A Domain-Specific Architecture for Elementary Function Evaluation

Anuroop Sharma
Christopher Kumar Anand anandc@mcmaster.ca
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1
© 12 September 2014

We propose a Domain-Specific Architecture for elementary function computation to improve throughput while reducing power consumption as a model for more general applications: support fine-grained parallelism by eliminating branches, eliminate the duplication required by co-processors by decomposing computation into instructions which fit existing pipelined execution models and standard register files. Our example instruction architecture (ISA) extension supports scalar and vector/SIMD implementations of table-based methods of calculating all common special functions, with the aim of improving throughput by (1) eliminating the need for tables in memory, (2) eliminating all branches for special cases, (3) reducing the total number of instructions. Two new instructions are required, a table-lookup instruction and an extended-precision floating-point multiply-add instruction with special treatment for exceptional inputs.

To estimate the performance impact of these instructions, we implemented them in a modified Cell/B.E. SPU simulator and observed an average throughput improvement of 2.5 times for optimized loops mapping single functions over long vectors.

Computing and Software Report CAS-14-06-CA

The authors thank NSERC, MITACS, Optimal Computational Algorithms, Inc., and IBM Canada for financial support.

Elementary function libraries are often called from performance-critical code sections, and hence contribute greatly to the efficiency of numerical applications, and the performance of these and libraries for linear algebra largely determine the performance of important applications. Current hardware trends impact this performance as

- longer pipelines and wider superscalar dispatch favour implementations which distribute computation across different execution units and present the compiler with more opportunities for parallel execution, but make branches more expensive;
- Single-Instruction-Multiple-Data (SIMD) parallelism makes handling cases via branches very expensive;
- memory throughput and latency which is not advancing as fast as computational throughput hinders the use of lookup tables;
- power constraints limit performance more than area.

The last point is interesting, and is gives rise to the notion of "dark silicon" in which circuits are designed to be un- or under-used to save power. The consequences of these thermal limitations vs silicon usage have been analyzed [6], and a number of performance-stretching approaches have been proposed [16] including the integration of specialized co-processors.

Our proposal is less radical: instead of adding specialized co-processors, add novel fully pipelined instructions to existing CPUs and GPUs, use the existing register file, reuse existing silicon for expensive operations, *e.g.*, fused multiply-add operations, eliminate costly branches, but add embedded look-up tables which are a very effective use of dark silicon. In the present paper, we demonstrate this approach for elementary function evaluation, *i.e.,* `libm` functions and especially vector versions of them.

To optimize performance, our approach takes the successful accurate table approach of Gal *et al* [8, 9] coupled with algorithmic optimizations [14, 15] and branch and table unifications [2], to reduce all fixed-power-, logarithm- and exponential-family functions to a hardware-based lookup followed by a handful of floating-point operations, mostly fused multiply-add instructions evaluating a single polynomial. Other functions like `pow` require multiple such basic stages, but no functions require branches to handle exceptional cases, including subnormal and infinite values.

Although fixed powers (including square roots and reciprocals) of most finite inputs can be efficiently computed using Newton-Raphson iteration following a software or hardware estimate [7], such iterations necessarily

introduce NaN intermediate values, which can only be corrected using additional instructions (branches, predication, or selection). Therefore, our proposed implementations avoid iterative methods.

For evaluation of the approach, the proposed instructions were implemented in a Cell/B.E. [4] SPU simulator, and algorithms for a standard math function library were developed that leverage these proposed additions. Overall, we found the new instructions would result in an average 2.5 times throughput improvement on this architecture, versus current published performance results (Mathematical Acceleration Subsystem, 5.0, IBM). Given the simple data dependency graphs involved, we expect similar improvements from implementing these instructions on all two-way SIMD architectures supporting fused multiply-add instructions. Higher-way SIMD architectures would likely benefit more.

In the following, the main approach is developed, and the construction of two representative functions, $\log x$ and $\log(x+1)$, are given in detail, providing insight by example into the nature of the implementation. In some sense these represent the hardest case, although the trigonometric functions require multiple tables, and there is some computation of the lookup "keys", the hardware instructions themselves are simpler. For a complete specification of the algorithms used, see [13].

