fdlibm, we handle exceptions faithfully. Moreover, in the context of vector math libraries, we strive to minimize branching and masking.

1) Branch and Masked Operations Avoidance: Generally one expects that none of the elements within a vector register triggers an exception. The approach taken here is to quickly recognize this situation and bypass exception handling immediately. The `vfclass` instruction which returns a 10-bit one-hot-encoding of 10 classes for floating-point numbers greatly facilitates this approach. Exceptional arguments often fall exactly into several of these classes. Take the logarithm function for example: the exceptional arguments are `NaN` (signaling and quiet), $\pm$`Inf`, ± 0, and strictly negative finite values. Moreover, a positive subnormal number is best normalized first for many common algorithms for this function. In one-hot-encoding, these arguments correspond to `0X3BF`. Hence an argument is exceptional if $\texttt{and}(\texttt{class}, \texttt{0x3BF}) > 0$ where `class` is the result of applying the `vfclass` to a vector register of input arguments. It is also common to have special handling of small-magnitude arguments so as to avoid generating spurious underflow signals. For example, computing $x \times \texttt{one\_ov\_ln2}$ is common in implementing $\exp(x)$; but this multiplication can trigger an unwarranted underflow signal if $|x|$ is very small.

The previous discussions show that a mask is naturally generated pinpointing exceptional input arguments. Using the population count instruction `vcpop` on that mask shows how many of the arguments within a vector register is exceptional. In the rare situation that these are present, we first generate the required results and their associated exception signals (see below). At the very end of the function computation, it suffices to merge – Step 4 of Listing 3 – these exceptional results with the normal one computed from non-exceptional inputs.

We also use the uniform approach that replaces these exceptional arguments with some "safe" values so that normal computation can be applied to the entire input vector register without triggering unwarranted numerical exceptions. The values 0 and 1 are common choices. Avoiding masked computation is of value as only one register `v0` is used for this purpose, and is hence a precious resource.

2) Exception Generation: For exceptional inputs we produce function-specific appropriate result values and exception signals. In general this can be effected by some simple arithmetic instructions. The addition $\texttt{vx} + \texttt{vx}$ for example on input `NaN` produces the canonical `NaN` result but also generates the invalid operation signal if a signaling `NaN` is present. The approximate reciprocal instruction $\texttt{vfrec}(\texttt{vx})$ generates a divide-by-zero exception on a ± 0 input. Hence for the $\log(x)$ function for example, the operation $\texttt{vfrec7}(\texttt{vx}) + \texttt{vx}$ produces the desired result **and** exception signals whenever input arguments are $\pm 0, \texttt{NaN}, +\texttt{Inf}$.

C. Accurate Function Computations

Algorithms and implementations for elementary functions are generally well understood [15] and many take the 3-step approach of reduction, approximation and reconstruction, perfectly illustrated by computing $\exp(x)$.

$$\begin{aligned} r &\approx x - n\log(2); && \text{reduction} \\ p &\approx \exp(r); && \text{approximation} \\ e^x &\approx 2^n p && \text{reconstruction} \end{aligned}$$

Nevertheless, it is non-trivial to implement the above so that the delivered result is within 1 ulp in double precision to the

Operations	# ops, an op is $+, -, \times$ or `fmadd`
$d(dd) + d(dd) \to dd$	6, 7, 8
$d(dd) \times d(dd) \to dd$	2, 3, 4
$d(dd)/d(dd) \to dd$	3, 4, 5 plus 1 div
$\sqrt{d(dd)} \to dd$	3, 4 plus 1 sqrt and 1 div

TABLE I
COST IN ORDER OF 0, 1, OR 2 INPUTS ARE DD

exact e^x. In general some steps must use precision higher than double precision. For example one must make use of a $\log(2)$ value to at least 11 more bits. For most hardware, double is the widest supported precision. Consequently, variables that require extra precision can only be represented in multiple double-precision variables such as a pair – commonly referred to as a double-double. One can see double-double arithmetic operations in double-precision function implementations at where precision is more critical, which is typically near the end of the entire computation.

