Low-Cost Microarchitectural Support for Improved Floating-Point Accuracy

William R. Dieter dieter@engr.uky.edu
Akil Kaveti Akil.Kaveti@uky.edu
Henry G. Dietz hankd@engr.uky.edu
Electrical and Computer Engineering Dept., University of Kentucky

***Abstract*—Some processors designed for consumer applications, such as Graphics Processing Units (GPUs) and the CELL processor, promise outstanding floating-point performance for scientific applications at commodity prices. However, IEEE single precision is the most precise floating-point data type these processors directly support in hardware. Pairs of native floating-point numbers can be used to represent a base result and a residual term to increase accuracy, but the resulting order of magnitude slowdown dramatically reduces the price/performance advantage of these systems.**

By adding a few simple microarchitectural features, acceptable accuracy can be obtained with relatively little performance penalty. To reduce the cost of native-pair arithmetic, a residual register is used to hold information that would normally have been discarded after each floating-point computation. The residual register dramatically simplifies the code, providing both lower latency and better instruction-level parallelism.

I. Introduction

Many problems solved on computers inherently involve real numbers, but computer arithmetic in modern processors is limited to integer and floating-point arithmetic. With integer arithmetic, the algorithm must be meticulously analyzed to choose scaling factors that avoid underflow or overflow. Floating-point arithmetic operates on numbers stored as a sign, exponent, and magnitude, essentially adjusting the scaling factors automatically. An IEEE 32-bit (single precision) floating-point number has a wide enough range and large enough mantissa to accurately represent most physical measurements. However, long sequences of floating-point computations found in many scientific and engineering application programs can introduce roundoff errors, cancellations, and other problems that lead to loss of accuracy when single precision is used [7]. Thus, 64-bit (double precision) and sometimes higher precision floating-point is commonly used for scientific computing.

Unfortunately, implementing a 64-bit floating-point pipeline requires significantly more hardware than a 32-bit pipeline. Commodity multimedia processors, like Graphics Processing Units (GPUs) or the Cell [9] are designed for tasks like rendering 3D scenes. For that purpose, the 16-bit floating point found in some GPUs often is sufficient, but 32-bit floating point is provided for highly complex models. Increasingly, researchers are working toward using GPUs to accelerate non-graphical applications that require greater accuracy, but the demand is not large enough to justify the extra circuit complexity required to support 64-bit floating point. More graphics applications will profit from additional low-precision pipelines, which is what ATI and nVidia build.

The residual register architecture discussed in this paper provides a low hardware cost way to reduce the performance penalty of using multiple 32-bit floating-point values to extend the accuracy of intermediate computations when needed. Of course, it also can be applied to enhance the accuracy of any other native precision. Our previous work in this area focused on efficient native-pair algorithms and data layouts for GPUs, software speculation algorithms to avoid the performance cost of more accuracy than necessary, and software implementations of the algorithms optimized for a GPU [6].

Higher than native floating-point precision can be obtained with either integer arithmetic or native floating-point hardware. Embedded processors without floating-point units (FPUs) implement IEEE floating-point entirely in software using integer arithmetic, and the Gnu Bignum Library [1] implements arbitrary precision floating point using native integer operations. An alternative, which is the focus of this paper, is to use multiple floating-point numbers to represent error residuals. Each higher-accuracy value is spread across the mantissas of a sequence of native floating-point values in which the exponents in the lower components serve to align the mantissas. Sequences of numbers can be used to obtain arbitrary precision [13], groups of four numbers can be used to roughly quadruple the native precision [8], or pairs of numbers can approximately double precision, commonly referred to as *double-double* when used with double precision numbers [3], [4], [11]. We refer to a generalized notion of double-double as *native-pair* when the native precision might not be double.

The primary problem with native-pair arithmetic is that it usually takes ten or more native operations for each native-pair operation. The only mechanism for higher accuracy in the IEEE 754 standard [10], without moving to a higher precision, is the "fused multiply-add," or `MADD` instruction. `MADD` adds the $2k$-bit product of two k-bit numbers to a $2k$-bit accumulator. If the high bits of the product and the accumulator cancel, bits from the normally discarded low portion of the product can be retained. Unfortunately, `MADD` instructions often are *not* implemented as true fused operations, so extra bits are not used in the addition and accuracy is not improved.