1 New Instructions

Driven by hardware floating-point instructions, the advent of software pipelining and shortening of pipeline stages favoured iterative algorithms (see, *e.g.*, [12]); the long-running trend towards parallelism has engendered a search for shared execution units [5], and in a more general sense, a focus on throughput rather than low latency. In terms of algorithmic development for elementary functions, this makes combining short-latency seed or table value look ups with standard floating point operations attractive, exposing the whole computation to software pipelining by the scheduler.

In proposing Instruction Set Architecture (ISA) extensions, one must consider four constraints:

- the limit on the number of instructions imposed by the size of the machine word, and the desire for fast (i.e., simple) instruction decoding,
- the limit on arguments and results imposed by the architected number of ports on the register file,

function	$\frac{\text{cycles}}{\text{double}}$ new	$\frac{\text{cycles}}{\text{double}}$ SPU	Speedup (%)	max error (ulps)	table size(N)	poly order	$\log_2 M$
recip	3	11.3	**376**	0.500	2048	3	∞
div	3.5	14.9	**425**	1.333	recip	3	
sqrt	3	15.4	**513**	0.500	4096	3	18
rsqrt	3	14.6	**486**	0.503	4096	3	∞
cbrt	8.3	13.3	**160**	0.500	8192	3	18
rcbrt	10	16.1	**161**	0.501	rcbrt	3	∞
qdrt	7.5	27.6	**368**	0.500	8192	3	18
rqdrt	8.3	19.6	**229**	0.501	rqdrt	3	18
log2	2.5	14.6	**584**	0.500	4096	3	18
log21p	3.5	n/a	n/a	1.106	log2	3	
log	3.5	13.8	**394**	1.184	log2	3	
log1p	4.5	22.5	**500**	1.726	log2	3	
exp2	4.5	13.0	**288**	1.791	256	4	18
exp2m1	5.5	n/a	n/a	1.29	exp2	4	
exp	5.0	14.4	**288**	1.55	exp2	4	
expm1	5.5	19.5	**354**	1.80	exp2	4	
atan2	7.5	23.4	**311**	0.955	4096	2	18
atan	7.5	18.5	**246**	0.955	atan2	2+3	
asin	11	27.2	**247**	1.706	atan2	2+3+3	
acos	11	27.1	**246**	0.790	atan2	2+3+3	
sin	11	16.6	**150**	1.474	128	3+3	52
cos	10	15.3	**153**	1.025	sin	3+3	
tan	24.5	27.6	**113**	2.051	sin	3+3+3	

Table 1: Accuracy, throughput and table size (for SPU/double precision).

- the limit on total latency required to prevent an increase in maximum pipeline depth,
- the need to balance increased functionality with increased area and power usage.

As new lithography methods cause processor sizes to shrink, the relative cost of increasing core area for new instructions is reduced. The net cost may even be negative if the new instructions can reduce code and data size, thereby reducing pressure on the memory interface (which is more difficult

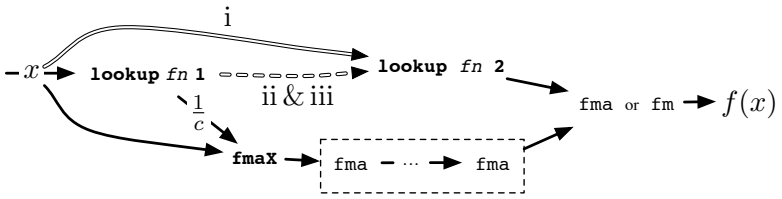

Figure 1: Data flow graph with instructions on vertices, for $\log x$, roots and reciprocals. Most functions follow the same basic pattern, or are a composition of such patterns.

to scale).

To achieve a performance benefit, ISA extensions should do one or more of the following

- reduce the number of machine instructions in compiled code,
- move computation away from bottleneck execution units or dispatch queues, or
- reduce register pressure.