A double-double (dd) variable (X, x) is a "head-tail" pair of double (d) where $|x| \ll |X|$. Arithmetic involving dd variables is well understood (see [9] e.g.) and its cost is high as summarized in Table I.

However, a doubling of precision is not needed but a mere handful of extra bits of precision, like 6 or 7, suffices to help deliver the final result to within 1 ulp of error. We therefore mostly use specialized extra-precision computation in place of general dd arithmetic. Some of these computations rely on the vector ISA while some other exploit the special form of the function in question. Here are some commonly used algorithms.

1) Extra-precision Sum: The need for an extra-precise sum $A + B$ of two floating-point variables arise very often. A number of double-double techniques are well known (see [16] for example). The general algorithm that is often called Knuth-2-sum requires 6 operations. In the case when $|A| \geq |B|$, the so-called fast2sum algorithm requires only 3 operations:

```
S = vfadd(A,B); s = vfadd(vfsub(A,S),B);
```

We also make use of a situation when both summands are non-negative, which save 1 operation from a general two-sum:

```
S = vfadd(A,B); X = vfmax(A,B); Y = vfmin(A,B);
s = vfadd(vfsub(S,X),Y);
```

2) Extra-precision FMA: The fused multiply-add instruction deliver $A + BC$ with 1 rounding for floating-point quantities A, B and C. In a number of situations, we need to obtain this expression to extra precision. For libm functions, it is not uncommon that $|A| \geq |BC|$. If this is the case, the 3-instruction sequence suffices:

```
S = vfmadd(B,C,A); s = vfmadd(B,C,vfsub(A,S));
```

The mathematical sum $S+s$ approximates $A+BC$ to double-double precision because $1/2 \leq S/A \leq 2$, making the subtraction $A-S$ exact [17]. Hence S is the correctly rounded value of $A + BC$ and s is the correctly rounded value of the exact trailing value $A + BC - S$. In $\texttt{expm1}(x)$, $\texttt{log}(x)$ and $\texttt{cos}(x)$ the leading terms of the core approximation are $x + x^2/2$, $x - x^2/2$ and $1 - x^2/2$, which all satisfy the condition here as $|x| < 1$ (after argument reduction) and that $x^2/2 = x \times (x/2)$ where the latter term is computed exactly.

3) Extra-precision Square Root: Extra-precision square root can be computed based on the observation that if $R = \sqrt{X}(1 - \delta)$ where $|\delta|$ bounded by machine epsilon, then $R\delta$ would be a good compensatory term; and $(X-R^2)/(2X) \approx \delta$, hence the cost tabulated in Table I. Using the approximate reciprocal instruction `vfrec7` for $1/X$ essentially provide a 7-bit approximation of δ, suffices for an extra-precision square root for the purpose of accurate function computation. The code sequence for extra-precise square root of a dd variable (X, x) is as follows.

```
R = vfsqrt(X);
delta = vfmul(vfadd(vfmadd(-R,R,X),x),vfrec7(X))
r = vfmul(vfmul(delta,R),0.5)
```

It substitutes the cost of the division in Table I with `vfrec7` which in all likelihood will be a fast instruction.

4) Fixed-Point Computations: Consider the common step in Horner's recurrence for polynomial evaluation: $r \times p + c_j$ where p is the partially computed polynomial. Because the vector fused-multiply-add cannot take a scalar input for the addend, we need two vector instructions to realize $r \times p + c_j$ in floating point. One needs to "splat" the scalar constant c_j into a vector register, followed by the fused multiply-add instruction. Nevertheless, to deliver the final result of a function computation to within 1 ulp of error, the last few operations within a function computation typically requires higher-than-native precision.

Consider the use of "double-double" in the tail end of Horner's recurrence $r \times p + c_j$. In this case, both p and r are in double-double while the coefficients c_j is a simple double-precision value. According to Table I, the multiplication needs 4 instructions. If we assume the coefficient c_j is dominant, we can save 3 instructions from the double-double addition (c.f. previous discussion on fast2sum), resulting in 4 instructions. Thus the recurrence step requires 8 instruction in general. One usually computes the beginning part of the Horner's recurrence in plain double precision before switching to double-double. At the first step of the transition, p is a simple double-precision variable, whose product with a double-double takes only 3 instructions, not 4. Therefore if the double-double portion involve k coefficients, it will take $8k - 1$ instructions. We can possibly do much better using the fixed-point support in RISC-V vector ISA.