II. Microarchitectural Support For Native-Pair

Much of the overhead of native-pair arithmetic comes from computing the residual, or error term, resulting from native

Manuscript submitted: February 22, 2007; Manuscript accepted: March 15, 2007; Final manuscript received: March 27, 2007.

floating-point arithmetic operations. For addition, subtraction, and multiplication part or all of the residual term is discarded by the FPU. We propose adding a *residual register* to save these discarded bits. The residual register is a floating-point register with a sign bit, n_e exponent bits, n_m+2 mantissa bits, and a complement flag bit, where n_e and n_m are the number of exponent and mantissa bits in a native floating-point number, respectively, not including the leading one bit in the mantissa implied by the IEEE 754 format [10]. Programs that do not use the residual register will get the usual result defined by the IEEE standard. Results stored in the residual register can be used to speed up extended-precision floating-point algorithms by replacing sequences of instructions that compute equivalent results with a single residual register access.

To simplify the hardware, the residual register stores unnormalized results. The residual register is normalized using the normalization unit already present for floating-point add or multiply, as it is moved to an architectural register. One way to implement this operation with minimal impact on the instruction set architecture (ISA) is to add a ”`MOVRR` *`reg`*” instruction that normalizes the residual register and stores it in an architectural register. In this case, we assume each floating-point operation overwrites the previous value of the residual register with the current residual value.

In out-of-order or superscalar processors a single physical residual register will become a structural hazard when multiple instructions produce residual results. To remove this hazard and allow the compiler more flexible scheduling, residuals from the last q operations can be stored in a logical queue, and the k^{th} most recent residual accessed with a ”`MOVRR` *`reg, k`*” instruction. In a processor using register renaming, residuals can be assigned to physical floating-point registers using logic similar to that used for assigning primary results to physical registers. A physical register assigned a residual can be freed as soon as $q+1$ subsequent operations not accessing that register have been issued. Thus, code not using the residual register would have the same number of physical registers available if only q new physical registers are added.

Alternatively, if the residual register(s) are made accessible as operand sources for the basic floating-point operations, there would be no need for `MOVRR` instructions. The ISA would have to be modified either to reserve existing register names or to change the instruction encoding to specify a floating-point register in which to store the residual. This approach requires more hardware to obtain the maximum benefit, however. An additional normalization unit dedicated to the residual results would be needed to make residuals immediately available as operands to later instructions.

In the following discussion, the sign, exponent, and mantissa of the floating-point number x are denoted as $sign(x)$, $exp(x)$, and $mant(x)$, respectively. The primary result of a floating-point operation is denoted $fl(x \circ y)$, and the residual is $res(x \circ y)$. Where $\circ$ may be ‘$+$’, ‘$-$’, or ‘$\times$’. In the IEEE 754 round-to-nearest-even mode, the residual register can guarantee $x \circ y = fl(x \circ y) + res(x \circ y)$, because $x \circ y$ is never more than $1/2$ unit in the last place from $fl(x \circ y)$. Other IEEE 754 rounding modes allow larger errors, and therefore residuals cannot always be represented precisely.

A. Native-Pair Addition and Subtraction

When two floating-point numbers a and b are added, the addend with the smaller magnitude is shifted to align its radix point with the radix point of the larger-magnitude addend. If $|b|$ is larger than $|a|$, a and b can be swapped, so we assume $|a| > |b|$ to simplify the discussion.

Figure 1 shows a high-level schematic example of a floating-point adder with a residual register. The logic that performs the functions inside the dashed lines is representative of logic typically present in a floating-point adder. Whether the adder is structured much like the one shown here, is a two-path design, or some other design, we assume the residual register circuitry is able to use these signals. The logic inside the area labeled “Residual Register” is added to the basic floating-point adder to support the residual register.

The adder has three logical stages: Pre-normalization, Addition, and Post-normalization, though the actual implementation may have more or fewer pipeline stages. Pre-normalization shifts the mantissa of the b to align it with the mantissa of a. The mantissa bits in b with significance less than $2^{exp(a)-(n_m+1)}$ are stored in the residual register with the least significant bit always in the rightmost position, with `SMASK` masking off any bits in b that were added to a. The exponent is set to $exp(b)$ in pre-normalization. After rounding occurs during addition, the complement flag, c, is set if the primary result was rounded to a higher magnitude, and the residual would have the same sign as the primary result. The logic for setting c can be expressed as $c = rnd \oplus sign(a) \oplus sign(b)$, where rnd is true if and only if rounding occurred regardless of the current rounding mode.