Considering the above limitations and ideals, we propose to add two instructions, the motivation for which follows below:

d = `fmaX` a b c an extended range floating-point multiply-add, with the first argument having 12 exponent bits and 51 mantissa bits, and non-standard exception handling;

t1 = `lookup` a b f t an enhanced table look-up based on one or two arguments, and containing immediate argument specifying the function number and the sequence of the lookup, e.g. the first lookup used for range reduction or the second lookup used for function reconstruction.

It is easiest to see them used in an example. Figure 1 describes the data flow graph (omitting register constants), which is identical for a majority of the elementary functions. The correct lookup specified as an immediate argument to `lookup`, and the final operation being a `fma` for the log functions and `fm` otherwise. All of the floating point instructions also take constant arguments which are not shown. For example, the `fmaX` takes an argument which is -1.

The dotted box in Figure 1 represents a varying number of fused multiply-adds used to evaluate a polynomial after the multiplicative range reduction performed by the `fmaX`. The most common size for these polynomials is order three, so the result of the polynomial (the left branch) is available four floating point operations later (typically about 24-28 cycles) than the result $1/c$. The second `lookup` instruction performs a second lookup, for example, the $\log x$, it looks up $\log_2 c$, and substitutes exceptional results ($\pm\infty$, NaN) when necessary. The final `fma` or `fm` instruction combines the polynomial approximation on the reduced interval with the table value.

The gray lines indicate two possible data flows for three possible implementations:

i the second `lookup` instruction is a second lookup, using the same input;

ii the second `lookup` instruction retrieves a value saved by the first `lookup` (in final or intermediate form) from a FIFO queue;

iii the second `lookup` instruction retrieves a value saved in a slot according to an immediate tag which is also present in the corresponding first `lookup`.

In the first case, the dependency is direct. In the second two cases the dependency is indirect, via registers internal to the execution unit handling the look-ups.

All instruction variations have two register inputs and one or no outputs, so they will be compatible with existing in-flight instruction and register tracking. On lean in-order architectures, the variants with indirect dependencies — (ii) and (iii) — reduce register pressure and simplify modulo loop scheduling. This would be most effective in dedicated computational cores like the SPUs in which pre-emptive context switching is restricted.

The variant (iii) requires additional instruction decode logic, but may be preferred over (ii) because tags allow `lookup` instructions to execute in different orders, and for wide superscalar processors, the tags can be used by the unit assignment logic to ensure that matching `lookup` instructions are routed to the same units. On Very Long Instruction Word machines, the position of `lookup`s could replace or augment the tag.

In low-power environments, the known long minimum latency between the `lookup`s would enable hardware designers to use lower power but longer latency implementations of most of the second `lookup` instructions.

To facilitate scheduling, it is recommended that the FIFO or tag set be sized to the power of two greater than or equal to the latency of a floating-point operation. In this case, the number of registers required will be less

than twice the unrolling factor, which is much lower than what is possible for code generated without access to such instructions. The combination of small instruction counts and reduced register pressure eliminate the obstacles to in-lining these functions.

We recommend that `lookup`s be handled by either a load/store unit, or, for vector implementations with a complex integer unit, by that unit. This code will be limited by floating-point instruction dispatch, so moving computation out of this unit will increase performance.

1.0.1 Exceptional Values

A key advantage of the proposed new instructions is that the complications associated with exceptional values (0, ∞, NaN, and values which over- or under-flow at intermediate stages) are internal to the instructions, eliminating branches and predicated execution.

Iterative methods with table-based seed values cannot achieve this in most cases because

1. in 0 and $\pm\infty$ cases the iteration multiplies 0 by ∞ producing a NaN;
2. to prevent over/underflow for high and low input exponents, matched adjustments are required before *and* after polynomial evaluation or iterations.

By using the table-based instruction twice, once to look up the value used in range reduction and once to look up the value of the function corresponding to the reduction, and introducing an extended-range floating point representation with special handling for exceptions, we can handle both types of exceptions without extra instructions.

