Fused Multiply-Add Microarchitecture Comprising Separate Early-Normalizing Multiply and Add Pipelines

David R. Lutz
ARM
3711 S. Mopac Expressway
Building 1, Suite 400
Austin, TX 78746
phone: +1 (512) 732-8076
david.lutz@arm.com

Abstract

We present an IEEE 754-2008 and ARM compliant floating-point microarchitecture that preserves the higher performance of separate multiply and add units while decreasing the effective latency of fused multiply-adds (FMAs). The multiplier supports subnormals in a novel and faster manner, shifting the partial products so that injection rounding can be used. The early-normalizing adder retains the low latency of a split path near/far adder, but does so in a unified path with less area. The adder also allows rounding on effective subtractions involving one input that is twice the normal width, a necessary feature for handling FMAs. The resulting floating-point unit has about twice the (IPC) performance of the best previous ARM design, and can be clocked at a higher speed despite the wider paths required by FMAs.

I. Introduction

With the publication of the IEEE 754-2008 standard, fused multiply adds (FMAs) have become a requirement for floating-point units. FMAs usually have higher accuracy than separate multiplies and adds, but despite many claims about better performance (most of which seem to involve systems that are constrained with respect to issue slots), in our experience FMAs have had significantly worse performance. In this paper we explore why the traditional fused microarchitecture is slow, and then demonstrate a new microarchitecture that delivers fused results while retaining the better performance afforded by separate multiplies and adds.

II. Fused Performance

According to the standard, the "operation $fusedMultiplyAdd(x, y, z)$ computes $(xy) + z$ as if with unbounded range and precision, rounding only once to the destination format." [10] An FMA is typically implemented as a single pipeline, consisting of a multiply array followed by an add, a normalization shift, and rounding [6]. The augend z is shifted during the multiplier array reduction, so all three operands are needed at instruction issue time. For a high-performance ARM processor, we estimate that this kind of pipeline would take about 7 cycles, significantly longer than our separate add (4 cycles) or multiply (4 or 5 cycles) pipelines, but shorter than a single multiply followed by a single add. The traditional fused pipeline is typically used for all operations, so adds are computed as $(x * 1) + z$ and multiplies are computed as $(x * y) + 0$, meaning that all multiplies and adds, indeed nearly all floating-point instructions, take 7 cycles.

The most obvious reason that traditional FMA implementations are slower is that adds and multiplies are slower. At least in our benchmarks, most instructions are not FMAs, and all of these non-FMA instructions benefit from a significantly shorter multiply or add pipeline. Latency is important for performance, and a 4-cycle add is much better than a 7-cycle add. There has been some recognition of this problem, and anecdotally we have heard of some companies putting in separate adders alongside their fused units, but this adds area and doesn't solve the problem of slow multiplies. Even worse, it turns out that dependent FMAs are slower on the traditional pipeline than they are when implemented as separate multiplies and adds.

A. Dependent FMAs

FMAs are not used in isolation. At least in our benchmarks, the result of an FMA tends to be passed to another FMA, most typically in some kind of dot product such as $ax+by+cz+dw$. ARM processors are usually programmed in C, so this dot product is evaluated left-to-right, as a multiply followed by 3 multiply-adds.

In the following figures MUL is a floating-point multiply, and FMA is a fused multiply-add. In figure 1 we have the conventional FMA microarchitecture, with a 7-cycle pipeline that performs all floating-point operations. The right-hand column of the table shows the 7 fused stages, f1 to f7. Since all three operands are needed before the next FMA can begin, the dot product requires a full 28 cycles (t, the leftmost column).

t	instructions	f1	f2	f3	f4	f5	f6	f7
1	1 MUL D0,D1,D2	1						
2			1					
3				1				
4					1			
5						1		
6							1	
7								1
8	2 FMA D0,D3,D4	2						
9			2					
10				2				
11					2			
12						2		
13							2	
14								2
15	3 FMA D0,D5,D6	3						
16			3					
17				3				
18					3			
19						3		
20							3	
21								3
22	4 FMA D0,D7,D8	4						
23			4					
24				4				
25					4			
26						4		
27							4	
28								4

Figure 1. Traditional fused dotproduct, $ax + by + cz + dw$

Splitting the FMA into a 4-cycle multiply followed by a 4-cycle add (figure 2) would seem to be slower, but note that we don't need the augend at the beginning of the multiply part of the operation. Indeed, the augend is not needed until cycle 5 of the FMA, which with proper scheduling makes the computation of $ax + by + cz + dw$ much faster.

