Low-Cost Microarchitectural Support for Improved Floating-Point Accuracy

William R. Dieter `dieter@engr.uky.edu`
Henry G. Dietz `hankd@engr.uky.edu`
Electrical and Computer Engineering Dept.
University of Kentucky
Lexington, KY 40506

Abstract

Some of the potentially fastest processors that could be used for scientific computing do not have efficient floating-point hardware support for precisions higher than 32-bits. This is true of the CELL processor, all current commodity Graphics Processing Units (GPUs), various Digital Signal Processing (DSP) chips, etc. Acceptably high accuracy can be obtained without extra hardware by using pairs of native floating-point numbers to represent the base result and a residual error term, but an 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. To support speculative use of faster, lower precision, arithmetic, a peak exponent monitor and an absorption counter are added to measure potential loss of accuracy.

1 Introduction

Many of the problems that have motivated the development of computers inherently involve real numbers that must be approximated using finite representations. Fixed-size representations lead to the simplest and generally most efficient hardware implementations, so computer hardware has been focussed on providing support for operations on a small number of fixed-size floating-point representations. The problem addressed in this paper is that higher-precision representations often carry circuit complexity costs that make the support of higher precisions economically infeasible.

For example, Graphics Processing Units (GPUs) are designed to serve one purpose: real-time creation of pleasing 2D renderings of scenes involving 3D objects, textures, lighting, etc. For that purpose, 16-bit floating point often is sufficient, with only the most complex models requiring anything like the accuracy provided by 32-bit floating point. Although a growing number of researchers are working toward using GPUs to accelerate scientific and engineering applications that require greater accuracy, the number of GPU users requiring higher accuracy than 32-bit floating-point operations provide is insufficient to justify the extra circuit complexity required to support 64-bit floating point. One 64-bit pipeline has roughly the same implementation cost as two to four 32-bit pipelines. More graphics applications will profit from additional low-precision pipelines, and that is what ATI and nVidia build.

The irony is that even in the most sensitive floating-point applications, the precision of 64-bit floating point rarely is needed to represent the answer, but is merely used to ensure that the sequence of operations used in the computation produces an accurate 32-bit floating-point result. Accuracy is not the same as precision; higher precision does not necessarily imply higher accuracy, nor is higher precision the only way to achieve better accuracy. Thus, the residual register architecture discussed in this paper provides a minimally intrusive way to allow multiple 32-bit precision floating-point values to be used to efficiently extend the accuracy of intermediate computations. Of course, this approach also can be applied to similarly enhance the accuracy of any other native precision.

1.1 Floating-Point Hardware Approaches

Floating-point hardware is not a new thing; by 1941, Konrad Zuse's Z3 was executing 22-bit floating-point operations using a sign bit, 7-bit exponent, and 14-bit mantissa[1]. However, the long history of floating-point hardware support is plagued by unavoidable tradeoffs between efficiency and accuracy. Accuracy traditionally has been associated with the precision of the mantissas on which floating-point hardware operates, though our work does not make this assumption.

Hardware efficiently supporting a wide range of floating-point precisions using "bit slice" implementations of operations on a variable-length mantissa has never been common, but appears in a variety of systems. From the late 1970s, the NorthStar FPB-A[2] was an S100 bus floating-point coprocessor board using standard TTL parts to implement microcoded Binary Coded Decimal (BCD) floating-point add, subtract, multiply, and divide with 2, 4, 6, 8, 10, 12, or 14 digit precision and a 7-bit base-10 exponent. In the early 1990s, each of the 16,384 SIMD processing elements of the MasPar MP1 supercomputer[3] implemented floating-point operations using 4-bit slice operations on the mantissa and specialized normalization hardware; although standard microcode provided only 32-bit and 64-bit IEEE 754 formats, the hardware could efficiently support any multiple of 4 bits precision up to 64 bits. The primary problem with sequential slicing approaches is that they inherently yield latency which is a function of precision.