RISC-V vector ISA supports fixed-point computations, which is a distinctive feature. Fixed-point instructions interpret 64-bit inputs to be a 2's complement integral value multiplied by the factor 2^{-63}. Hence all input and output values are of the form $2^{-63}I$ where I is an integer $-2^{63} \leq I < 2^{63}$. The saturation multiplication `vsmul` operates on $2^{-63}I_x$ and $2^{-63}I_y$. It essentially multiplies the two integers I_x and I_y followed by right shifting 63 bits and round according to a rounding setting. Out-of-bound results are saturated. Corresponding saturated addition/subtraction instructions are also defined in the ISA; a fused multiply-add instruction however is not available. Thus fixed-point arithmetic can carry as much as 63 bits of precision when quantities are scaled optimally. In practice, if we have two real quantities a and b such that their round-to-integer value $A = 2^{63}a$ and $B = 2^{63}b$ fall within the range of 64-bit 2's complement integer, then the result of a saturated multiplication of A and B corresponds to the product $2^{63}a \times b$ in the absence of saturation (overflow).

Consider the Horner's recurrence $r \times p + c_j$. If R, P and C_j are the fixed-point analogues for the three variables, they can be more than 60-bit precise when scaled well. Moreover, if P and C_j are of the same scale and R's scale factor is 2^{63}, then it takes also only two instructions for this common step. Even when R's scale factor is not 2^{63}, three instructions suffice. Specifically, supposed the last k terms of a Horner's recurrence needs to be done with extra precision. Converting a floating-point value to fixed points takes two instruction in general: multiplication with a scale factor 2^q followed by floating point to integer conversion. The other direction also takes 2 instruction, applied reversely. The conversion overhead in completing a Horner's recurrence in fixed-point is thus 6 instructions: converting p and r to fixed point and the final result back to floating point. Hence if R carries a scale factor of 2^{63}, the cost of completing a Horner's recurrence in fixed-point with k coefficients is $2k+6$; otherwise it will be $3k+6$. For instance, if $k=4$, instructions needed for extra-precision evaluation in double-double is $8k-1=31$ while the two scenarios of fixed-point arithmetic are $2k+6 = 14$ and $3k+6=18$.

Compared to a purely floating-point double-double approach, using a mixture of floating-point and fixed-point arithmetic deliver equally precise result in fewer instructions, even taking into account the needed floating-fixed point conversions. Indeed, the result can be even more precise simply because we can afford to do more extra-precise computations.

V. Illustrative Examples

We present here examples in our library that utilize some of the techniques in the previous section.

A. Common Structure of Exception Handling

The power function $\texttt{pow}(x, y) = x^y$ has a large number of exceptional cases [18]. Luckily, the condition that one of x or y belongs to this group of 6 special values $\pm$0, $\pm$INF, sNaN and qNaN include most of the exceptional cases. When neither x nor y is one of these 6 values, we can compute $|x|^y$ numerically followed by a simple check for $x<0$ and handle that appropriately.

As Section IV-B suggests, we use the `vfclass` instruction and mask the result with the "stencil" `0x399`. Listing 4 is the illustrative snippet. It is reasonable to assume that exceptional cases are rare in practice and thus the performance inside the handling is not critical. We continue using the `vfclass` and various vector instructions inside the exception handling, considering each of the three mutually exclusive cases of both x and y being special, or only one is.

```
x_class = vand(vfclass(vx), 0x399);
y_class = vand(vfclass(vy), 0x399);
special = vmor(vmsgt(x_class,0), vmsgt(y_class,0));
if (vcpop(special) > 0) {
    // performance non-critical
    // create vz_special as the result and generate
    appropriate signals
    // substitue exceptional argument with safe
    values
    vx = vmerge(vx, 1.0, special); vy = vmerge(vy,
    0.0, special);
}
// All values in vx and vy are now safe to compute
// compute results vz for these safe values
vz = vmerge(vz, vz_special, special)
```