The multiply (ax) starts at time 1. Since we are not dependent on that result to start the first FMA, we can start that at time 2 (by, eventually to become $ax + by$). When the FMA requires ax, at time 5, the result is available because the first multiply has completed. Similarly, the second FMA ($ax + by + cz$) can begin at time 6 even though the augend $ax + by$ will not be produced until time 10.

In general, the time between dependent FMAs is based on add time rather than multiply-add time. The seemingly higher-latency microarchitecture has lower latency in practice (about $4/7$ of the latency given these example pipelines).

More generally, if an add takes a cycles, a multiply takes m cycles, and a classic FMA takes f cycles, then the sum of n products takes fn cycles on the classic fused architecture,

t	instructions	m1	m2	m3	m4	a1	a2	a3	a4
1	1 MUL D0,D1,D2	1							
2	2 FMA D0,D3,D4	2	1						
3			2	1					
4				2	1				
5					2				
6	3 FMA D0,D5,D6	3				2			
7			3				2		
8				3				2	
9					3				2
10	4 FMA D0,D7,D8	4				3			
11			4				3		
12				4				3	
13					4				3
14						4			
15							4		
16								4	
17									4

Figure 2. Fused dotproduct implemented with separate multiplies and adds, $ax + by + cz + dw$

but only $a(n-1) + m + 1$ cycles on our architecture. When $m \approx a$, as in our parts, then our method is faster as long as $a < f$, saving approximately $n(f - a)$ cycles.

B. Performance Modeling

We used RealView SoC Designer (formerly MaxSim), a cycle based simulator, with a Cortex-A8 cycle accurate model to measure the performance difference between the two implementations of FMA. The NEON SIMD execution unit was our baseline. This unit can initiate two multiply-adds per cycle. Multiplies take 4 cycles, adds take 4 cycles, and mutliply-adds (which are not fused) take 8 cycles. Multiply-adds can be initiated without the augend being available as long as the augend will be available by the time the 4-cycle add begins. Results obtained under this scenario are labeled "4+4 stage".

The traditional fused scenario is modeled so that all floating-point instructions take 7 cycles. Results obtained under this scenario are labeled "7 stage".

ARM processors tend to be programmed in C, and we know of no good fortran compiler, so we have no good way to run SPECfp. At the time this experiment was run, we had only a very small set of benchmarks suitable for SIMD simulation, namely 4 graphics benchmarks from *Quake*, and two audio benchmarks from MP3:

- affinetris - affine rasterizer
- geometry - triangle processing
- lighting - applies simple lighting
- persptris - perspective rasterizer
- imdct - inverse modified discrete cosine transform
- synfilt - synthesis filter

Table 1 shows the results of these runs. The two columns

%FP and %FMA show the percentage of executed instructions that were floating-point instructions and the percent of instructions that were FMAs. The 4+4 implementation beats the traditional fused implementation in every case, and does particularly well in the geometry and lighting benchmarks that make heavy use of dot products.

Benchmark	%FP	%FMA	7-Stage Cycles	4+4-Stage Cycles	Speedup
affinetris	34%	5%	3012	2928	3%
geometry	75%	19%	1245	1075	16%
lighting	67%	18%	2876	2336	23%
persptris	32%	6%	3710	3583	4%
imdct	75%	8%	968	929	4%
synfilt	64%	42%	29159	28221	3%

Table I
BENCHMARK RESULTS FOR TWO IMPLEMENTATIONS OF FMA

Both theory and our benchmarking agree that splitting the FMA into separate multiplies and adds helps with performance. The rest of this paper explores how to do this split.

III. EARLY-NORMALIZING MULTIPLIER

While multiplier latency is unimportant for our FMA latency, it is still valuable in our benchmarks, and saving a cycle here is worth about 2% to overall floating-point performance. The challenge is how to maintain minimum latency in the presence of subnormal inputs and results. One of our design goals is to not raise any exceptions, so we have to deal with subnormals in the main processing flow – no flushing the pipeline and normalizing inputs or denormalizing outputs. The contribution of this section is to show how to do all this while adding only one cycle to the multiplier, namely an early-normalization cycle before the final addition.

