2011 20th IEEE Symposium on Computer Arithmetic

Latency Sensitive FMA Design

Sameh Galal, Mark Horowitz
Stanford University
{sameh.galal,horowitz}@stanford.edu

***Abstract*— The implementation of merged floating-point multiply-add operations can be optimized in many ways. For latency sensitive applications, our cascade design reduces the accumulation dependent latency by 2x over a fused design, at a cost of a 13% increase in non-accumulation dependent latency. A simple in-order execution model shows this design is superior in most applications, providing 12% average reduction in FP stalls, and improves performance by up to 6%. Simulations of superscalar out-of-order machines show 4% average improvement in CPI in 2-way machines and 4.6% in 4-way machines. The cascade design has the same area and energy budget as a traditional fused multiple-add FMA.**

Keywords- Fused Multiply Add

I. INTRODUCTION

A high performance floating-point unit is a major component of modern CPU and GPU designs. Its applications range from multimedia and 3D graphics processing to scientific and engineering applications. Most recent designs incorporate an integrated multiply-accumulate unit due to the frequency of multiply-accumulation operations, such as those found in dot products and matrix multiplication. These multiply-accumulate units usually implement the fused multiply add operation (FMA) which was first introduced in IBM RS6000 FPU [1]. This operation has been recently added to the IEEE floating-point arithmetic standard, IEEE741-2008. The standard defines fusedMultiplyAdd(A, C, B) as the operation that computes $(A \times C) + B$ initially with unbounded range and precision, rounding only once to the destination format. As a result, fused multiply-add has lower latency and higher precision than a multiplication followed by an addition.

In this paper we focus on optimizing the design of FPUs in CPUs, which are more latency sensitive than GPU designs. We evaluate the design alternatives using the SPEC CFP2000 floating point benchmark suite [2]. To understand how FP latency affects these applications, we classify FMA dependencies according to where the result is used in a subsequent instruction, as shown in Figure 1:

- Accumulation Dependency: the result is accumulated in a subsequent ***fadd*** or ***fmadd*** instruction (bypass through f_B).
- Multiply-Add Dependency: the result goes through a fused multiply and then an add (bypass through f_A or f_C).
- Other Dependencies: the dependent instruction is not ***fmadd***, ***fadd*** or ***fmul***.

For the rest of the paper we will use a tuple notation to indicate the latency for the different kind of dependencies to compare different designs. For example, a (3,7,8) design has a 3 cycle accumulation latency, 7 cycle multiply-add latency and an 8 cycle latency for other non-FMA dependent instructions.

Traditional FMA design does not make a distinction between the latency of accumulation and multiply-add, resulting in designs that have equal latencies for all dependencies. For example, the IBM Power5 FMA is a (6,6,6) design, but the Power6 FMA is (6,6,7), because the design is optimized to handle feed forwarding of dependent instructions before the rounding stage [3]. We review such a design in Section 3 and use it as a reference for a state of the art FMA design. We then introduce our cascade implementation of the FMA instruction (CMA) which has been optimized for accumulation dependencies with a small effect on the other latencies. CMA allows the accumulation operand f_B to enter the pipeline much later than in a traditional FMA implementation, allowing for shorter accumulation latency. We then optimize this path by introducing overlapping bypass paths for exponent and significand to make the accumulation dependent latency as short as possible. We demonstrate how a CMA can achieve a (3,7,8) latency at the same clock rate of an FMA(6,6,7). Figure 2 shows the FMA and CMA pipelines and their bypass paths and how these bypass paths reduce the effective latency of the instructions. The cascade design can achieve

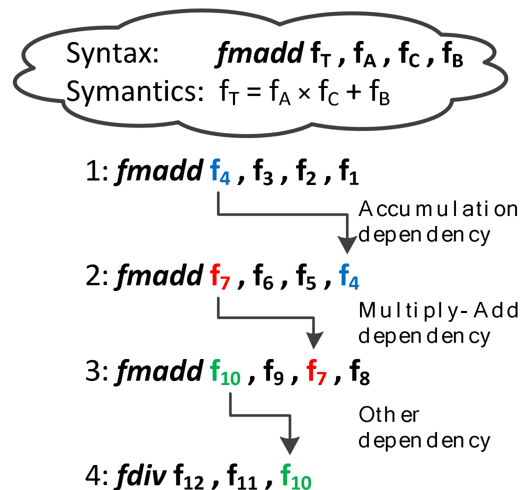


Figure 1: FMA dependency types

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.26

IEEE computer society

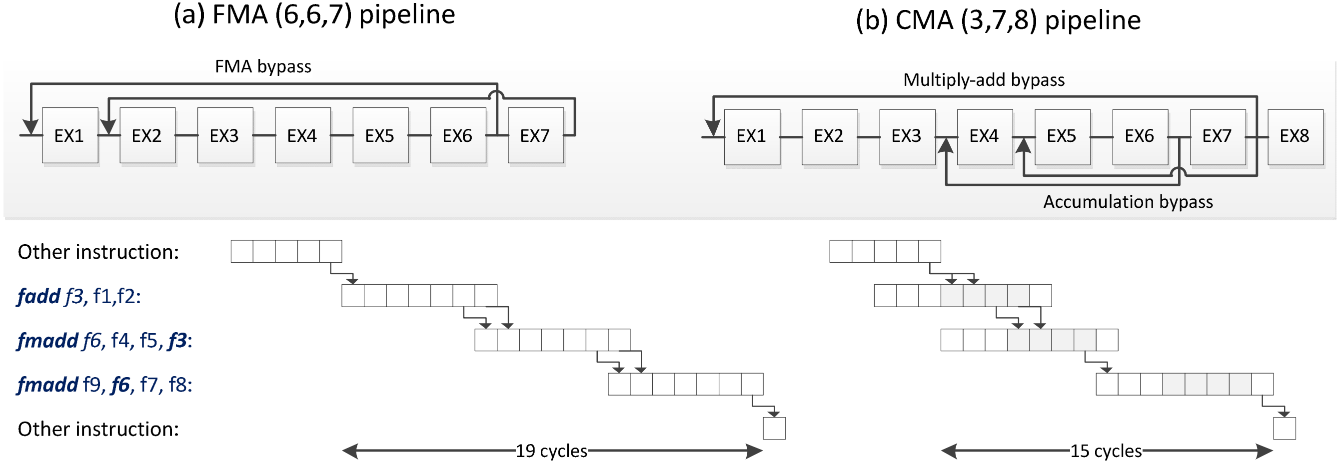


Figure 2: Block diagram of FMA and CMA pipelines with their respective bypass paths, and a timing diagram of an example instruction trace for both pipelines. The CMA architecture has shorter accumulation latency than FMA.

such aggressive accumulation latency because of staggered feedback as shown in Figure 2 (and elaborated in Section 4).