Listing 4. Exception Handling

B. Extra-precision Simulation in Floating Point

We illustrate some instances of situation-specific extra precision simulation in floating-point arithmetic. The hyperbolic functions $\cosh(x)$ and $\sinh(x)$ are defined as $(e^x \pm e^{-x})/2$. Straightforward use of these formula requires essentially obtaining an extra precise e^x delivered in double-double format. This is followed by a double-double reciprocal to get e^{-x}, and a subsequent double-double addition/subtraction. A fair number of double-double basic operations are involved in all, and that all the steps are essentially sequentially dependent. We use an alternative approach that reuses common expressions between the computation of e^x and e^{-x} and specialized extra-precise simulation. An expensive division and a few data dependencies are eliminated.

Since $\cosh(-x) = \cosh(x)$ and $\sinh(-x) = -\sinh(x)$, it suffices to focus on $x \geq 0$. A usual approach in computing e^x first reduces the argument x to $r \in [-\log 2/2, \log 2/2]$ via $x = n \log 2 + r$ where n is an integer. Thus $\cosh(x)$ and $\sinh(x)$ are given by $2^{n-1}(e^r \pm s e^{-r})$ where $s = 2^{-2n}$. Using the Remez algorithm [19], we pick an approximating polynomial to e^r with the leading terms $1 + r + r^2/2$:

$$e^r \approx 1 + r + r^2/2 + r^3(p(r^2) + rq(r^2)).$$

Here $p(r^2)$ and $rq(r^2)$ are the even and odd parts on the polynomial starting from the third degree. Hence

$$e^r \approx 1 + \left(r + \frac{r^2}{2}\right) + r^3(p(r^2) + rq(r^2)),$$
$$e^{-r} \approx 1 - \left(r - \frac{r^2}{2}\right) - r^3(p(r^2) - rq(r^2)).$$

The value before the final scaling of 2^{n-1} is

$$[(1 \pm s)] + \left[\left(r + \frac{r^2}{2}\right) \mp s\left(r - \frac{r^2}{2}\right)\right] + U,$$

where

$$U = r^3\left[(p(r^2) + rq(r^2)) \pm s(p(r^2) - rq(r^2))\right].$$

Because $|r| \geq |r^2/2|$ as $|r| < 1$, we apply the 3-instruction sequence in IV-C to get $r \pm r^2/2$ accurately as $(B_{\text{pos}}, b_{\text{pos}})$ and $(B_{\text{neg}}, b_{\text{neg}})$. We compute the expressions $B_{\text{pos}} \pm sB_{\text{neg}}$

```
#define FAST2SUM(X,Y,Z,z) \
Z = vfadd(X,Y); z = vfadd(vfsub(X,Z),Y);
#define FAST2FMA(A,B,C,Z,z) \
Z = vfmadd(A,B,C); z = vfmadd(A,B,vfsub(C,Z));
   ..... some code, obtained r, s, U,     .....
r_prime = vfmul(r, 0.5);
FAST2SUM(1.0, -s, A, a); // use s for cosh
FAST2FMA(r, r_prime, r, B_pos, b_pos);
FAST2FMA(r, -r_prime, r, B_neg, b_neg);
FAST2FMA(B_neg, -s, B_pos, B, b); // use s for cosh
// P = U + (a + (b + (b_pos - s b_neg)))
Z = vfadd(A, vfadd(B, P));
// return 2^(n-1) * Z
```

Listing 5. `cosh` and `sinh`

extra precisely also with the 3-instruction sequence because $|B_{\text{pos}}| \geq s|B_{\text{neg}}|$. This is clear if $s = 2^{-2n} \leq 1/2$ (note that $n \geq 0$). If $s = 1$, this means that $n = 0$ and hence $r = x \geq 0$, implying $|B_{\text{pos}}| \geq |B_{\text{neg}}|$. Listing 5 illustrates this.