In the case of finite inputs, the value $2^{-e}/c$, such that

$$\frac{2^{-e}}{c} \cdot x - 1 \in [-2^{-N}..2^{-N}]$$

returned by the first `lookup` is a normal *extended-range* value. In the case of subnormal inputs, *extended-range* are required to represent this lookup value because normal IEEE value would saturate to ∞. Treatment of the large inputs which produce IEEE subnormals as their approximate reciprocals can be handled (similar to normal inputs) using the extended range representation. The extended range number is biased by +2047, and the top binary value (4095) is reserved for $\pm\infty$ and NaNs and 0 is reserved for

function	finite > 0	$+\infty$	$-\infty$	± 0	finite < 0
recip	$(\frac{2^{-e}}{c})_{\text{ext}}$, $(\frac{2^{-e}}{c})_{\text{sat}}$	0, 0	0, 0	0, $\pm\infty$	$(-\frac{2^{-e}}{c})_{\text{ext}}$, $(-\frac{2^{-e}}{c})_{\text{sat}}$
sqrt	$(\frac{2^{-e}}{c})_{\text{ext}}$, $\frac{2^{e/2}}{c}$	0, ∞	0, NaN	0, 0	0, NaN
rsqrt	$(\frac{2^{-e}}{c})_{\text{ext}}$, $\frac{2^{-e/2}}{c}$	0, 0	0, NaN	0, ∞	0, NaN
log2	$(\frac{2^{-e}}{c})_{\text{ext}}$, $e+\log_2 c$	0, ∞	0, NaN	0, $-\infty$	0, NaN
exp2	c, $2^I \cdot 2^c$	0, ∞	NaN, 0	0, 1	c, $2^{-I} \cdot 2^c$

$+_{\text{ext}}$	finite	$-\infty$	∞	NaN
finite	c	c	c	0
$-\infty$	c	c	0	0
∞	c	0	c	0
NaN	c	c	c	0

$*_{\text{ext}}$	finite	$-\infty$	∞	NaN
finite$\neq 0$	c	2	2	2
$-\infty$	$-\infty_f$	$-\infty_f$	$-\infty_f$	$-\infty_f$
∞	∞_f	∞_f	∞_f	∞_f
NaN_0	NaN_f	NaN_f	NaN_f	NaN_f
NaN_1	2_f	2_f	2_f	2_f
NaN_2	$1/\sqrt{2}_f$	$1/\sqrt{2}_f$	$1/\sqrt{2}_f$	$1/\sqrt{2}_f$
NaN_3	0_f	0_f	0_f	0_f

Table 2: *top:* Values returned by `lookup` instructions, for IEEE floating-point inputs $(-1)^s 2^e f$, which rounds to the nearest integer $I = \text{rnd}((-1)^s 2^e f)$. In case of exp2, inputs < -1074 are treated as $-\infty$ and inputs > 1024 are treated as ∞. For inputs < -1022, we create subnormal numbers for the second lookup. *bottom:* Treatment of exceptional values by `fmaX` follows from that of addition and multiplication. The first argument is given by the row and the second by the column. Conventional treatment is indicated by a "c", and unusual handling by specific constant values.

± 0 similar to IEEE floating point. When these values are supplied as the first argument of `fmaX`, they override the normal values, and `fmaX` simply returns the corresponding IEEE bit pattern.

The the second `lookup` instruction returns an IEEE double except when used for divide, in which case it also returns an extended range result.

In Table 2, we summarize how each case is handled, and describe it in detail in the following section. Each cell in Table 2 contains the value used in the reduction, followed by the corresponding function value. The first is given as an extended-range floating-point number which trades one bit of stored precision in the mantissa with a doubling of the exponent range. In all cases arising in this library, the extra bit would be one of several zero bits, so no actual loss of precision occurs. For the purposes

of elementary function evaluation, subnormal extended-range floating point numbers are not needed, so they do not need to be supported in the floating point execution unit. As a result, the modifications to support extended-range numbers as inputs are minor.