When the residual register is moved to an architectural register, the residual mantissa is complemented if the complement flag is set, and the alignment unit already present in the post-normalization stage aligns the residual mantissa. The residual sign is set to $rnd \oplus sign(a)$, and the exponent is computed from the $exp(b)$, $exp(a) - exp(b)$, and the complement flag. More details about how the residual register and complement flag are set and how to update the native-pair software algorithms proposed by others are given in a recent technical report [5].

Subtraction is trivially different from addition. The sign bit of the subtrahend is toggled and the two numbers are added.

B. Native-Pair Multiplication

Setting the multiplication residual register is simpler than setting the addition residual register. Multiplication of two n bit numbers produces a result with up to $2n$ bits. The mantissa, $mant(rr)$ stores the low n bits of the product after a multiply, and $exp(rr)$ is set to $exp(p) - (n_m + 1)$ to align the residual mantissa with p. When the result is rounded to a smaller magnitude, $sign(rr)$ is set to $sign(p)$ and the complement flag is cleared. If p is rounded to a larger magnitude then $a \cdot b = p + r = p - 2^{exp(p)-n_m} + rr$, so $r = 2^{exp(p)-n_m} - rr$. That is, $sign(rr) = \overline{sign(p)}$ and the complement flag is set. A high-level schematic of a multiplier with a residual register is shown in Figure 2.

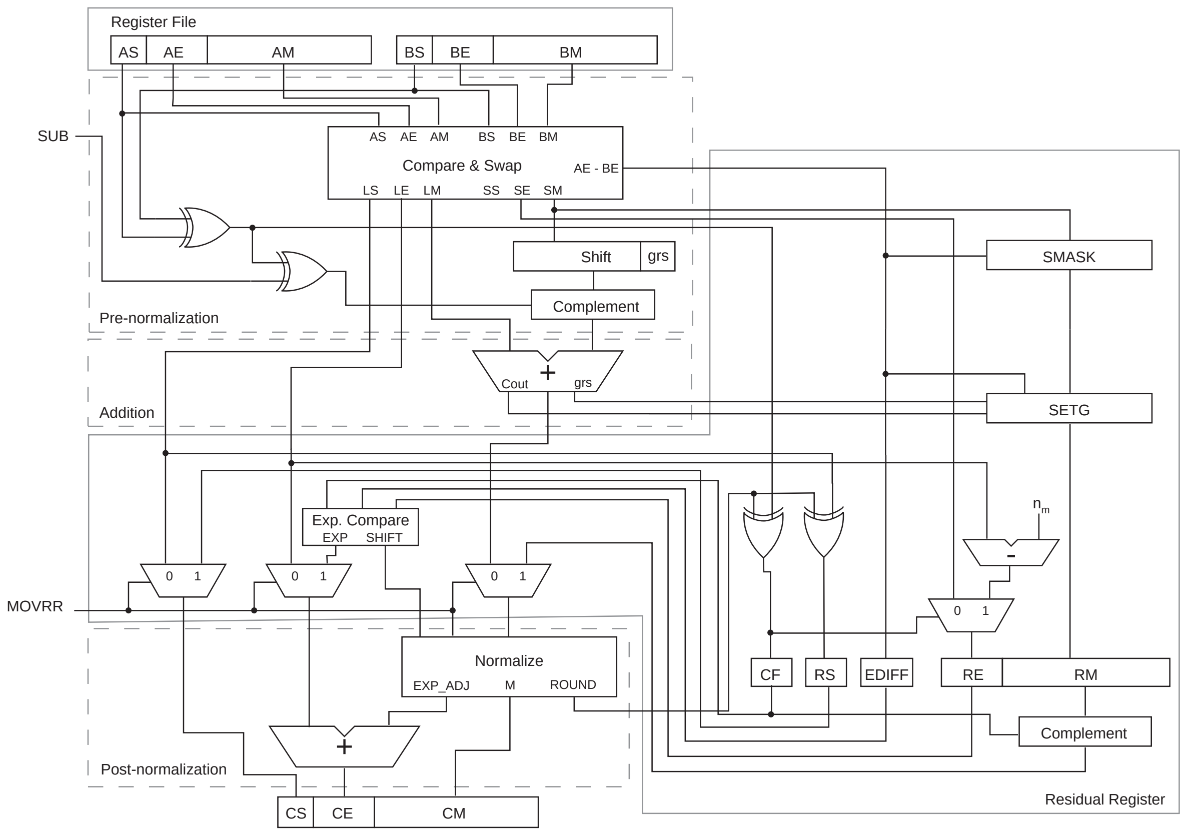

Figure 1. High-level schematic of a floating-point adder with a residual register