C. Fixed-Point Computations

The inverse tangent function lends itself to computation almost entirely in fixed-point arithmetic. Since $\arctan(-x) = -\arctan(x)$, a standard algorithm works with $|x|$ and restores the sign near the end of the task. As

$$\arctan(x) = \pi/2 - \arctan(1/x) \quad \text{for } x > 1,$$

the main task is to compute the function $\arctan(t)$ on $[0, 1]$ which is typically approximated by an odd polynomial:

$$\arctan(t) \approx t + ts\sum_{j=0}^{L} p_j s^j, \quad s = t^2, \quad 0 \leq t \leq 1.$$

The variable t is either x or $1/x$. The coefficients p_j are approximately $-1/3, 1/5, -1/7, \ldots$, which decay rather slowly. Together with the fact that t can be as large as 1, one needs $L = 19$ for a double-precision `atan` function. Furthermore, when computed in floating-point arithmetic, extra-precise simulation is needed for the Horner's recurrence that iterates from $j = 19$ downwards as soon as j reaches 3.

Alternatively, we note that as long as $sp(s)$ is computed with a small absolute error Δ, this error is propagated as $t\Delta$ in $t + t(sp(s) + \Delta)$. Since the final result's magnitude is never smaller than $t - t^3/3$ (when $x \leq 1$) or $\pi/4$ (when $x > 1$), an absolute error of $t\Delta$ is never bigger than $|\Delta|$ in relative error. This makes computing $sp(s)$ in fixed-point arithmetic with a scale factor of 2^{63} ideally suited for the task of implementing the `atan` function. As noted previously, scaling both the coefficients and the variable with 2^{63} results in an efficient two-instruction per Horner's recurrence step computation. Note however that t can be as large as 1 and thus a scale of 2^{63} can result in overflow for both s and s^2. The term s^2 is needed if we want to introduce some parallelism by computing say the odd and even part of the polynomial $\sum_{j=0}^{19} p_j s^j$ in parallel. We exploit the fact that $-s$ and $-s^2$ can both be scaled by 2^{63} without overflow and thus apply the "negated" Horner's recurrence step of $P = -(X \times P) + P_j$. Listing 6 shows the scaling and the negated Horner's recurrence.

```
  ... exception filtering, create t to be x or 1/x
// (t, dt) is the double-double of 2^62 x or 2^62 /
    x
T_62 = vadd(vfcvt_x(t), vfcvt_x(dt));
// (t,dt) in fixed-point, scale 2^62
nT_63 = vsll(vfrsub(T_62,0),1);
// -(t,dt) in fixed-point, scale 2^63
nS_62 = vsmul(T_62,nT_63);
// -(t,dt)^2 in fixed-point, scale 2^62
nS_63 = vsll(nS_62, 1);
// -(t,dt)^2 in scale 2^63
// we can now use the negated Horner's recurrence to
// compute P0 + S*(P1 + S*(P2 + ... + S*P19))
Poly = vrsub(vsmul(nS_63,P19),P18);
Poly = vrsub(vsmul(nS_63,Poly),P17);
  ..... compute recurrence until P0 ...
```

Listing 6. `atan`

```
#define DD(X,x_head,x_tail) x_head = vfcvt_f(X); \
x_tail = vfcvt_f(vsub(X,vfcvt_x(x_head)))
#define DIV2_7(A,a,B,b,Q,q) Q = vfdiv(A,B); \
resid = fadd(fmadd(-Q,b,fmadd(-Q,B,A)),a); \
q = vfmul(resid,vfrec7(B))
  ... beginning computation: exception handling, etc
// ... at this point all elements in vx is in [0, 1)
one_p_x = vfadd(vx,1.0); one_m_x = vfrsub(vx,1.0);
ratio = vfmul(one_p_x,vfrec7(one_m_x));
n = vsub(vsrl(vadd(vsrl(FasU(vx),44),0x96),8),1023);
X = fcvt_x(vfmul(vx,0x1.0p60));
A = vadd(X,One); B = vsll(vrsub(X,One),n);
Numer = vsub(A,B); Denom = vadd(A,B);
DD(Numer, E, e); DD(Denom, F, f);
DIV2_7(E,e,F,f,r_hi,r_lo); r = vfadd(r_hi,r_lo);
// ...compute n*log(2)+r_hi+r_lo+r^3*poly(r^2)
// ...with floating-point technique to obtain
// ...n*log(2)+r_hi as double-double
```