Section 5 presents the delay, power and area results of implementing the CMA and FMA design, and Section 6 concludes the paper.

II. Application Study

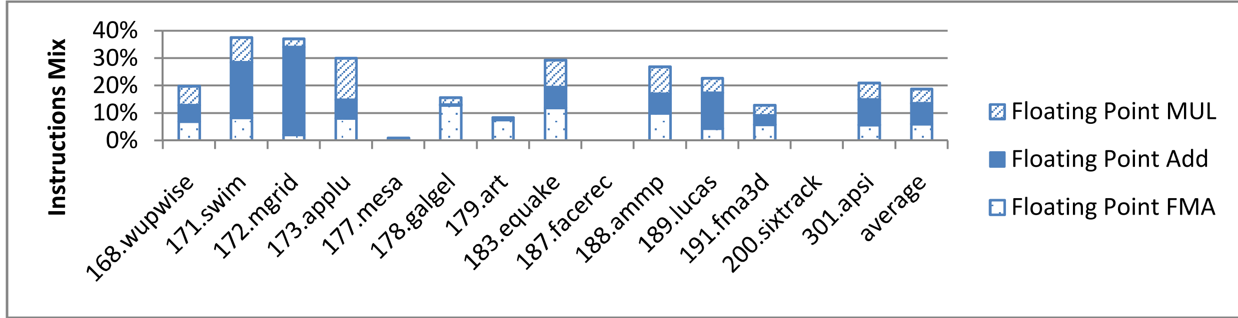
The effect of the different instruction dependencies in FPU design and their respective latencies is application dependent, since for applications with parallelism, data dependencies can be hidden by interleaving execution of parallel (non dependent) work to keep the machine busy during the "stall" time. For example, on a 6 stage FPU, interleaving the execution of 6 threads will keep the unit busy and hide any data dependencies. This technique is used in GPU designs. For such parallel workloads, W/GFlops and mm²/GFlops are the critical parameters to optimize [4].

For applications with less parallelism, the performance effect of these latency changes are important, and depend on the amount of parallelism that the processor can extract from the application: only when FP operations are on this critical path will the latency changes matter. We first studied a simple single-issue, in-order model to quickly explore the frequencies of the different dependency paths, and to gain intuition for the types of trade-offs that might exist. To provide this information, we modified the M5 architecture simulator [5] built for the PowerPC architecture to count the three different FP latency stalls. The modified simulator stored the total number of stalled cycles for every design and calculated the average latency penalty by dividing by the total number of ***fmadd, fmul*** and ***fadd*** instructions. Finally, we calculated the average time penalty by dividing the average latency penalty by the clock frequency.

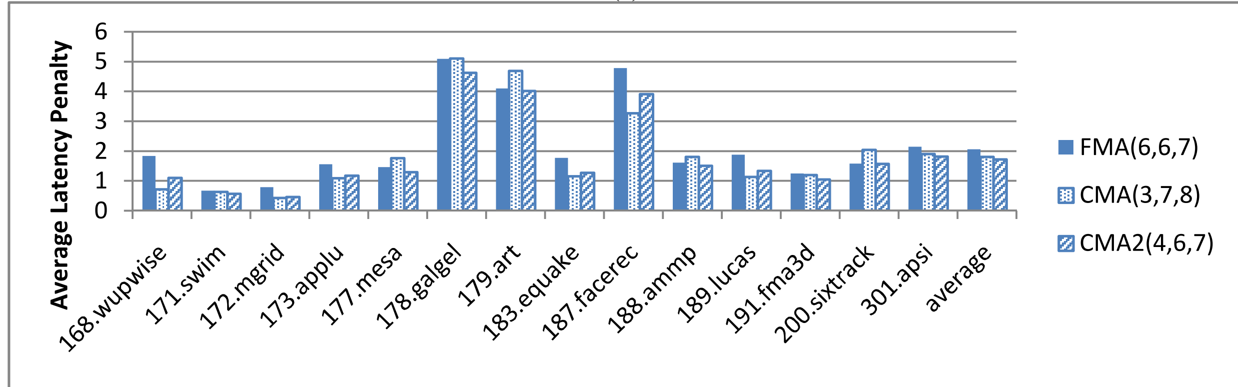
This study revealed the importance of the accumulation latency, so we focused on creating a design which maximized the overall performance (at small power changes) using asymmetric latencies. In the end we compared FMA (6,6,7), CMA (3,7,8), and CMA2(4,6,7). The last design is a variation of CMA which uses more hardware and power to get a slightly better multiplier-add latency at the cost of increasing the accumulation latency. These latencies were chosen based on the delay characteristics of the respective designs so that all of them run at the same clock frequency. We simulated the reference set of CFP2000 benchmarks using the gcc PowerPC cross compiler with the –O3 optimization directive. The PowerPC architecture was chosen because it has had the FMA instruction for a long time and has more mature FMA compiler support. The compiler optimizes for a 6 cycle FPU, which matches our base FMA architecture.

Figure 3 shows the in-order model results. On average, ***fmadd***, ***fmul*** and ***fadd*** instructions make up around 20% of these application's instructions, but are much smaller in three (mesa, facerec, and sixtrack). We ignore these applications in the averages in Figure 3(b) and (c) since FP performance is not critical for them. Figure 3(b) shows the average latency penalty for each application. CMA(3,7,8) achieves an average latency penalty of 1.81 cycles across the benchmark which is 13% lower than the 2.07 average latency penalty incurred by the FMA(6,6,7) design. CMA2(4,6,7) achieves a slightly better average latency penalty of 1.73, but in this simple model, the change in the two latencies essentially balances out. Figure 3(c) shows the performance loss from FP stalls. FMA(6,6,7), CMA(3,7,8) and CMA2(4,6,7) incur total performance penalties of 41%, 33.7% and 33.1% respectively. Therefore, CMA(3,7,8) and CMA2(4,6,7) architectures will be ~5-6% faster than an FMA(6,6,7) architecture at the same clock frequency, if the average instructions per cycle (IPC) of all non-FP instructions is one.