As with the addition residual register, this design adds multiplexers in the critical path. The delay added by the multiplexers may affect the clock cycle time. The implementation in Figure 2 assumes that all $2n$ product bits are available. Some multiplier architectures only compute the carries for low order bits in the product. For these multipliers, additional hardware is required to compute the bottom n bits of the product. Adding support for the low order bits adds complexity to the multiplier, but no more hardware is required than is needed to implement a fused `MADD` instruction.

C. Other Native-Pair Floating-Point Operations

Compared to addition, subtraction, and multiplication, floating-point divide and square root instructions typically have a high latency. Moreover, current divide and square root algorithms do not produce a directly usable residual. Though it may be possible to implement a residual register for divide and square root, the savings in execution time is not sufficient to justify the added circuit complexity. Even so, the software native-pair divide and square root operations both use native-pair multiplication [5] and get a modest speedup from the multiply residual register.

A fused multiply-add instruction uses an adder with the full $2n_m$-bits of precision in the product to minimize the loss of accuracy. If the hardware does not support fused multiply-add, the residual register can be used to compute the fused multiply-add in only a few instructions [5]. The residual register hardware is simpler than that for a fused multiply-add instruction because the multiply-add requires an adder twice as wide as the native floating-point size.

III. Residual Register Results

Both the add and multiply residual register algorithms have been tested with a C program simulating operations on pairs of numbers in the “Gaussian” and “Heavy Cancellation” pseudo-random sequences described by McNamee [12]. The results of the native-pair operations implemented using the software-only algorithm were compared with results using the simulated residual register. For each test sequence and operation, no errors were found in the simulation of one billion operations.

A VHDL model of the add and multiply residual registers and a FPU [2] were synthesized for a Xilinx Virtex4 FPGA to evaluate hardware complexity and speed of the design. Table I compares 32-bit FPUs without residual register, 32-bit FPUs with residual register, and 64-bit FPUs. All percentages are relative to the 32-bit design without a residual register. For both addition and multiplication, the increase in size and minimum clock period for the 32-bit residual register is much smaller than the increase for the 64-bit FPU. Even if the increased clock cycle time for the 64-bit FPUs is partially hidden by pipelining, 64-bit floating point operations will still have higher latency than their 32-bit counterparts.

The complexity cited in Table I assumes the 64-bit FPUs replace the 32-bit FPUs. In that case, all floating-point add and multiply operations have the higher latency of 64-bit operations, even though 32-bit floating-point may be sufficient in