Many computers have processor hardware designed to implement just one precision of floating-point value which is considered to be sufficient for the primary applications targeted. This is particularly common in processors intended to work with "natural" data types associated with audio and video. For such types, the accuracy of commonly-used analog converters tends to drive the choice of mantissa size; 8-bit converters naturally led to 16-bit floating point and the proliferation of 16-bit and 20-bit converters has made 32-bit floating point particularly appealing. This evolutionary pattern has been followed by Digital Signal Processors (DSPs), SIMD Within A Register (SWAR) [4] instruction set extensions for conventional processors, and by Graphics Processing Units (GPUs) in high-end video cards. For example, 32-bit floating point is featured in many of the popular TI TMS320C3x series DSPs[5]. SWAR extensions such as AMD's 3DNow![6], Intel's SSE[7], and Motorola's AltiVec[8] all initially provided only operations on datapath-sized vectors of 32-bit floating-point values (although these extensions to general-purpose processors have more recently taken advantage of the wider datapaths to also support 64-bit floating-point operations). At this writing, both nVidia and ATI are marketing high-end GPUs focussed on 32-bit floating-point support, as is the CELL processor[9].

Many mainframe and workstation processor makers have taken advantage of the fact that the IEEE 754[10] standard allows operations to be performed using any equal or higher precision. The floating-point hardware pipeline(s) have been designed to use just one precision. Results are stored in the desired equal or lower precision with little additional hardware complexity. Perhaps the best known example is the Intel X87 floating-point model used in processors from Intel, AMD, Cyrix, etc. The X87 floating-point mechanism allows 32-bit, 64-bit, or 80-bit operands and results, but is essentially an 80-bit pipeline with a small amount of additional hardware allowing the pipeline to run at either of the two reduced precisions. Without changing the pipeline precision, conversions are applied at register load/store, with the strange result that the accuracy of a computation can be dramatically different depending on whether the intermediate values were kept in registers or converted and stored/loaded from memory. Of course, careful register allocation can use this fact to provide higher accuracy for computations while using only as much memory space and access bandwidth as the lower-precision formats. The problem is that implementing an 80-bit, or even a 64-bit, floating-point pipeline requires significantly more hardware than a 32-bit pipeline; further, pure 32-bit operations may be slower than necessary thanks to pipeline timing being optimized for the higher-precision use of the function units.

The reason that 32-bit, 64-bit, and 80-bit floating-point formats are so common today is that they are codified in the IEEE 754 standard[10]. IEEE 754 is closely linked to the development of Intel's 8087[11] – the microcoded floating-point coprocessor that first implemented X87. Before the mid 1980s, nearly every computer manufacturer had their own floating-point formats – some manufacturers even had multiple incompatible formats on different models. IBM's systems were well known for base 16 floating point, many others used binary and still others used base 10. Floating-point representations had different sizes as well; the number of bits in a floating-point value on various machines included 16, 18, 22, 24, 32, 36, 48, etc. Generally, the standardization of floating-point formats has been a huge help in ensuring that predictably accurate and reasonably portable floating-point code can be written, but the focus of the IEEE 754 standard was on producing really good numerical characteristics for a few formats, not on minimizing the hardware nor maximizing the flexibility.

Within the IEEE 754 guidelines, there is really only one standard mechanism that facilitates higher accuracy without moving to a higher precision. The concept of a "fused multiply-add," here called a `MADD` instruction, allows the extra bits that naturally exist as two mantissas are multiplied to be preserved through the addition. More specifically, multiplication of two k-bit mantissas naturally creates a $2k$-bit result, but only the properly-rounded top k bits would be used as the multiplication result and some hardware only generates these bits. Within a fused MADD instruction that generates all $2k$ bits, if the value added to happens to cancel some or all of the high bits of the multiplication result, bits from the normally-discarded low portion of the mantissa multiplication result can be retained. In this way, a fused MADD instruction provides accuracy as high as twice the precision. It should be noted that MADD instructions often are *not* implemented as true fused operations, in which case extra bits are not used in the addition and no improvement in accuracy is obtained.

1.2 Floating-Point Software Approaches

Many processors – over 70% of all processors sold in 2002 [12] – do not implement floating-point operations directly in hardware, instead simulating floating-point operations using integer operations to explicitly operate on exponent and mantissa components each stored as signed integer values. This approach has been employed not only in microcontrollers targeted at low-cost embedded-system applications, but also in supercomputers aimed at the scientific computing community. For example, the SIMD processing element hardware of the Thinking Machines CM1 and CM2[13] provided only 1-bit multiplexor operations from which integer and floating-point operations of any precision could be constructed. Thus, floating-point speed of an individual processing element was traded for a reduction in circuit complexity that allowed many more processing elements, achieving reasonable floating-point throughput by parallel execution of up to $65,536$ processing elements. There also were many "scaled integer" techniques implemented in software on various machines, but these techniques were quickly forgotten as floating-point hardware support became more common. Multiple precisions can be implemented easily, if not efficiently, using any of these integer-based approaches. For example, the Gnu Bignum library [14] provides a wide range of precisions using native integers in much the same way that bit-slice hardware implementations use slices.