Listing 7. `atanh`

D. Mixed Floating-Point Fixed-Point Computation

Section IV-C4 presented a common scenario where fixed-point arithmetic can be used in the ending stage of a computation in lieu of double-double simulation. The inverse hyperbolic tangent function $\text{arctanh}(x)$ is an example where the beginning, rather than the ending, stage needs extra precision that can be provided by fixed-point arithmetic.

The arctanh function can be defined as:

$$\text{arctanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right).$$

Common implementations essentially compute $1 + x$, $1 - x$ and $(1+x)/(1-x)$ in double-double arithmetic followed by a special logarithm function implementation that accepts double-double input. Inside this logarithm implementation, further double-double operations are needed: Typically, an input y to the logarithm function is scaled by a factor of 2^{-n} so that $2^{-n}y = r \in [1/\sqrt{2}, \sqrt{2}]$. At this point,

$$\log(y) = 2\,\text{arctanh}\left(\frac{y-1}{y+1}\right)$$

is approximated by an odd polynomial in $r = (y-1)/(y+1)$ which in turn requires double-double computation of $y-1$, $y+1$ and $(y-1)/(y+1)$.

We simplify the implementation for $\operatorname{arctanh}$ as follows. First, we approximate $(1+x)/(1-x)$ to about 7 bits using the `vfrec7` instruction. This value allows us to obtain a scale factor $s = 2^{-n}$ so that $s(1+x)/(1-x)$ lies in $[\alpha/2, \alpha]$, $\alpha = 1.4252$. Thus

$$\operatorname{arctanh}(x) = (n/2)\log(2) + (1/2)\log(s(1+x)/(1-x)).$$

Now,

$$\frac{1}{2}\log\left(s\frac{1+x}{1-x}\right) = \operatorname{arctanh}\left(\frac{(1+x)-(1-x)/s}{(1+x)+(1-x)/s}\right).$$

The numerator and denominator are easily computed in fixed point with a scale factor of 2^{60} without ever overflowing. After both fixed-point values are computed, we convert them back to double-double format and compute one double-double division. Note that the numerator and denominator in fixed point are error free as long as $x \geq 2^{-8}$. If $x < 2^{-8}$, the absolute error in the fixed-point numerator constitutes too large a error relative to x; but in this case we use x itself rather than the quotient as the argument to that core $\operatorname{arctanh}$ approximating polynomial. Listing 7 is the illustration.

1) Accuracy Measurements: We built our C library using LLVM clang and ran tests with the emulator QEMU, which also provides a long-double (112 mantissa bits) libm against which our accuracy measurement is performed. Each function `F` is tested on tens of millions of arguments on meaningfully selected regions, using the long-double version f as reference. For example, the `exp` function is tested at where it nearly over or underflows; the `expm1` and `log1p` functions are tested with tiny arguments; the `pow` function is tested where x^y almost over or underflow, including when $log(x)$ is close to zero. Table II tabulates the maximum observed error in ulp. For functions with multiple versions, we report here on the one that we deemed to be a reasonable choice; but all versions have accuracy to be within 1 ulp of the correct answer. For the exponential functions, it is well understood that when the result underflows, the error could be as large as 0.5 ulp plus half of the error without underflow (see [20] for example). The main reason is that the scaling operation of $2^n \times y$ for a floating-point value y is no longer error free. We deem the cost to avoid this extra error too high for the marginal benefits expected.

VI. Final Thoughts

We developed an open-source double-precision vector math library specifically for RISC-V, which is a first to the best of our knowledge. In reflection, there are some "we really wish we had" instructions or features: some of those low-level IEEE recommended functions such as `logb` or `scalb`; the addend of FMA can be a scalar; fused-multiply-add and sign manipulation for fixed point; static rounding mode instructions for integer/floating-point conversions and fixed-point arithmetic. Nevertheless, we found overall that the RISC-V vector ISA is well designed and greatly facilitated our work.