Our basic scheme is to use injection rounding. A good overview of multiplier rounding is given in [1] (but note that none of the three presented methods handle subnormals). Our method is conceptually simpler than any of these, consisting of injections into the appropriate bits of two 106-bit adders, one for results in the range $[1, 2)$ and one for results in the range $[2, 4)$. The carry-out bit of the low-order adder determines which adder contains the correct result, and the selected result is already correctly rounded (except that the low order bit might have to be zeroed for round to nearest even).

Figure 3 shows the multiplier datapath. The multiplier consists of 5 stages, one of which is optional. The first three stages, E1 to E3, are just a standard Booth recoding followed by an array reduction from 27 to 2 partial products. Stage E1 is light on logic to allow forwarding paths. The end of stage 3 (which in the diagram is shown at the end of the optional stage E4N) consists of some half-adder based manipulation of the two partial products to allow for the

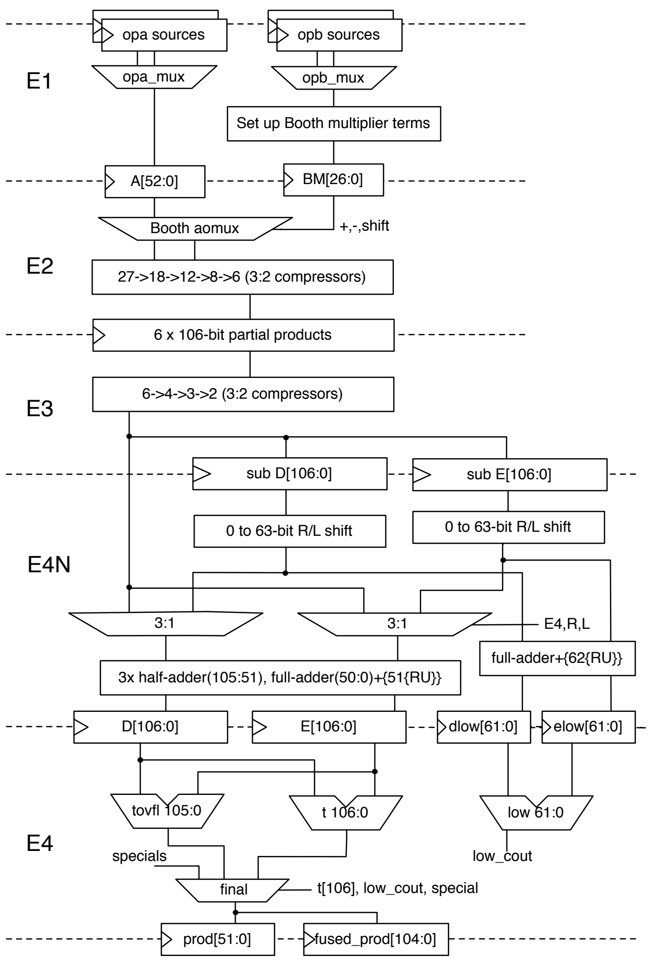

Figure 3. Multiply Datapath

rounding injection. Two spaces for the injection are made by "adding" the upper bits three times with half adders, while a full adder incorporates most of a "round up" injection in the low order 52 bits. Since three levels of half adders open up three consecutive zeros in the carry word, the carry out of the full adder can be accomodated while still leaving two consecutive zeros for rounding.

Stage E4 (please disregard E4N for now) is just an addition with injection rounding at two fixed locations. A carry out from the adder *t* indicates that the sum is in the range $[2, 4)$, and thus the result from the overflow adder, *tovfl* should be used. If there is no carry out, then the *t* adder produces the correct result.

The design as presented works for ARM's flush-to-zero mode, a mode in which subnormal inputs or outputs are treated as zeros. For IEEE 754 compliance, we need additional logic.