An interesting alternative, which is essentially the approach that this paper improves upon using new hardware structures, involves use of 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 only to align the mantissas. This general approach is perhaps best known in its application to create an approximation to quad precision using two 64-bit doubles, where it is commonly referred to as *double-double*[15],[16],[17]. Our previous work in this area focused on efficient GPU algorithms for using the approach on 16-bit, 24-bit, and 32-bit floating-point formats, speculation algorithms to avoid paying the performance penalty for more accuracy than necessary, and software implementations of the algorithms optimized that run on a GPU[18]. We generically refer to this "doubling" of mantissa precision as *native-pair* arithmetic, since a pair of native-precision values are used to represent each more precise value. Note that native-pair values are not exactly double the precision of a single native value. The IEEE format always has an implied '1' in the most significant position, so the low half may not be aligned flush with the high half. In other words, zeros may intervene between the two mantissas. The total number of bits of precision represented by a native-pair is the number of bits in the two mantissas plus the number of zeroes between the two mantissas. As discussed in Section 4.1, a pair of 32-bit floating-point values has only 48 bits of mantissa, but typically can perform like 49 or more bits. The primary problem is that native-pair arithmetic is slow – usually taking about ten native operations for each native-pair operation. This paper introduces inexpensive hardware to reduce the cost of native-pair operations.

Like the integer-based schemes above, it is possible to combine more than two native values to achieve still greater accuracy. However, the native exponent range becomes the ultimate limit on how many times the native precision can be multiplied. Because each additional native-format value must have an exponent that is adjusted to align its mantissa as an extension of the mantissa of the previous value, the dynamic range of the constructed multi-precision values is not equal to that of the native format, but effectively is narrower. Ignoring this effect was rarely a problem given the number of exponent bits in an IEEE 754 compliant 64-bit binary floating-point value, but a 32-bit floating-point value only has an 8-bit exponent. Thus, a native-pair value will have twice the native mantissa precision only if the exponent of the low value is in range, which implies the high value exponent must be at least 24 greater than the native bottom of the exponent range. With an 8-bit exponent, we have lost nearly 10% of the dynamic range. Similarly, treating four 32-bit values as an extended-precision value reduces the effective dynamic range by 3×24, or 72 exponent steps – which is a potentially severe problem. Put another way, using more than ten 32-bit floating-point values together would constrain the exponent to the point that the data type is effectively a fixed-point value and no additional precision can be obtained.

2 Microarchitectural Support For Native-Pair

Much of the overhead of native-pair arithmetic comes from computing the residual or error term resulting from floating-point arithmetic operations. For addition, subtraction, and multiplication part or all of the residual term is easily obtainable inside the floating-point unit as a side effect of the native floating-point operation. For example, when two floating-point numbers are added, the addend with the smaller magnitude is shifted to the right to align the radix points of the two addends. Any excess significant bits are discarded and a primary result is produced. Native-pair addition achieves higher precision by storing the discarded bits in a second floating-point number. With standard floating-point hardware, native-pair needs four instructions to compute precisely the bits that are discarded. Moreover, division and square root use multiplication with a native-pair result, so speeding up native-pair multiplication will speed up native-pair division and square root.

We propose adding a *residual register* to save this kind of "left over" information for floating-point addition, subtraction, and multiplication. The residual register is simply a floating-point register with a sign bit, n_e exponent bits, and $n_m + 2$ mantissa bits, where n_e is the number of exponent bits in a native floating-point number and n_m is the number of mantissa bits in a native floating-point, not including the leading one bit implied by the IEEE format [10]. The residual register also has a complement flag to indicate whether the residual mantissa should be complemented as it is moved to an architectural register (see Section 2.1 for details.)

The value stored in the residual register is not necessarily normalized, and does not have an implied leading 1 bit. The lack of normalization is hidden from application programs, however, because the residual register cannot be directly accessed. A simple way to access the residual register[1] is by adding a `MOVRR` instruction to the instruction set architecture. The "`MOVRR` *`reg`*" instruction copies the residual register value to an architectural register and normalizes it to IEEE 754 format. Each arithmetic operation overwrites the previous value of the residual register with the current residual value. The residual register does not change the primary result produced by any floating-point instructions. That is, programs that do not use the residual register will get the result mandated by the IEEE 754 standard whether they are run on a processor with or without residual register hardware.