Take, for example the first cell in the table, recip computing $1/x$ for a normal positive input. Although the abstract values are both $2^{-e}/c$, the bit patterns for the two look ups are different, meaning that $1/c$ must be representable in both formats. In the next cell, however, for some subnormal inputs, $2^{-e}/c$ is representable in the extended range, but not in IEEE floating point, because the addition of subnormal numbers makes the exponent range asymmetrical. As a result, the second value may be saturated to ∞. The remaining cells in this row show that for $\pm\infty$ input, we return 0 from both lookups, but for ± 0 inputs the first `lookup` returns 0 and the second `lookup` returns $\pm\infty$. In the last column we see that for negative inputs, the returned values change the sign. This ensures that intermediate values are always positive, and allows the polynomial approximation to be optimized to give correctly rounded results on more boundary cases. Both `lookup`s return quiet NaN outputs for NaN inputs.

Contrast this with the handling of approximate reciprocal instructions. For the instructions to be useful as approximations 0 inputs should return ∞ approximations and vice versa, but if the approximation is improved using Newton-Raphson, then the multiplication of the input by the approximation produces a NaN which propagates to the final result.

The other cases are similar in treating 0 and ∞ inputs specially. Noteworthy variations are that $\log_2 x$ multiplicatively shifts subnormal inputs into the normal range so that the normal approximation can be used, and then additively shifts the result of the second `lookup` to componsate; and 2^x returns 0 and 1 for subnormal inputs, because the polynomial approximation produces the correct result for the whole subnormal range.

In Table 2, we list the handling of exceptional cases. All exceptional values detected in the first argument are converted to the IEEE equivalent and are returned as the output of the `fmaX`, as indicated by sub-script $_f$ (for final). The subscripted NaNs are special bit patterns required to produce the special outputs needed for exceptional cases. For example, when `fmaX` is executed with NaN_1 as the first argument (one of the multiplicands) and the other two arguments are finite IEEE values, the result is 2 (in IEEE floating point format).

$$\textbf{\textit{fmaX}}\ \mathrm{NaN}_1\ \mathrm{finite}_1\ \mathrm{finite}_2 \quad = \quad \mathrm{NaN}_1 \cdot \mathrm{finite}_1 + \mathrm{finite}_2 = 2$$

If the result of multiplication is an ∞ and the addend is the ∞ with the

opposite sign, then the result is zero, although normally it would be a NaN. If the addend is a NaN, then the result is zero. For the other values, indicated by "c" in table 2, `fmaX` operates as a usual fused multiply-accumulate except that the first argument (a multiplicand) is an extended range floating point number. For example, the fused multiplication and addition of finite arguments saturate to $\pm\infty$ in the usual way.

Finally, for exponential functions, which return fixed finite values for a wide range of inputs (including infinities), it is necessary to override the range reduction so that it produces an output which results in a constant value after the polynomial approximation. In the case of exponential, any finite value which results in a non-zero polynomial value will do, because the second `lookup` instruction returns 0 or ∞ and multiplication by any finite value will return 0 as required.

Lookup Internals The `lookup` instructions perform similar operations for each of the elementary functions we have considered. The function number is an immediate argument. In assembly language each function could be a different macro, while in high level languages the pair could be represented by a single function returning a tuple or struct.

A simplified data-flow for the most complicated case, $\log_2 x$, is represented in Figure 2. The simplification is the elimination of the many single-bit operations necessary to keep track of exceptional conditions, while the operations to substitute special values are still shown. Critically, the diagram demonstrates that the operations around the core look-up operations are all of low complexity. The graph is explained in the following, where letter labels correspond to the blue colored labels in Figure 2. This representation is for variant (ii) or (iii) for the second `lookup` and includes a dotted line on the centre-right of the figure at **(a)**, deliniating a possible set of values to save at the end of the first `lookup` where the part of the data flow below the line is computed in the second `lookup` instruction.

Starting from the top of the graph, the input **(b)** is used to generate two values **(c)** and **(d)**, $2^{-e}/\mu$ and $e+\log_2 \mu$ in the case of $\log_2 x$. The heart of the operation are two look up operations **(e)** and **(f)**, with a common index. In implementation (i) the look ups would be implemented separately, while in the shared implementations (ii) and (iii), the lookups could be implemented more efficiently together.

