2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)

ARM Floating-Point 2019: Latency, Area, Power

David R. Lutz
ARM Austin Design Center
david.lutz@arm.com

ABSTRACT

We have had little or no speed increase from process in the past few years, but latency continues to decrease due to algorithmic improvements [1] and a decision to spend more area on CPU datapaths [2]. A binary64 floating-point (FP) add now takes two cycles when done as part of a 2+2-cycle FMA, and even one cycle when done as part of an in-order vector reduction. Smaller and more specialized FP operations (bfloat16) are even faster. Finally, the decision to spend more area on datapath logic took a new twist this year when we applied it to GPUs, cutting dynamic power there by a third.

I. 2+2-CYCLE FMA

According to the IEEE 754-2008 standard, the operation FMA(s, a, x) computes $s+a\times x$ "as if with unbounded range and precision, rounding only once to the destination format". An FMA is usually implemented as a single pipeline, consisting of a multiply array followed by an add, a normalization shift, and rounding [3]. The augend s is shifted during the multiplier array reduction, so all three operands are needed at instruction issue time. Typical latency for an FMA is currently four cycles [4]. The traditional FMA pipeline is usually used for addition and multiplication as well as FMA, with adds computed as $s+(1\times x)$ and multiplies computed as $0+(a\times x)$, meaning that multiplies and adds also take four cycles.

We do not use the traditional FMA microarchitecture. Instead, we perform our FMAs as separate 2-cycle unrounded multiplies and 2-cycle adds, a strategy that improves the latency of addition, multiplication, and sums-of-products. For binary64, this means the multiplier returns an unrounded result comprising sign, 11-bit exponent, and 105-bit fraction; one of the adder inputs also has this format, with the extra fraction bits zeroed for non-FMA adds. The augend in our scheme is not used during the multiplication cycles, so FMAs accumulating to a common augend can issue every two cycles rather than every four cycles.

Our FMA is fully 754-2008 compliant in hardware, performing both normal and subnormal arithmetic at full speed, avoiding subnormal penalties that can be over 100 cycles and the associated security vulnerabilities [4], [5]. The latency (including forwarding) for all of our operations is 4 cycles for FMA, 3 cycles for multiply (the third cycle is for rounding), and 2 cycles for add.

The most common use of FMAs is for sums of products. Given the usual implementation, an in-order sum of four products $s = ax + by + cz + dw$ would take 16 cycles on a 4-cycle FMA, as shown in figure 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
fmul s,a,x	M	M	M	M												
fma s,b,y					F	F	F	F								
fma s,c,z									F	F	F	F				
fma s,d,w													F	F	F	F

Fig. 1. In-order sum-of-product latency for 4-cycle FMA

Given the same sum of 4 products, our 2+2-cycle FMA completes in only 9 cycles instead of 16, as shown in figure 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
fmul s,a,x	M	M	M													
fma s,b,y		M	M	A	A											
fma s,c,z				M	M	A	A									
fma s,d,w						M	M	A	A							

Fig. 2. In-order sum-of-product latency for 2+2-cycle FMA

An unpublished 2018 performance study performed at ARM found that the 2+2-cycle FMA improved SpecFP results by about 6% over a 4-cycle FMA [6]. This is more than the 3-4% published in [7], [8], suggesting that the advantage of our approach is increasing as overall FMA latency decreases.

Operation	4-cycle FMA	2+2-cycle FMA
FMA	4	4
add	4	2
multiply	4	3
in-order FMA sum of n products	$4n$	$2n+1$

TABLE I
LATENCY COMPARISON FOR TWO FMA IMPLEMENTATIONS

II. SINGLE-CYCLE BINARY64 ADDER

All of the latencies in the previous section include forwarding to important consumers: add, mul, FMA, and permute all forward to each other in "zero" cycles. In fact, the forwarding takes the majority of a cycle, and that portion of the cycle has increased with newer processes and additional vector units. The 2-cycle adder now uses less than a third of the second cycle for logic. Since our 2-cycle adder handles 106-bit FMA significands, we wondered if we could add in less than two cycles if we restricted ourselves to non-FMA adds.

978-1-7281-3366-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ARITH.2019.00025

We remade the near path with better LZAs and much better masking logic [1], also spending area to compute all 4 possible near-path differences: $a-b, a-(rshift1)b, b-a, b-(rshift1)a$. Far path shifting is started speculatively on both operands, and while the shifting of the smaller operand completes the larger operand is speculatively incremented (+1 and +2) for rounding. Three far path adders compute all of the possible rounded results. The floating-point adder includes two 54-bit incrementers and seven 54-bit adders in all, and the logic completely fills our cycle with no time left for forwarding. It is single-cycle, but not usable for the usual additions in one cycle.