Library Functions				Maximum Deviation in ulps			
`exp`	`exp2`	`exp10`	`expm1`	0.56	0.56	0.75	0.77
when result underflows				0.77	0.77	0.82	N/A
`log`	`log2`	`log10`	`log1p`	0.55	0.57	0.56	0.66
`pow`	`cbrt`			0.55	0.52		
`sin`	`sinpi`	`cos`	`cospi`	0.79	0.76	0.76	0.77
`tan`	`tanpi`			0.62	0.61		
`sinh`	`cosh`	`tanh`		0.67	0.59	0.76	
`asin`	`asinpi`	`acos`	`acospi`	0.66	0.71	0.64	0.65
`atan`	`atanpi`	`atan2`	`atan2pi`	0.55	0.55	0.55	0.55
atan2pi underflows				0.75			
`asinh`	`acosh`	`atanh`		0.55	0.56	0.54	

TABLE II
Error: $|(\mathtt{F}(x) - f(x))/\mathrm{ulp}(f(x))|$, f is the long-double function.

The author thanks Andrew de los Reyes, Keeran Rothenfusser, Brendan Sweeney, Michael Wan and Liang-Kai Wang for support and valuable comments.

References

[1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume 1: Base user-level ISA," University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011.

[2] K. Asanovic and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014.

[3] The RISC-V International, 2023. [Online]. Available: https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions

[4] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[5] *FDLIBM: Freely Distributed LIBM*, Sun Microsystems Inc., 2010. [Online]. Available: https://www.netlib.org/fdlibm/

[6] J.-M. Muller, N. Brunie, F. de Dinechine, C.-P. Jeannerod, M. Joldes, V. Lefevre, G. Melquiond, N. Revol, and S. Torres, *Handbook of floating-point arithmetic*. BIRKHAUSER, 2019.

[7] *GNU Project*, The Free Software Foundation, 2018. [Online]. Available: https://www.gnu.org/software/libc/

[8] ROCm, "ROCM/ROCM-device-libs: ROCM device libraries." [Online]. Available: https://github.com/ROCm/ROCm-Device-Libs

[9] M. M. Joldes, J.-M. Muller, and V. Popescu, "Tight and rigorous error bounds for basic building blocks of double-word arithmetic," *ACM Transactions on Mathematical Software*, vol. 44, pp. 1–27, 2017. [Online]. Available: https://hal.science/hal-01351529v3/document

[10] "Intel MKL," Jan 2024. [Online]. Available: https://en.wikipedia.org/wiki/Math_Kernel_Library

[11] [Online]. Available: https://github.com/ARM-software/optimized-routines/tree/master/math/aarch64

[12] C. Lauter, "A new open-source SIMD vector libm fully implemented with high-level scalar C," in *Asilomar Conference on Signals, Systems and Computers*, 11 2016, pp. 407–411.

[13] N. Shibata and F. Petrogalli, "Sleef: A portable vectorized library of C standard mathematical functions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, p. 1316–1327, 2020.

[14] E. Bavier, N. Knight, H. de Lassus Saint-Genies, and E. Love, "Vectorized nonlinear functions with the RISC-V vector extension," in *Proceedings of IEEE Symposium on Computer Arithmetic*, 2023.

[15] J.-M. Muller, *Elementary functions: Algorithms and implementation*. Birkhauser, 1997.

[16] P. Kornerup, V. Lefévre, N. Louvet, and J.-M. Muller, "On the computationof correctly-rounded sums," INRIA, Tech. Rep. RR-7262, 2010.

[17] P. H. Sterbenz, *Floating-point computation*. Prentice Hall, 1974.

[18] [Online]. Available: https://en.cppreference.com/w/cpp/numeric/math/pow

[19] W. Fraser, "A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable," *Journal of the ACM*, vol. 12, no. 3, pp. 295–314, 1965.

[20] P. T. P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, no. 2, p. 144–157, 1989.