If one of the inputs is subnormal, then the two partial products have leading zeros, injection rounding will usually not be correct at the fixed locations, and the final sum will require a nontrivial normalization followed by rounding. A second problem occurs if the exponents are small enough so that the result is subnormal. In this case rounding cannot be done until the final sum is denormalized. Previous designs have dealt with these problems by adding, then normalizing the sum, and then rounding the sum. Given the frequency target of our design, this technique would have cost us two additional cycles: one for normalization/denormalization, and one for rounding.

We solve both these problems by shifting the two partial products so that their sum will be appropriately normalized/denormalized. This shifting, left or right, occurs in stage E4N (E4 normalization). The shift amount is completely determined by the two input exponents and the number of leading zeros in the input mantissas. A normal result with one subnormal input will require left shifting of the partial products. A subnormal result can require either left or right shifting, depending on the computed location of the leading one and whether the product had a subnormal input. In any case, once the partial products are shifted the addition and injection rounding can be handled in only one cycle.

Right shifting, which only happens for subnormal results, loses data from the main adder, both sticky bits and whether the shifted out bits generate a carry into the main adders. We save these bits and add them with a 62-bit adder, marked "low" in the diagram. Since the overflow adder, tovfl, is not needed for a subnormal result (it being impossible to reach the $[2, 4)$ range when adding subnormals), we reuse it to compute the same sum as the t adder, but with the addition of a carry-in bit. The carry out of the 62-bit low adder is used to pick between the two adder results.

The final mux has 4 possible choices: output of the t adder, overflow output of the tovfl adder, subnormal output of the tovfl adder, or special results (NaN or infinity). The chosen result is already correctly rounded, with the exception of a 1-bit fixup for results that incorrectly round to an odd number.

For a multiply that is part of a fused multiply-add, there is no rounding injection, and a full 106-bit mantissa is forwarded to the adder. The exponent is still 11 bits, but we pass along several extra bits of information: whether the exponent has overflowed or underflowed, and whether the product is a true infinity, NaN, or zero.

IV. Early-LZA Unified-Path Adder

The add pipeline performs all operations other than multiply or divide/square root, and it is by far the most important determinant of floating-point engine performance. Floating-point addition is more complicated than multiplication, consisting of (1) exponent analysis and difference computation, (2) operand alignment, (3) addition (with rounding injection), and (4) normalization. For many years, the state of the art in adders has been some form of near/far split, which saves a cycle by noticing that nontrivial alignment and nontrivial normalization are mutually exclusive. The near path is for differences involving exponents that are equal or off by one, differences that cause some cancellation of leading bits. Such differences don't require rounding, but they do require normalization after the addition. The far path is for all other sums and differences, and it requires alignment and rounding but only trivial (1-bit) normalization. This split add path was first published in [2], and has been refined in several other designs [7], [5].

A near/far adder with injection rounding would require only three cycles in our model (plus one additional cycle for operand forwarding and selection). Unfortunately the near path does not work in the presence of a double-length operand, such as the multiplier output during an FMA. The problem is that after the sum is computed the remaining number still requires rounding, and it is not properly aligned for an injection. After the normalization step we can round, but it requires an additional cycle. The contribution of this section is to show how to handle this situation by performing an early normalization before the computation of the sum, obviating the need for an extra cycle and saving the area required for the near path.

Figure 4 shows the adder datapath. The adder consists of 4 stages: E1 is operand forwarding and selection; E2 selects larger and smaller operands and computes the LZA of the operand differences; E3 either shifts the smaller operand to match exponents or shifts both operands to prenormalize the eventual sum; and E4 adds with an injection rounding to get the final result. This pipeline is quite similar to a typical far path pipeline, although it has two surprising features: an LZA in E2 and a left shift in E3.

The LZA (leading zero anticipator, a way to anticipate how many leading bits are lost due to cancellation [8]) happens much earlier than in a conventional near/far adder. There are actually two LZAs. We examine the low order bit of the input exponents and require a 1-bit shift if they differ. We then compute the LZAs of $a - (possibly\ shifted)b$ and $b - (possibly\ shifted)a$. We simultaneously determine which operand is actually larger and use that information to pick the correct LZA value. If the exponent difference is not zero or one, or if the operation is not an effective subtraction, then the LZA value is ignored.