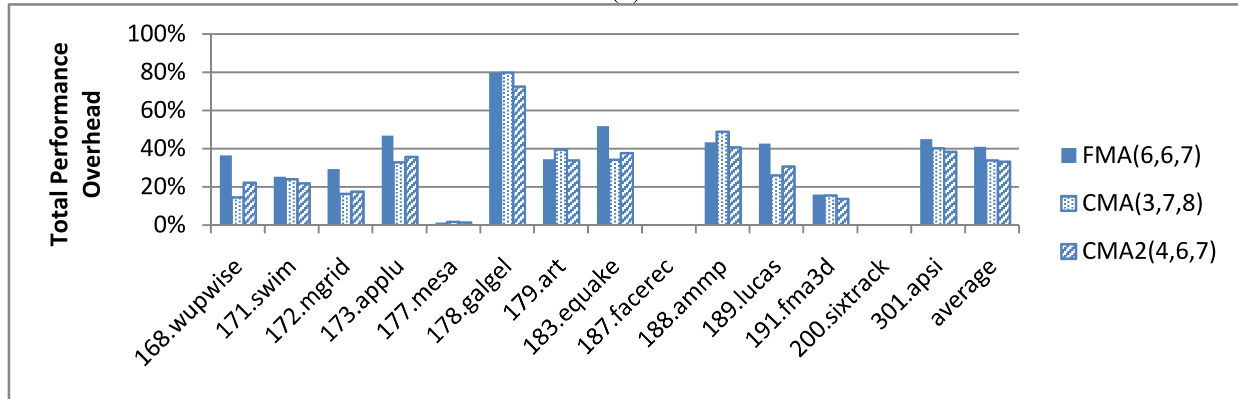
An in-order machine is very latency sensitive, as any subsequent dependent instructions stalls the pipeline execution until the floating point instruction has finished. Out of order superscalar designs are less latency sensitive because they exploit instruction level parallelism (ILP) to find non-dependent instructions to issue while waiting for executing instructions, resulting in higher IPC. However, long FPU latency still affects performance when the



(a)



(b)



(c)

Figure 3: CFP 2000 benchmark on a simple single-issue in-order model. (a) Floating point instruction mix as percentage of total number of instructions. (b) Average latency penalty (c) Total performance overhead (assuming IPC=1 except for FP operations) for FMA(6,6,7), CMA(3,7,8) and CMA2(4,6,7) designs.

available ILP is not enough to keep the functional units busy, resulting in stalls.

To test the effectiveness of the proposed cascade design in out of order machines, we modified the scheduler of the out of order model of the M5 simulator to support the FMA(6,6,7) and CMA(3,7,8) architectures. For the CMA(3,7,8) design, the scheduler was modified to allow ***fadd*** and accumulation-dependent ***fmadd*** instructions to issue up to 5 cycles earlier if the critical operand was produced by preceding ***fmadd, fmul*** or ***fadd*** instructions and up to 3 cycles earlier if produced by other instructions. Additionally, dependent ***fmul*** and multiply-add dependent ***fmadd*** are issued up to one cycle earlier. On the other hand, for the FMA(6,6,7) scheduler, any accumulation dependent or multiply-add dependent ***fmadd, fmul*** or ***fadd*** instructions are issued up to 1 cycle earlier. Using the modified model, the CFP 2000 benchmarks were run with 2-FPU and 4-FPU configurations to see how the performance improvement scales with increased number of functional units, which should increase the sensitivity to FPU latency. The results of the floating point rich benchmarks are summarized in Table 1. The CMA design shows an average reduction in cycles per instruction (CPI) over FMA of 3.97% for the 2-FPU case and 4.62% for the 4-FPU machine as illustrated in Figure 4.

The results indicate that the proposed CMA(3,7,8) design achieves an average performance improvement of 4-6% for a

TABLE 1
OUT OF ORDER PERFORMANCE RESULTS

Benchmark Name	Instruction Mix *fmadd*	*fadd*	*fmul*	Total Floating	CPI (Cycles Per Instructions) 2-FPU configuration CMA (3,7,8)	FMA (6,6,7)	4-FPU configuration CMA (3,7,8)	FMA (6,6,7)
168.wupwise	9%	7%	10%	25%	1.301	1.358	1.282	1.355
171.swim	8%	19%	9%	36%	1.479	1.536	1.425	1.487
172.mgrid	3%	42%	3%	48%	1.061	1.219	1.04	1.22
173.applu	11%	9%	22%	41%	1.689	1.715	1.625	1.654
178.galgel	45%	3%	5%	53%	2.311	2.375	2.312	2.374
179.art	9%	2%	0%	11%	4.196	4.177	4.19	4.177
187.facerec	6%	10%	4%	19%	0.975	0.99	0.969	0.991
188.ammp	7%	5%	7%	19%	1.689	1.665	1.701	1.66
189.lucas	3%	13%	5%	22%	1.568	1.639	1.559	1.638
301.apsi	6%	15%	12%	33%	1.823	1.796	1.605	1.607
					Average Performance Improvement of CMA over FMA (weighted by % of FP instructions)			
					3.97%		**4.62%**	