Partial decoding of subnormal inputs **(g)** is required for all of the functions except the exponential functions. Only the leading non-zero bits are needed for subnormal values, and only the leading bits are needed for normal values, but the number of leading zero bits **(h)** is required to properly form

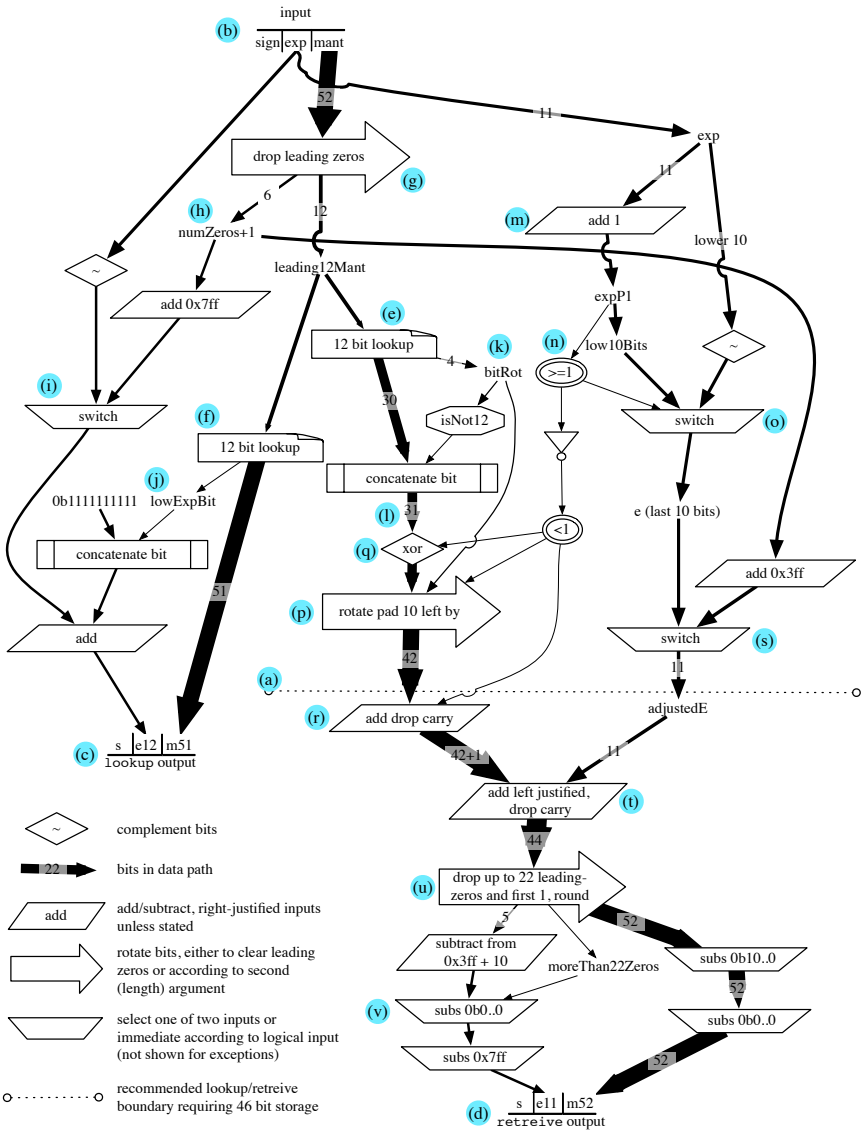

Figure 2: Bit flow graph with operations on vertices, for $\log x$ lookup. Shape indicates operation type, and line width indicates data paths width in bits. Explanation of function in the text.

the exponent for the multiplicative reduction. The only switch **(i)** needed for the first `lookup` output switches between the reciprocal exponents valid in the normal and subnormal cases respectively. Accurate range reduction for subnormals requires both extreme end points, *e.g.* 1/2 and 1, because these values are exactly representable. As a result, two exponent values are required, and we accommodate this by storing an exponent bit **(j)** in addition to the 51 mantissa bits.