Several definitions will be helpful to explain our notation. This paper presumes that any input number is precisely represented in the format used. For example, a variable x cannot be assigned a value precisely equal to $1/3$, but it can be assigned the 32-bit IEEE 754 approximation to that value. 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 '$\times$'. For these operations, the primary and residual results ideally have the property that $x \circ y = fl(x \circ y) + res(x \circ y)$.

IEEE floating-point arithmetic allows for several different rounding modes [10]. The property $x \circ y = fl(x \circ y) + res(x \circ y)$ holds true for the round-to-nearest mode, but is not necessarily true for other modes. How the floating-point unit determines which direction to round does not change the residual computation. The residual logic only needs to know that rounding occurred and if it did, which direction the primary result was rounded. When $fl(a \circ b) = a \circ b$, the primary result is precisely correct and the residual is zero. When $fl(a \circ b) < a \circ b$, the primary result p has been *rounded down* to the floating-point value with the next lower magnitude. The residual r should have the same sign as p to make $a \circ b = p + r$. If $fl(a \circ b) > a \circ b$, then p has been *rounded up* to the value with the next larger magnitude. The r should have an adjusted value and the opposite sign of p to decrease the magnitude of p when added to it.

2.1 Native-Pair Addition and Subtraction

When two floating-point numbers a and b are added, the addend with the smaller magnitude is shifted to make its radix point align with the radix point of the larger-magnitude addend. If signs of both numbers are the same, the magnitudes are added. If they are different, the magnitudes are subtracted. In either case the result gets the sign of the larger addend. We discuss addition in detail, but the subtraction $a - b$ is performed simply by toggling the sign bit of b and adding the two numbers. Floating-point adders determine whether a or b has larger magnitude, and swap them if necessary, so we can assume a has a larger magnitude than b without loss of generality.

How the residual register is set depends on the signs of the two numbers and whether rounding occurred when computing the sum. 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 in the rightmost position, and the exponent set to $exp(b)$ when $exp(a) - exp(b) \leq n_m + 1$ or the complement flag is not set. When $exp(a) - exp(b) > n_m + 1$ and the complement flag is set, the residual register

[1]More sophisticated handling of multiple residual registers is discussed in Section 2.1.1; the discussion here centers on a single residual register to simplify the exposition.

gets the bits in b with significance ranging from $exp(a) - n_m + 1$ down to $exp(a) - 2(n_m + 1)$, and the exponent is set to $exp(a) - 2(n_m + 1)$. The latter case only arises when the rounding mode is not set to round-to-nearest. The sign and complement flag are set based on the signs of a and b, and whether p is rounded up or down. When the residual register is moved to an architectural register it is normalized and potentially complemented as described below. There are four cases:

Case 1: a and b have the same signs, the result is rounded down or precisely correct; When a and b with the same signs are added, b is shifted to the right to align its radix point with a and the magnitudes are added. The sign of the residual register, $sign(rr)$ is set to $sign(a)$, and the complement flag is cleared. When a and b have the same signs and the primary result is not rounded, $p + rr = a + b$. The residual value, rr, need not be normalized until it is stored to an architectural register.

Case 2: a and b have the same signs, the result is rounded up; When rounding occurs and a and b have the same sign, the magnitude of p is $2^{exp(a)-n_m}$ larger than if it had been rounded down. For $a + b = p + r$ to be true, $p + r = p - 2^{exp(a)-n_m} + rr = p - (2^{exp(a)-n_m} - rr)$ or $r = (2^{exp(a)-n_m} - rr)$. Thus sign bit is set to the opposite of $sign(a)$ and the complement flag is set. When rr is copied to an architectural register, the `MOVRR` instruction computes $2^{exp(a)-n_m} - rr$, either using the floating-point adder or by computing the two's complement of the mantissa bits of rr, after shifting to the appropriate position.

It is possible to perform this shifting at the time rr is stored to an architectural register, however, this would require holding $exp(a)$ until that time. To avoid having to store $exp(a)$, rr can be shifted to the right and $exp(rr)$ increased if necessary so that $exp(rr) \geq exp(a) - (n_m + 1)$. Some accuracy may be lost due to this shift if $exp(b) < exp(a) - (n_m + 1)$, but for round-to-nearest mode, this Case 2 only happens when $|b| \geq 2^{exp(a)-(n_m+1)}$. No mantissa bits are lost when complementing rr, so the property $a + b = p - (2^{exp(a)-n_m} - rr)$ is preserved. Accuracy only will be lost in a rounding mode that causes p to be rounded up when $|b| < 2^{exp(a)-(n_m+1)}$.