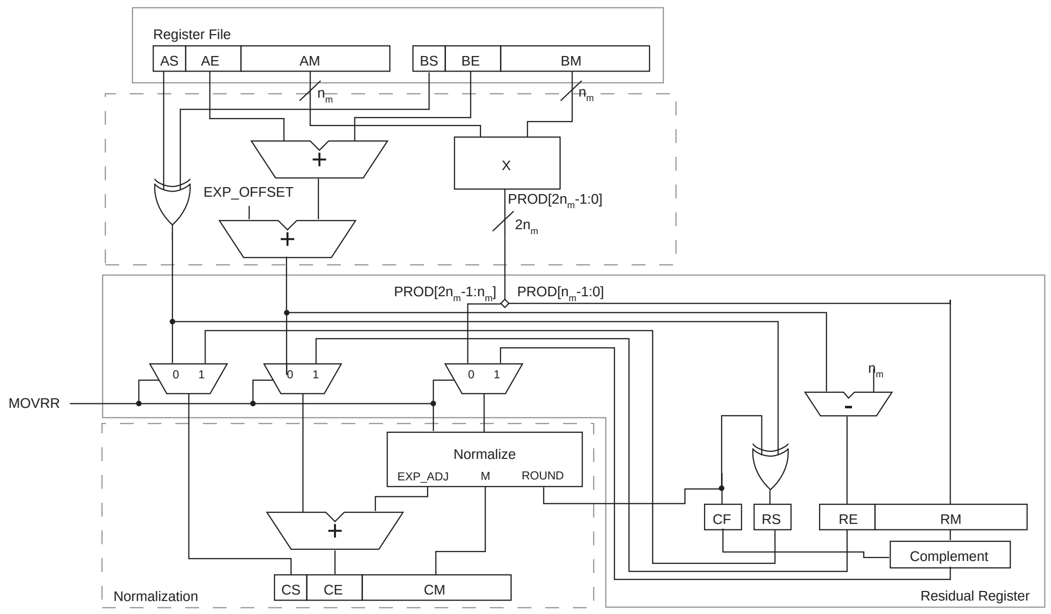

Figure 2. High-level schematic for the multiply residual register

Table I
COMPARISON OF IMPLEMENTATION COST AND DELAY OF ADDERS

	Add				Multiply			
	Implementation Cost		Minimum Period		Implementation Cost		Minimum Period	
	(slices)	% Increase	(ns)	% Increase	(slices)	% Increase	(ns)	% Increase
32-Bit without Residual Register	1461	0.0	18.2	0.0	2154	0.0	38.8	0.0
32-Bit with Residual Register	1620	10.9	18.9	3.8	2265	5.2	39.5	1.8
64-Bit without Residual Register	2272	55.4	59.4	226.4	6142	185.1	114.5	195.1

many cases. If the 32-bit FPUs are kept for higher performance on 32-bit operations, then the implementation cost increases to include both 32-bit and 64-bit FPUs.

IV. CONCLUSION

Although 32-bit floating-point hardware is now widely available at low cost, a significant number of applications require higher accuracy results than 32-bit intermediate calculations directly provide. Because the applications targeted by these processors do not need higher precision arithmetic, it is not economically justifiable to add 64-bit floating-point hardware support. Native-pair arithmetic can increase the accuracy of 32-bit floating point to be competitive with that of 64-bit floating point. However, native-pair arithmetic carries an order of magnitude performance penalty, mainly to compute residual terms using standard floating-point instructions.

Several low-cost microarchitectural changes reduce the overhead of computing these residuals for native-pair computations. The primary change is the augmentation of addition, subtraction, and multiplication hardware with residual registers: a modest hardware enhancement, changing the ISA only in that a new instruction is added to access the residual value.

REFERENCES

[1] The GNU MP bignum library. http://www.swox.com/gmp/.
[2] J. Al-Eryani. Fpu. OpenCores Arithmetic Core & Coprocessor, http://www.opencores.org/projects.cgi/web/fpu100/overview, January 2007.
[3] D. H. Bailey, Y. Hida, K. Jeyabalan, X. S. Li, and B. Thompson. Multiprecision software directory. *http://crd.lbl.gov/˜dhbailey/mpdist/*.
[4] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.
[5] W. R. Dieter and H. G. Dietz. Low-cost microarchitectural support for improved floating-point accuracy. Technical Report ECE-2006-10-14, University of Kentucky, ECE Dept., Lexington, KY 40506-0046, http://www.engr.uky.edu/˜dieter/pub/TR-ECE-2006-10-14, October 2006.
[6] H. G. Dietz, W. R. Dieter, R. Fisher, and K. Chang. Floating-point computation with just enough accuracy. *Lecture Notes in Computer Science*, 3991:226 – 233, Apr 2006.
[7] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
[8] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proc. IEEE Symp. on Comp. Arith.*, page 0155, 2001.
[9] O. Hwa-Joon, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. H. Dhong. A fully-pipelined single-precision floating point unit in the synergistic processor element of a cell processor. *Symposium on VLSI Circuits*, June 2005.
[10] IEEE. *IEEE Standard for Binary Floating Point Arithmetic Std 754-1985*, 1985.
[11] S. Linnainmaa. Software for doubled-precision floating-point computations. *ACM Trans. Math. Softw.*, 7(3):272–283, 1981.
[12] J. M. McNamee. A comparison of methods for accurate summation. *SIGSAM Bull.*, 38(1):1–7, 2004.
[13] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proc. IEEE Symp. on Comp. Arith.*, pages 132–143. IEEE Computer Society, June 1991.