On the right hand side, the look up **(e)** for the second `lookup` operation also looks up a 4-bit rotation, which also serves as a flag. We need 4 bits because the table size 2^{12} implies that we may have a variation in the exponent of the leading nonzero bit of up to 11 for nonzero table values. This allows us to encode in 30 bits the floating mantissa used to construct the second `lookup` output. This table will always contain a 0, and is encoded as a 12 in the bitRot field. In all other cases, the next operation concatenates the implied 1 for this floating-point format. This gives us an effective 31-bits of significance **(l)**, which is then rotated into the correct position in a 42-bit fixed point number. Only the high-order bit overlaps the integer part of the answer generated from the exponent bits, so this value needs to be padded. Because the output is an IEEE float, the contribution of the (padded) value to the mantissa of the output will depend on the sign of the integer exponent part. This sign is computed by adding 1 **(m)** to the biased exponent, in which case the high-order bit is 1 if and only if the exponent is positive. This bit **(n)** is used to control the sign reversal of the integer part **(o)** and the sign of the sign reversal of the fractional part, which is optimized by padding **(p)** after xoring **(q)** but before the +1 **(r)** required to negate a two's-complement integer.

The integer part has now been computed for normal inputs, but we need to switch **(s)** in the value for subnormal inputs which we obtain by biasing the number of leading zeros computed as part of the first step. The apparent 75-bit add **(t)** is really only 11 bits with 10 of the bits coming from padding on one side. This fixed-point number may contain leading zeros, but the maximum number is $\log_2((\text{maximum integer part}) - (\text{smallest nonzero table value})) = 22$, for the tested table size. As a result the normalization **(u)** only needs to check for up to 22 leading zero bits, and if it detects that number set a flag to substitute a zero for the exponent **(v)** (the mantissa is automatically zero). The final switches substitute special values for $\pm\infty$ and a quiet NaN.

If the variants (ii) or (iii) are implemented, the hidden registers must either be saved on context/core switches, or such switches must be disabled

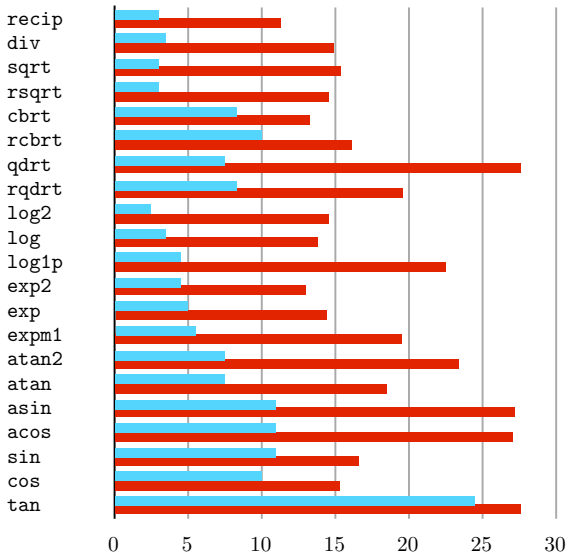

Figure 3: Throughput, measured in cycles per double, for implementations of elementary function with (upper bars) and without (lower bars) the novel instructions proposed in this paper.

during execution of these instructions, or execution of these instructions must be limited to one thread at a time.

2 Evaluation

Two types of simulations of these instructions were carried out. First, to test accuracy, our existing Cell/B.E. functional interpreter, see [1], was extended to include the new instructions. Second, we simulated the log lookups and `fmaX` using logical operations on bits, *i.e.*, a hardware simulation without timing, and verified that the results match the interpreter.

Performance Since the dependency graphs (as typified by Figure 1) are close to linear, and therefore easy to schedule optimally, the throughput and latency of software-pipelined loops will be essentially proportional to the number of floating-point instructions. Table 1 lists the expected throughput for vector math functions with and without the new instructions. Figure 3 demonstrated the relative measured performance improvements. Overall, the addition of hardware instructions to the SPU results in a mean 2.5× throughput improvement for the whole function library. Performance improvements on other architectures will vary, but would be similar, since the

acceleration is primarily the result of eliminating instructions for handling exceptional cases.