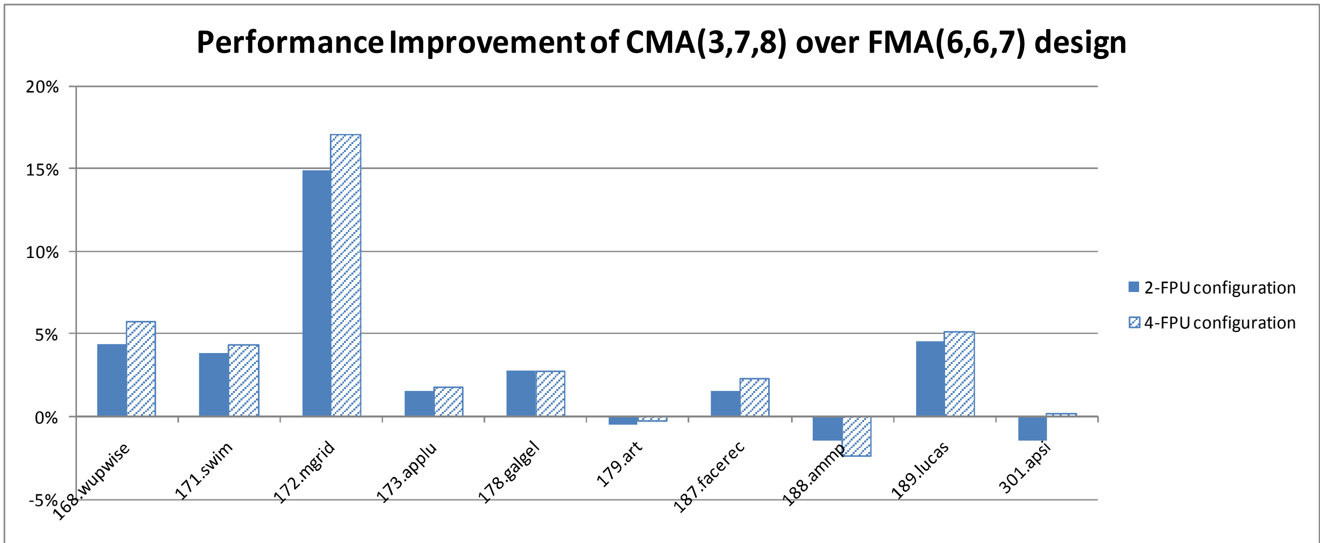


Figure 4: CPI reduction in CFP 2000 benchmarks for out of order 2-FPU and 4-FPU machines.

wide spectrum of designs ranging from simple in-order single issue designs to out of order superscalar designs. Having demonstrated the potential advantage of a cascade design, the following sections review the design of FMA and explain the details of the proposed CMA implementation.

III. FUSED MULTIPLY ADD DESIGN

Fused Multiply Add units have been extensively studied and implemented in several designs [6][7]. They reduce latency by performing alignment of the addend significand (S_B) in parallel with the multiplication tree of S_A and S_C. Furthermore, the multiplier output is kept in carry save format and added to the aligned addend, thereby saving one extra add operation. However, since the addend's exponent might be smaller or larger than the sum of multiplicands' exponents, the addend significand can be shifted from all the way to the left of the multiplier result to all the way to the right, requiring the datapath to have wide shifters and adders.

Several modifications to improve latency have been proposed. Parallel path designs that compute different datapaths in parallel and select the correct answer based on different cases have been proposed, but seem to have large area overhead [8]. Some FMA designs also aim to improve the accumulation latency as well. Intel demonstrated an 80-core throughput chip that employed an 11-stage multiply-

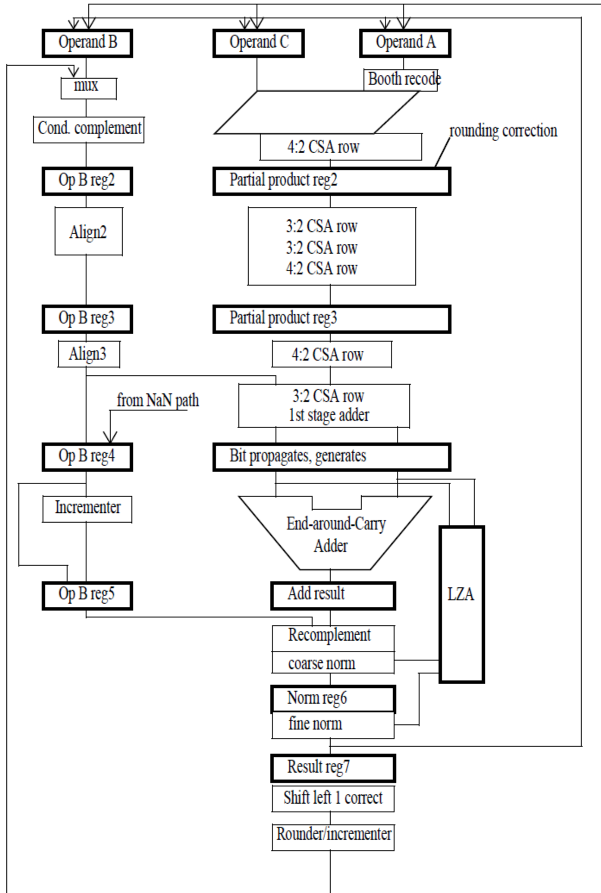


Figure 5: Power6 FMA Significand Datapath (reproduced from [3])

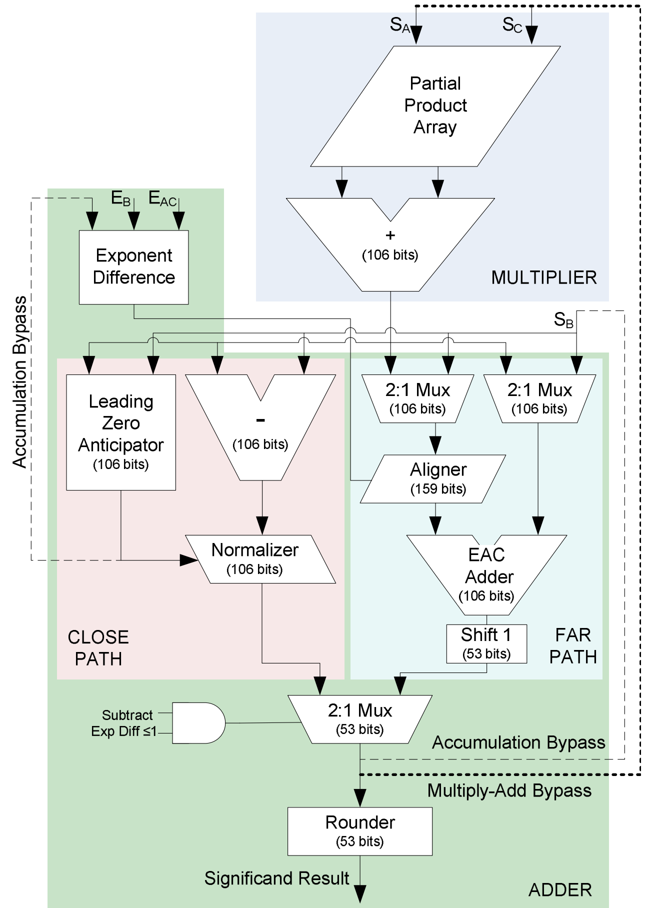


Figure 6: Simplified CMA significand datapath (multiplier; adder: far path, close path) with accumulation bypass path shown as dashed line and multiply-add bypass path shown as dotted line.