In traditional near/far adders, the LZA is done at the same time as the addition, with the LZA value being used to normalize the sum. We do the LZA in E2, two cycles before the addition in E4. We use the computed LZA to prenormalize (left shift) the operands in E3, discarding the leading bits of the operands. Since these leading bits just produce zeros when added, we have not lost any information. Furthermore, the operands are now properly aligned for a rounding injection, so the two adders in E4 produce properly rounded results. Serendipitously, the LZA correction problem [3] is handled automatically by having two adders in

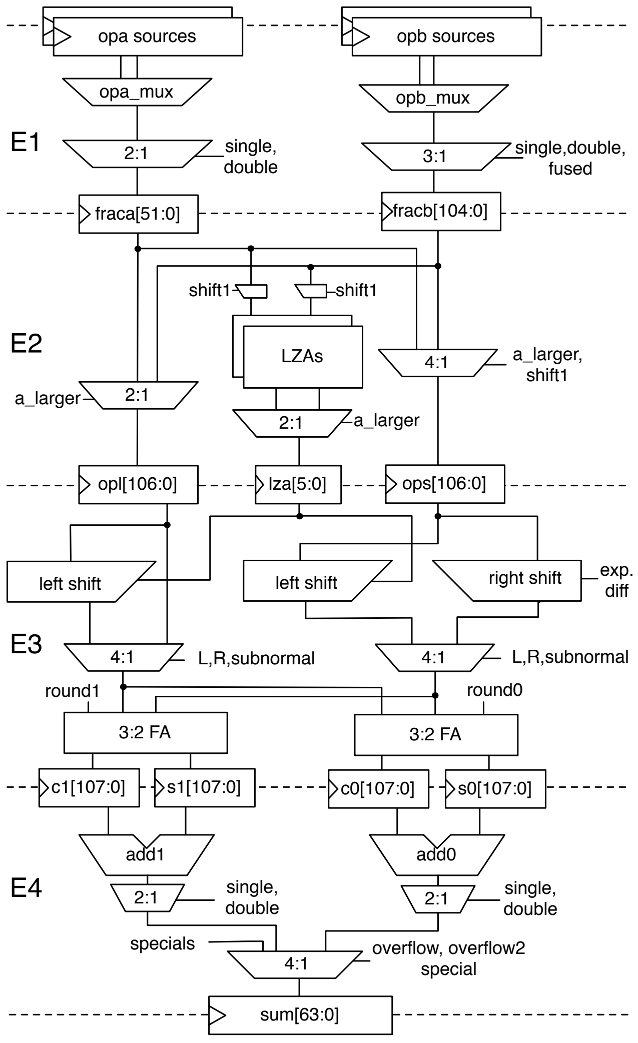

Figure 4. Add Datapath

E4.

There is no true near or far path in our adder. Operations that would have used the far path ignore the LZA result and right shift the smaller operand by an amount computed from the exponent difference of the two operands. Operations that would have used the near path use the LZA result to left shift the two operands in E3. In both cases, rounding constants are added with 3:2 adders at the end of E3. This is slightly inelegant (and wasteful of flops) compared to our multiplier rounding, but the rounding constants are quite a bit more complicated in the adder.

Because of a peculiarity of fused inputs, the final sum can actually achieve the value 4.0 (denoted overflow2 in the mux selects). The problem is that the fused result is not rounded, and so it may have an incorrect exponent. Suppose that the fused result consists of more than 54 leading ones. If rounded it would have a higher exponent, but it is not rounded. If we now subtract a number near to this one, the exponent can have any of three values: one less than the stated exponent, equal to the stated exponent, or one more than the stated exponent. The way the adder handles subtraction is to assume 1 less than the larger exponent and then allow the sum to overflow to that exponent. With fused we can also overflow to the next higher exponent.

Subnormals are not nearly as problematic for the adder as they are for the multiplier. The main changes for subnormals are changing the leading bit in opl and ops in E2, and adjusting the shift amount by one in the 4:1 muxes in E3. There is also a potential for overflow2 in the final addition (a difference can be subnormal, normal with minimum exponent, or normal with exponent minimum plus 1). Other than these few oddities, subnormals fit nicely into the normal flow.