Accuracy We tested each of the functions by simulating execution for 20000 pseudo-random inputs over their natural ranges or $[-1000\pi, 1000\pi]$ for trigonometric functions and comparing each value to a 500-digit-precision Maple [11] reference. Table 1 shows a maximum 2.051ulp error, with many functions close to correct rounding. This is well within the OpenCL specifications [10], and shows very good accuracy for functions designed primarily for high throughput and small code size. For applications requiring even higher accuracy, larger tables could be used and polynomials with better rounding properties could be searched for using the lattice-basis-reduction method of [3].

3 Conclusion

We have demonstrated considerable performance improvements for fixed power, exponential and logarithm calculations by using novel table lookup and fused multiply-add instructions in simple branch-free accurate-table based algorithms. Performance improved less for trigonometric functions, but this improvement will grow with more cores and/or wider SIMD. These calculations ignored the effect of reduced power consumption caused by reducing instruction dispatch, function calls and branching and reducing memory accesses for large tables, which will mean that these algorithms will continue to scale longer than conventional ones.

For target applications, just three added opcodes pack a lot of performance improvement, but designing the instructions required insights into the algorithms, and even a new algorithm [2] for the calculation of these functions. We invite experts in areas such as cryptography and data compression to try a similar approach.

Some work in this paper is covered by US patent application 20110296146.

References

[1] Anand, C.K., Kahl, W.: An optimized Cell BE special function library generated by Coconut. IEEE Transactions on Computers (2009)

[2] Anand, C.K., Sharma, A.: Unified tables for exponential and logarithm families. ACM Transactions on Mathematical Software 37(3) (2010)

[3] Brisebarre, N., Chevillard, S.: Efficient polynomial L^∞ approximations. In: Proceedings of the 18th IEEE Symposium on Computer Arithmetic. pp. 169–176. IEEE Computer Society Press, Los Alamitos, CA, Santa Monica, USA (2007)

[4] Corporation, I.B.M.: Cell Broadband Engine Programming Handbook. IBM Systems and Technology Group, https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC (2008)

[5] Ercegovac, M.D., Lang, T.: Multiplication/division/square root module for massively parallel computers. Integr. VLSI J. 16(3), 221–234 (1993)

[6] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. SIGARCH Comput. Archit. News 39(3), 365–376 (Jun 2011)

[7] Fike, C.T.: Starting approximations for square root calculation on ibm system /360. Commun. ACM 9(4), 297–299 (1966)

[8] Gal, S.: Computing elementary functions: A new approach for achieving high accuracy and good performance. In: Proceedings of the Symposium on Accurate Scientific Computations. pp. 1–16. LNCS 235, Springer-Verlag, London, UK (1986)

[9] Gal, S., Bachelis, B.: An accurate elementary mathematical library for the IEEE floating point standard. ACM Trans. Math. Softw. 17(1), 26–45 (1991)

[10] Group, K.O.W.: The opencl specification version: 1.0 document revision: 29 (2008)

[11] Maplesoft: Maple 12 User Manual. Maplesoft (2008)

[12] Schmookler, M.S., Agarwal, R.C., Gustavson, F.G.: Series approximation methods for divide and square root in the Power3(TM) processor. In: ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic. p. 116. IEEE Computer Society, Washington, DC, USA (1999)

[13] Sharma, A.: Elementary Function Evaluation Using New Hardware Instructions. MSc thesis, McMaster University (2010)

[14] Tang, P.T.P.: Table-driven implementation of the logarithm function in ieee floating-point arithmetic. ACM Trans. Math. Softw. 16(4), 378–400 (1990)

[15] Tang, P.T.P.: Table-driven implementation of the expm1 function in ieee floating-point arithmetic. ACM Trans. Math. Softw. 18(2), 211–222 (1992)

[16] Taylor, M.B.: Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In: Proceedings of the 49th Annual Design Automation Conference. pp. 1131–1136. DAC '12, ACM, New York, NY, USA (2012)