accumulate unit with single cycle accumulation latency [9][10]. Unfortunately, this design is not an IEEE FMA operation, because it does not preserve intermediate precision. A Bridge FMA design has been proposed to add FMA functionality by adding a bridge unit to slightly modified adder and multiplier designs [12]. The area of this bridge FMA unit is nearly as large as a separate FMA and adder units, which makes this approach less appealing.

Another FMA design tries to improve the latency of additions by separating addition cases into two groups. One, where the exponents are far apart, does not require normalization, and the alignment is done after multiplication. The other, where the exponents are close, skips the shifter, which gives time for post addition normalization [11]. We use this idea in our cascade designs. That design also keeps the multiplier output in carry save format, which results in complicating the addition datapath. We use this idea in our (4,6,7) CMA2 design variant to improve the overall latency, but it does add energy and the accumulation latency is degraded because of the extra carry save adder and wider datapath required. Figure 7 illustrates the datapath of the significand of the CMA2 design.

The Power6 FMA is a recent IEEE-compliant 7 cycle 13 FO4 design with a 6 cycle latency for dependent instructions (Figure 5). It achieves the reduced dependency latency by forwarding the unrounded results with special control signals to indicate if the result is to be incremented. Special terms added in the multiplier tree are used to generate the correct product. For example, if A is forwarded and Increment signal is asserted, an additional A term is added in the multiplication tree to produce $A\times C+A = (A+1)\times C$. Such a design has (6,6,7) latency by the metrics introduced earlier. This FMA design is used as the standard design for comparison because it is IEEE-compliant and has the shortest latency of FMA architecture for the least area and energy.

IV. Cascade Multiply Add Design

One can compute a multiply add by simply cascading the two functional elements. However because of the requirement of unlimited precision for intermediate results of FMA instructions, the multiplier and adder are different from traditional floating point adders/multipliers. For example, a double precision CMA design contains the following stages

- A multiplier that takes 2 double-precision operands A,C to generate the result A×C in "quad" precision (106 bit mantissa, 13 bit exponent)
- An asymmetric adder that takes a double precision operand B and the "quad" precision multiplier output to produce a double-precision result (A×C)+B

Thus, CMA is just an implementation variant of FMA that produces exactly the same result for FMA instructions with unlimited intermediate precision and rounding only once at the end. The cascade design is a less common implementation but still has been used in some designs [13]. We evaluated the CMA design since the add portion can be optimized to be very fast using parallel paths algorithms where either alignment or normalization steps are saved [14] which might make up for the slight increase in overall latency. The overall latency increases because the multiplier tree outputs are combined using an adder before being fed to the cascaded adder. Since the add operations start "late" in the overall pipeline, they cause less stall time than would occur in a normal FMA.

Figure 6 illustrates the datapath of the significand of the CMA design we have developed. It employs an adder with far path datapath for calculating the sum or difference when the exponent difference is greater than 1 and a close path datapath that calculates the difference when the exponent difference is ≤ 1, which is the only case where there could be massive cancellation and a need for a big normalizing shifter. The design has been optimized to shorten accumulation latency and handle forwarding of unrounded results (with increment signals) to shave a cycle off the accumulation and multiply-add latencies as was done in the FMA design. The next two sections discuss the details of these optimizations.

A. Removing the Rounding Latency Overhead

To reduce the overall latency of dependent instructions, our CMA design implements a bypass path for dependent instructions that feeds forward the unrounded result and an increment signal.

Implementing the bypass for the multiplier inputs A , C is similar to the design used by IBM [3]. We modify the multiplier tree to have one extra term that can be either S_A if Inc_A signal is asserted, or S_C if Inc_C is asserted. As for the input B, the adder part has been modified to accept the inputs S_B, Inc_B and $S_{A\times C}$. The idea is to merge the incrementation of B with the addition to A×C using carry save adders. The implementation of the near path and far path adders that support the increment signal is done as follows:

1) Close Path: The close path handles the subtraction case of $S_{A\times C}$ (106 bits) and S_B (53 bits) which are aligned on the MSB. The absolute difference of two binary numbers x, y is usually calculated as follows:

$$abs(x-y) = \begin{cases} x - y = x + \bar{y} + 1 & , y < x \\ -(x - y - 1) - 1 = \overline{x + \bar{y}} & , y \geq x \end{cases}$$

Therefore, the operation can be implemented using a compound adder to produce $(x + \bar{y})$ and $(x + \bar{y} + 1)$, and a