Finally, we developed this early-LZA adder to accomodate FMAs, but it turns out to be an improvement over near/far adders even if the design does not do FMAs. In our adder, the entire near path is replaced by two left shifters and two LZAs. The timing is equivalent to a near/far adder (better if doing FMAs), but the adder is considerably simpler and smaller.

V. Conclusion

Most of section II is adapted from a presentation we made at the Asilomar Conference in 2006. At that time, we argued that fused performance is much worse than what is available given separate multiplies and adds. In hindsight, it turns out that fused implementation was responsible for the bad performance, not FMAs themselves.

The idea of using injection rounding in an FMA unit was presented in [4], but the approach differs from ours in that it retains a monolithic FMA microarchitecture, i.e., the augend and the product terms are all input at the beginning of the FMA operation. An essential part of our design is that the augend is independent of the multiplication, and is not required until the multiplication has completed.

An early-normalizing adder is described in [9], but the overall floating-point unit is much simpler than what is presented here, with no subnormal support and no FMA support.

This paper has shown that a floating-point unit can perform FMAs more quickly by splitting them into separate multiplies and adds. It seems counterintuitive that this would be true because a split FMA actually takes more cycles than a traditional FMA; the key insight is that dependent split FMAs can be issued based on add time rather than multiply-add time. The other compelling advantage of the split implementation is that it does not compromise the performance of any other floating-point operations.

The early normalizing multiply and add units presented in this paper are slightly simplified versions of the highest performance floating-point microarchitecture ever produced by ARM. The realization of this microarchitecture more than doubles the floating-point instructions per clock in our benchmarks, and at the same time enables higher frequency. The improvement comes from many things, including better memory system performance and out-of-order execution, but better latency in the floating-point engine itself and better control logic for that engine are a large part of it.

In the excellent *Handbook of Floating-Point Arithmetic* [6] one of the authors says: "The current trend in the processor world is to converge toward a single unifying operator, the FMA. The FMA replaces addition, subtraction, and multiplication, delivers to most applications better throughput and better accuracy for a reduced register usage, and allows for efficient and flexible implementations of division, square root, and elementary functions." We disagree with most of this statement, at least as it applies to the traditional FMA implementation, and hope that this paper provokes some changes in future implementations. The traditional FMA has worse latency for addition, subtraction, and multiplication, and even FMA throughput is worse than what can be achieved with separate multiply and add pipelines. Division and square root based on FMAs have far worse latency than what is available using iterative algorithms. Traditional FMAs do not even have good latency for dot products. The sometimes better accuracy of FMAs is just as available with the implementation presented here. FMA performance is better on separate multiply and add pipelines.

Acknowledgement

The author is grateful to the anonymous referees for highlighting previous related work.

References

[1] Guy Even and Peter-Michael Seidel. A comparison of three rounding algorithms for ieee floating-point multiplication. *IEEE Transactions on Computers*, 49(7):638–650, July 2000.

[2] Paul Michael Farmwald. *On the Design of High Performance Digital Arithmetic Units*. PhD thesis, University of California Livermore, 1981.

[3] Chris N. Hinds and David R. Lutz. A small and fast leading one predictor corrector circuit. In *2005 Asilomar Conference on Signals, Systems, and Computers*, pages 1181–1185, October 2005.

[4] Tomas Lang and Javier D. Bruguera. Floating-point multiply-add-fused with reduced latency. *IEEE Transactions on Computers*, 53(8):988–1003, August 2004.

[5] David Raymond Lutz and Christopher Neal Hinds. Data processing apparatus and method for performing floating point addition. *US Patent Number 7,437,400*, October 14, 2008.

[6] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[7] Ajay Naini, Atul Dhablania, Warren James, and Debjit Das Sarma. 1-GHz HAL SPARC64 dual floating-point unit with RAS features. In *15th IEEE Symposium on Computer Arithmetic*, pages 173–183, June 2001.

[8] Martin S. Schmookler and Kevin J. Nowka. Leading zero anticipation and detection – a comparion of methods. In *15th IEEE Symposium on Computer Arithmetic*, pages 7–12, June 2001.

[9] Hon P. Sit, Monica Rosenrauch Nofal, and Sunhyuk Kimn. An 80 mflops floating-point engine in the intel i860 processor. In *International Conference on Computer Design*, pages 374–379, October 1989.

[10] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, 2008.