Case 3: a and b have opposite signs, the result is rounded down; When a and b have opposite signs, their magnitudes are subtracted. The least significant bit stored in $mant(a)$ has a weight of $2^{exp(a)-n_m}$. All less significant bits in a are assumed to be zero. To subtract the less significant mantissa bits of b from a, the less significant bits of a must borrow $2^{exp(a)-n_m}$from the more significant bits. This borrow makes setting the mantissa, exponent, and complement flag of the residual for this case identical to Case 2. However, the result should be added to p, so $sign(rr)$ is set to the sign of a.

As with Case 2, $a + b = p + (2^{exp(a)-n_m} - rr)$ is always true for round-to-nearest, but not necessarily true for other rounding modes.

Case 4: a and b have opposite signs, the result is rounded up; In Case 3, the magnitudes of a and b are subtracted to give a residual value of $2^{exp(a)-n_m} - rr$. In Case 2, rounding has increased p, so we need to subtract the residual from $2^{exp(a)-n_m}$. In this case both situations are true, so we must subtract the residual value twice and subtract it from p. That is, $r = 2^{exp(a)-n_m} - (2^{exp(a)-n_m} - rr) = rr$. Thus we clear the complement flag, but set $sign(r)$ to the opposite of $sign(a)$.

2.1.1 Architectural Requirements

Figure 1 shows a high-level schematic example of a floating-point adder with a residual register. There are many ways to implement a floating-point adder, but the logic that performs the functions inside the dashed lines is typically present somewhere in the adder. Whether the adder is structured very 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 is broken down into three logical stages: Pre-normalization, Addition, and Post-normalization, though the actual implementation may have more or fewer stages. The Pre-normalization stage determines which addend is smaller and shifts its mantissa to align it with the mantissa of the larger addend. During pre-normalization the `SMASK` block masks off the bits in the smaller mantissa that were *not* lost due to shifting, leaving those that were lost. The least significant bit of the smaller mantissa is always in the rightmost position, so no shifting is required. The exponent difference, which is used to align the mantissas is also used to determine how many bits to mask.