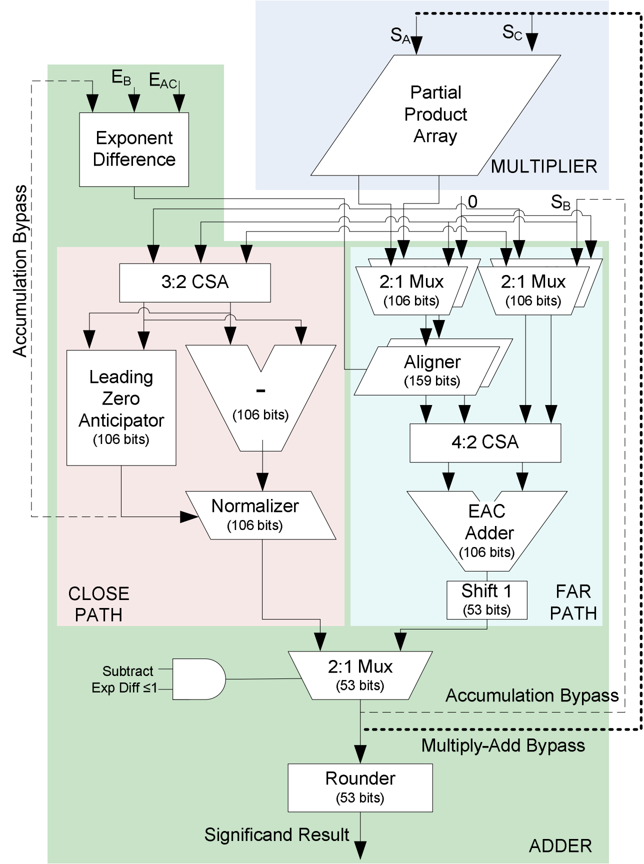


Figure 7: Simplified CMA2 significand datapath (multiplier; adder: far path, close path) with accumulation bypass path shown as dashed line and multiply-add bypass path shown as dotted line. Notice the duplicated Aligner and Mux units and extra CSA adders in the adder datapath

mux to choose between $\overline{(x + \bar{y})}$ and $(x + \bar{y} + 1)$ if there is a carry out from $(x + \bar{y})$.

Additionally, S_B needs to be incremented before the absolute difference operation if Inc_B is asserted. It is straightforward to merge the incrementation of S_B with the absolute difference operation by right padding S_B with 53 bits of Inc_B (to match the width of the multiplier output), which makes adding 1 at the least significant position produce the resulting effect of incrementing S_B.

2) Far Path: The far path handles addition and subtraction when the exponent difference is greater than 1 (Figure 8). The addend with the bigger exponent (S_{big}) can be as wide as 106 bits. The addend with the smaller exponent (S_{small}) is shifted right by the amount of exponent difference and becomes 159 bits wide after shifting. In case of subtraction, S_{small} is inverted before being fed to the adders. A compound adder of 106 bits summing S_{big} and S_{small}[158:53] that produces *sum* and *sum*+1 is sufficient for calculating the sum and difference [15]. Finally, only the uppermost 53 bits of the result is retained after normalization (possible right shift in case of addition and

left shift in case of subtraction) and guard and sticky bits are calculated. To support incrementation of S_B, the design is modified by having an adder that produces *sum*, *sum+1*, and *sum+2*. Choosing between the three results gives the equivalent result of incrementing S_B before the add operation. The correct result is chosen according to the following rules:

When $Exp_B > Exp_{A\times C}$ (Figure 8(a)): S_B is right padded with Inc_B. and:

$$S_{big} = \{S_B, \{(53)\{Inc_B\}\}$$

$$S_{small} = \{S_{A\times C}, 53'b0\} \gg (Exp_B - Exp_{A\times C})$$

If Inc_B is asserted, the result of addition becomes *sum+1*, while the result of subtraction becomes *sum+2*.

When $Exp_{A\times C} > Exp_B$ (Figure 8(b)): S_B is the smaller fraction, and in case of incrementation, we need to add 1 to the LSB of S_B which is then fed to the alignment shifter. To combine the incrementation with alignment and add operation we pad the lower bits with Inc_B so that after shifting, adding 1 to the LSB is still equivalent to incrementing S_B before shifting. Logically for S_{small} we will create a 159 operand to feed into the adder, and we will add the carry at the LSB. So

$$S_{big} = S_{AC}$$

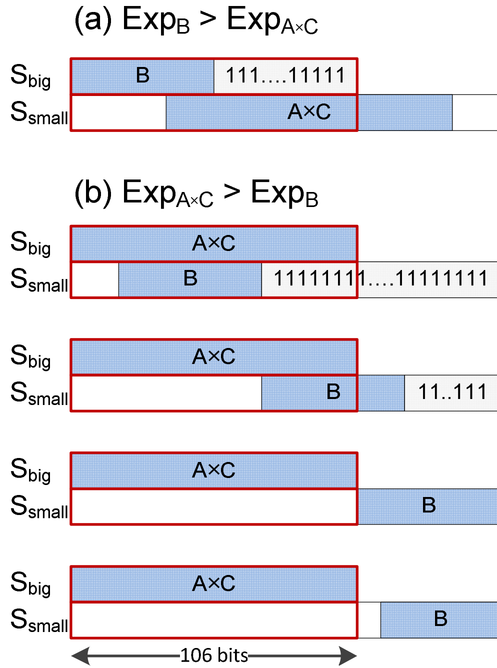


Figure 8: Far Path addition of mantissa of B and A×C with Inc_B asserted. The boxes indicate the portion of the fractions that are fed to the adder. The padded ones and 1 added at the least significant bit produce the equivalent of increment of B. In case $Exp_{A\times C}$> Exp_B: Carry in to the 106 bit adder is carry in to the effective 159 bit adder ANDed with the 53 LSBs, since they all need to be '1' for the carry to propagate to the upper 106 bits.

$$S_{small} = \{S_B, (106)\{Inc_B\}\} \gg (Exp_{A\times C} - Exp_B)$$

Since S_{big} is zero for the 53 LSBs, carry-in to the 106 bit adder is generated by carry-in ANDed with the lower 53 bits of S_{small} which is used to choose between *sum* and *sum+1* in the case of addition. This handles all the shift cases. As for subtraction, S_{small} is inverted before being fed to the adder. Since $\overline{S_{small}} = -(S_{small} + 1)$, then the result of subtraction is always *sum* if Inc_B is asserted.