III. In-Order Vector Reduction

ARM now has a scalable vector extension (SVE) option that does arithmetic on vectors that are multiples of 128-bits in length (e.g. 256 or 384 bits, with a maximum length of 2048). SVE includes an FADDA instruction that adds all of the elements of a vector to a scalar in order. The single-cycle adder in section II can be put to good use for this purpose.

We spend an initial cycle to analyze the first two inputs to the adder, in particular the signs (effective addition or subtraction) and exponents (normal/subnormal/least significant bit difference). We then begin cycle 2 with enough information to set up key muxes for the significand logic. Significands are not needed until a few inverter delays later. The exponent that is the result of the cycle 2 addition is computed slightly before the significand, and there is time to do the same analysis that we did in cycle 1 for sign and exponents (now using the result and the third input). Continuing in this manner we are able to add n vector elements in order in only $n+2$ cycles: 1 setup cycle, n iterative cycles, 1 forwarding cycle. Interestingly the exponent to exponent logic takes only one cycle, the significand to significand logic takes only one cycle, but the exponent plus significand logic for any given iteration is slightly longer than a cycle. We have long been used to clock skewing for multiple-cycle operations, but this is skewing within a single-cycle loop.

IV. Bfloat16

Bfloat16 is a non-standard format consisting of a sign bit, 8 exponent bits and 7 fraction bits; basically the top half of a binary32 number. The new format is important for machine learning, and very high bandwidth is desirable for matrix multiplications using it. The fundamental 128-bit operations for ARM are BFDOT: each 32-bit lane adds two bfloat16 products to a binary32 accumulator; and BFMMLA: 2×4 times 4x2 bfloat16 matrix multiply, added to a 2x2 binary32 matrix. BFMMLA is computed as two BFDOT operations, and it does a total of 16 bfloat16 multiplications per 128-bits of datapath per cycle.

Fortunately the matrix multiplications are tiny and always exact. They can be computed in about 1/2 cycle. The binary32 additions are not exact, but are much simpler than standard addition because subnormal inputs and results are treated as zeros. If we use the remainder of the multiplication cycle to swap smaller and larger magnitude adder inputs, we can also compute the binary32 addition in about 1/2 cycle, meaning that we have time in that cycle to swap larger and smaller operands for the next binary32 addition. The net effect is that we have much less speculative work to do for each addition, and the seven significand adders and two incrementers from section II can be replaced by two significand adders, saving considerable area.

V. GPU Area vs. Power

ARM usually combines functional units to save area on both GPUs and CPUs. For example, a binary32 multiplier can also perform one or two binary16 multiplies on the same multiplier array. Such an approach saves area, but not necessarily power. A binary32 multiplier array uses power propotional to $24 \times 24 = 576$, while a binary16 multiplier array uses power proportional to $11 \times 11 = 121$. If there are numerous binary16 multiplications, we can save power by building separate binary16 arrays and clock gating so that only the required functional units are used. We also save a bit of power even for binary32 multiplications because the binary32 array control logic is simplified.

GPUs also tend to use some variant of the classic FMA architecture (4-cycle FMA from section I). This design burns extra power if we are doing additions or multiplications rather than just FMAs. Replacing this design with a variant of our 2+2-cycle FMA (with separate multipliers and adders) allows us much finer control over where power is used. For example, the multiplier is not used at all if we are just doing an addition.

Combining these two ideas gives us 6 separate units (binary32 mul, binary32 add, 2x binary16 mul, 2x binary16 add) rather than one multifunctional unit (combined binary32 and 2x binary16 FMA). Using this strategy along with separate clock-gated flops for each unit saved about 33% of dynamic power compared to the original design.

GPUs are somewhat different from CPUs in that much more of their power is used by the datapath, but this result makes me wonder if we should reconsider our CPU designs using the same principal. Have we been too focused on area minimization?

References

[1] D. R. Lutz, "Optimized leading zero anticipators for faster fused multiply-adds," in *2017 Asilomar Conference on Signals, Systems, and Computers*, pp. 741–744, Oct. 2017.

[2] D. R. Lutz. http://arith22.gforge.inria.fr/slides/s1-lutz.pdf, June 2015.

[3] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[4] A. Fog. http://www.agner.org/optimize/microarchitecture.pdf, Sept. 2018.

[5] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*, pp. 623–639, May 2015.

[6] M. Kennedy. email, 2018.

[7] D. R. Lutz, "Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines," in *20th IEEE Symposium on Computer Arithmetic*, pp. 123–128, July 2011.

[8] S. Galal and M. Horowitz, "Latency sensitive fma design," in *20th IEEE Symposium on Computer Arithmetic*, pp. 129–138, July 2011.