B. Optimizing the Accumulation Loop

The accumulation loop can be reduced by noticing that the result exponent is known to within ±1 in advance of the result mantissa in carry save format as an output of the adder. In the near path, the exponent is the difference between the larger exponent and the leading zero anticipator (LZA)

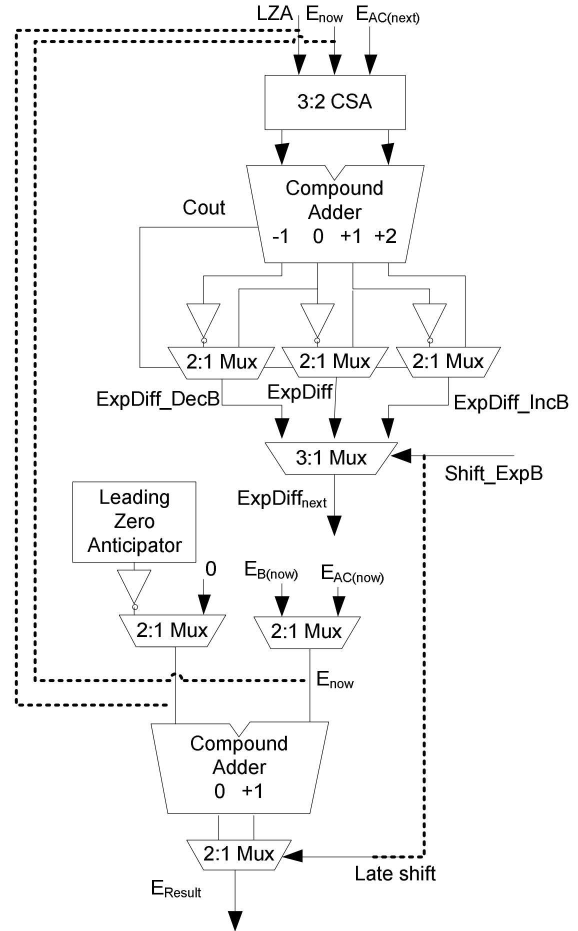


Figure 9: Simplified exponent datapath indicating the feedback loops. Since we don't know the output of the final normalization (*Shift_ExpB*) we take the output of the current operation (E_{now}) and the output of the LZA and combine them with the next multiplier output ($E_{AC(next)}$) to compute the next exponent difference ($ExpDiff_{next}$). Since $E_{now} + LZA$ can be off by one, we need to compute both options, and we need to compute the absolute value of the result (the 2-1 mux driven by *Cout*)

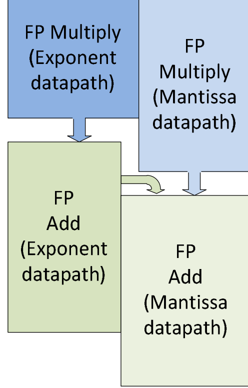


Figure 10: Block diagram of CMA mantissa and exponent datapaths showing the staggered timing of the exponent and mantissa

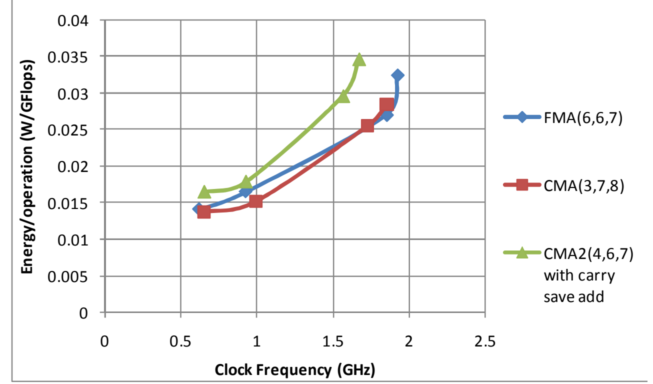


Figure 11: Energy efficiency tradeoff curves of different fused multiply-add architectures.

count. In the far path, the exponent is just the bigger exponent of the two addends, but might be incremented if a late right shift is needed in case of addition or decremented if a late left shift is needed in case of subtraction.

Figure 9 illustrates the exponent datapath implementation to achieve reduced accumulation latency. An exponent difference unit takes as input E_{now}, LZA, and $E_{AC(next)}$. It computes:
$abs\left(E_{now} + LZA - E_{AC(next)} + x\right)$, where x = -1, 0, 1. corresponding to the exponent difference if the last result is normalized to the left, not shifted or normalized to the right. A late select based on normalization of the mantissa is used to select the correct exponent difference for next stage.

The mantissa datapath is architected to start operation after the exponent difference is found, resulting in overlapping bypass loops of the exponent datapath and mantissa datapath, as shown in Figure 10. This late mantissa datapath design has several advantages. First, the exponent difference is done in parallel with the multiplication, removing the exponent difference stage from the critical path between the multiplier and adder; thereby shortening the total latency of CMA design and making it roughly the same as FMA one. Second, the critical path for an accumulation dependent instruction is improved from 4 cycles to 3 cycles without noticeably affecting the latency of independent instructions. Finally, since exponent difference is performed first, power optimizations such as fine-grained clock gating of the far/near path of the adder based on exponent difference can be introduced, although no such optimization was implemented in the presented power figures.

TABLE 2
UNPIPELINED FMA VS. CMA DESIGN LATENCIES

	FMA	CMA	CMA2 with CSA
Accumulation Latency (ns)	2.14	1.03	1.29
Multiply-Add Latency (ns)	2.14	2.4	2.28
Average Latency (ns)	2.14	1.715	1.785
Area (μm²)	33149	36660	41429
Energy/op (pJ)	17.9	19.3	21.864

C. Handling Denormals

The cascade design handles denormals at the adder input without a problem. It also produces denormal numbers correctly when they are the result of the operation. However multiplying a denormal with a normal number can result in an unnormalized input to the adder which could generate cancellations in the far path and therefore require normalization. To solve for such a problem a special exception is raised and a normalization stage after the multiplier is used to normalize the product.

V. Timing, Power and Area

An FMA, a CMA design and a CMA2 with multiplier outputs in carry save format have been implemented and verified using SystemVerilog and synthesized using TSMC 45nm technology libraries. To determine the relative latencies, unpipelined versions of the designs were synthesized. Table 1 summarizes the result. CMA has the least accumulation latency while FMA has the least multiply-add latency. These latencies were the basis for choosing the latency cycles we evaluated in our application study.

For comparing the delay and energy of the designs, the FMA design and the CMA2 were synthesized using a 7-stage pipeline while for the CMA design an 8-stage pipeline was synthesized. The datapath optimization flow starts by synthesizing a design for a certain timing constraint, inserting pipeline registers and doing register retiming to pipeline the design. Then the resulting design is placed and routed and the required clock network is generated. After the design is routed, the design is reoptimized and parasitics are extracted and annotated to the netlist. Activity factors for dynamic power calculations are calculated for random input vectors and assuming full utilization of the FPU. The timing and power of the design are then reported using Primetime timing tool. This procedure is repeated over a wide range of supply voltages, threshold voltages, and clock periods to

TABLE 3
EFFICIENT FRONTIER DESIGNS (ENERGY/OP VS. FREQUENCY) FOR DIFFERENT FMA ARCHITECTURES

V_{dd}	V_{th}	Frequency (GHz)	Area (μm²)	Power(mW)		FO4(ns)	Cycle Time (FO4)	mm²/ GFlops	W/ GFlops
				Dynamic	Leakage				
Double Precision FMA(6,6,7)									
0.72	standard	0.62	47269	17.9	0.9	24	67	0.038	0.014
0.81	low	0.93	43651	30.5	2.3	17	64	0.024	0.016
0.9	low	1.85	52240	103.6	4	14	39	0.014	0.027
0.9	low	1.92	71089	204	7	14	37	0.018	0.032
Double Precision CMA(3,7,8)									
0.72	standard	0.65	49571	17.4	0.9	24	64	0.038	0.014
0.81	low	0.99	44950	28	2.2	17	59	0.023	0.015
0.9	low	1.72	54578	96.7	4.7	14	41	0.016	0.026
0.9	low	1.85	61133	134	6.1	14	39	0.017	0.028
Double Precision CMA2(3,6,7) with multiplier outputs in Carry Save format									
0.72	standard	0.65	58990	20.9	1.1	24	64	0.045	0.016
0.81	low	0.93	52357	32.9	2.7	17	64	0.028	0.018
0.9	low	1.56	63530	110.2	6.4	14	46	0.02	0.03
0.9	low	1.67	81944	64.3	5	14	43	0.025	0.035

choose the most energy efficient designs. After generating the data, the points on the efficient frontier of minimum energy/op designs for a certain performance targets are extracted from data points and are plotted in Figure 11. Table 2 provides the power, area and design parameters of these efficient frontiers. Examining the data, FMA(6,6,7) and CMA(3,7,8) have very similar energy and area cost, while CMA2(4,6,7) requires roughly 20% more energy and area.

VI. CONCLUSION

When optimizing an FMA design, it is critical to understand that the effective latency of the operation depends on which unit (multiplier or adder) will consume the output, and whether latency matters at all. For applications with abundant parallelism, the latency penalty will be zero and throughput oriented metrics such as W/GFlops and mm²/GFlops should be the optimization target. For more latency sensitive applications, a cascade design provides a number of parameters that can be optimized, and in particular it allows one to create a design with very low effective latency between operations with a sum dependence. The reduction in latency depends on two main optimizations: forwarding of unrounded results and tightening the accumulation bypass path by staggering the exponent and mantissa datapath of the adder. Building and synthesizing the design reveals it does not incur area or energy overheads over existing FMA designs. Using an architectural simulator and SPEC2000 FP benchmark we found the CMA design to have 6% performance gain for a simple single issue in-order designs and 4-4.5% gain for out of order superscalar designs.

REFERENCES

[1] Hokenek, E.; Montoye, R.K and Cook, P.W, “Second-generation RISC floating point with multiply-add fused”, IEEE Journal of Solid-State Circuits, Oct. 1990, pp. 1207–1213.

[2] Standard Performance Evaluation Corporation, “CFP2000 (Floating Point Component of SPEC CPU2000)”, http://www.spec.org/cpu2000/CFP2000/

[3] Son Dao Trong, Martin S. Schmookler, Eric M. Schwarz, Michael Kroener, “P6 Binary Floating-Point Unit”, IEEE Symposium on Computer Arithmetic 2007: 77-86

[4] S. Galal, M. Horowitz, “Energy-Efficient Floating Point Unit Design,” IEEE Transactions on Computers, Vol. pp, issue 99, June 2010

[5] N. L. Binkert, et al., “The M5 Simulator: Modeling Networked Systems” IEEE Micro, Vol. 26, No. 4. (2006), pp. 52-60.

[6] R.M. Jessani and M. Putrino, “Comparison of Single- and Dual-Pass Multiply-Add Fused Floating-Point Units,” IEEE Transactions on Computers, Vol. 47, pp. 927-937. 1998.

[7] H.-J. Oil, , et. al., “A Fully Pipelined Single-Precision Floating-Point Unit in the Synergistic Processor Element of a Cell Processor”, IEEE J. Solid-State Circuits 41, No. 4, 759-771 (2006)

[8] P.-M. Seidel “Multiple Path IEEE Floating-Point Fused Multiply-Add”, Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems, 2003. MWSCAS '03.

[9] S. R. Vangal, et al., “An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS” IEEE J. Solid-State Circuits, pp. 29-41, Vol. 43, Jan. 2008

[10] S. R. Vangal, Y. V. Hoskote, N. Y. Borkar and A. Alvandpour, “A 6.2-GFLOPS floating-Point multiply-accumulator with conditional normalization,” IEEE J. Solid-State Circuits, pp. 2314-2323, Vol. 41, Oct. 2006

[11] J.D. Bruguera and T. Lang, “Floating-Point Fused Multiply-Add: Reduced Latency for Floating-Point Addition,” Proceedings of the

17th IEEE Symposium on Computer Arithmetic. pp. 42-51, June, 2005.

[12] E. Quinnell, E. Swartzlander , and C. Lemonds, "Bridge Floating-Point Fused Multiply-Add Design," IEEE Transactions on VLSI Systems, Vol. 16, No. 12. (December 2008), pp. 1727-1731

[13] N. Ide, et al., "2.44-GFLOPS 300-MHz Floating-Point Vector-Processing Unit for High-Performance 3-D Computer Graphics Computing," IEEE J. Solid-State Circuits, vol. 35, no. 7, pp. 1025-1033, July 2000.

[14] P. M. Farmwald, "On the design of high performance digital arithmetic units," PhD Thesis, Stanford University, Stanford, CA, 1981

[15] E.M. Schwarz, "Binary Floating-Point Unit Design: the fused multiply-add dataflow", High-Performance Energy-Efficient Microprocessor, Edited by V.G. Oklobdzija and R.K. Krishnamurthy, Springer, Chapter 8, 2006.