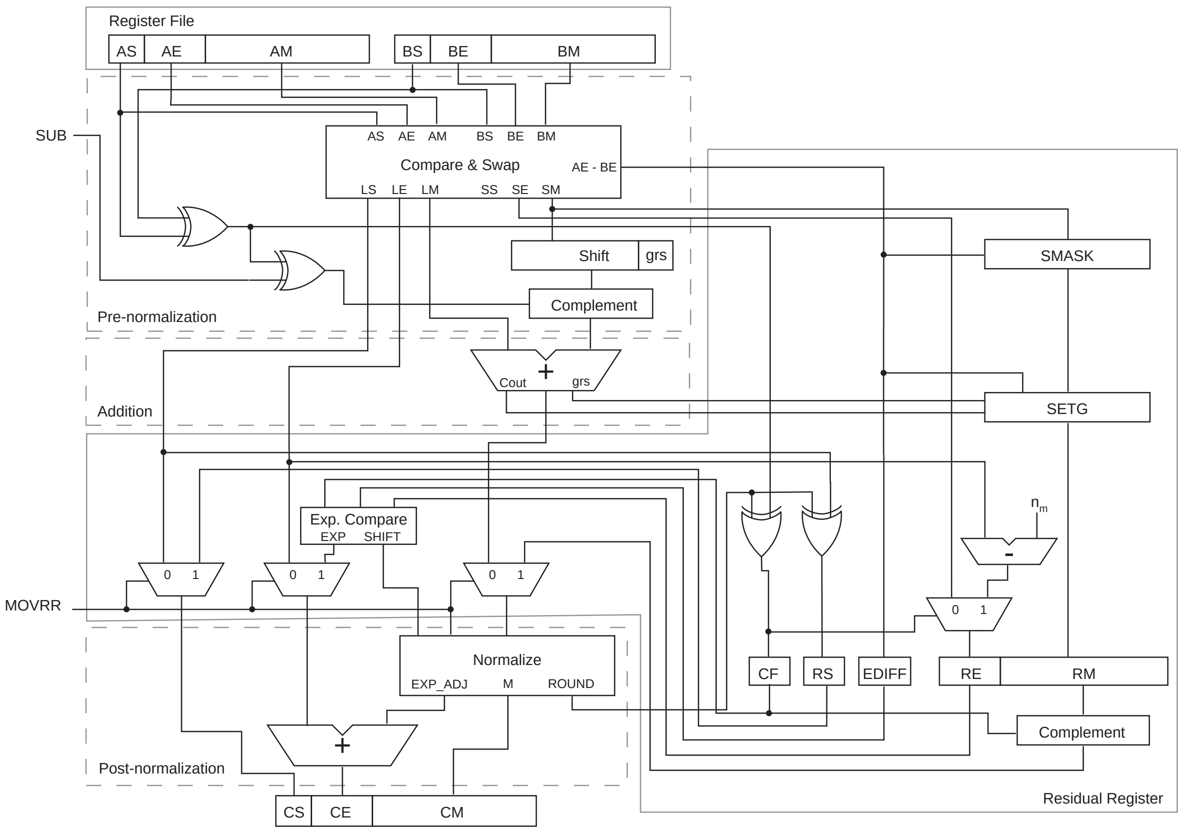


Figure 1: High-level schematic of a floating-point adder with a residual register

If the addition results in a carry out, the position that was the least significant bit of the addends will become the guard bit of the primary result. This guard bit is inserted to the left of the current most significant bit in the residual to become the new most significant bit.

During an ADD or SUB instruction, the MOVRR flag is zero, allowing the sign, exponent, and mantissa bits to flow from Addition to Post-normalization as usual. After the primary result has been normalized the ROUND signal indicates whether rounding occurred (i.e., the unrounded result is different from the rounded result.) The residual register stores the complement flag, sign bit, and exponent, as described above, based on whether rounding occurred.

During a MOVRR instruction, Pre-normalization and Addition are skipped, and the MOVRR signal is asserted. The residual sign is copied to the result. The exponent is computed from the stored exponent, the exponent difference, and the complement flag, as described above. The exponent computation is essentially an addition. If the complement flag is set, the mantissa is complemented. In any case, the mantissa is shifted to normalize the residual result using the same alignment circuitry used by ADD and SUB instructions for post-normalization.

The additional delay with the residual register is not much more than without the residual register. The multiplexers between Addition and Post-normalization are on the critical path for getting results. Other residual register computations will not affect the critical path unless the additional loading on signal lines causes significant delay. Normalization of the residual could be done at the same time as the computation of the result, but an additional barrel shifter would be needed. Having a residual register does not place any additional demand on the existing floating-point unit (FPU) unless and until MOVRR is executed, at which time the FPU's existing alignment circuitry can be used to normalize the residual.

The residual register as described here holds the residual from the most recent FPU operation. In some processors, it may be necessary to have multiple physical residual registers corresponding to the relevant positions within a superscalar pipeline. Many designs can profit from having multiple addressable residual registers so that more complex compile-time instruction orders can be accommodated. However, none of these variations requires circuitry that is more complex than that usually used for tracking values as they move through the pipeline.

Algorithm 1 Native-pair normalization without using the residual register

```
nativepair nativepair_normalize(native hi, native lo) {
  nativepair r; native hierr;
  r.hi = hi + lo; hierr = hi - r.hi; r.lo = hierr + lo;
  return(r); }
```

Algorithm 2 Native-pair normalization using the residual register

```
nativepair nativepair_normalize(native hi, native lo) {
  nativepair r;
  r.hi = hi + lo; r.lo = getrr();
  return(r); }
```

Furthermore, the use of one or more implicitly-set residual registers does not require that changes be made to a machine's basic floating-point instruction encoding or out-of-order instruction scheduling. The only change to the instruction set architecture would be the addition of a `MOVRR` instruction, which could either implicitly operate on the preceding instruction's residual register or, preferably, can be parameterized to specify the residual from a particular operation. For example, the residual from the kth most recent residual-generating operation might be selected by specifying k in the `MOVRR` instruction as ”`MOVRR` *`reg,k`*”. In this case, a fixed-size pool of q residual registers is treated as a circular queue, and a `MOVRR` instruction could be placed anywhere after the instruction that generates the desired residual up to $q-1$ residual-generating instructions later. The translation of k into a renamed register reference can be done at instruction decode. Additional flexibility in out-of-order execution can be obtained by simply constraining `MOVRR` instructions to specifying $k < q-1$.

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. This would require modifying the instruction set architecture by either reserving existing register names or changing the instruction encoding to allow an increased register namespace that would include residual registers. To obtain the maximum benefit from this approach, the residual results need to be immediately available as operands to later instructions; depending on details of the FPU architecture, that might require normalization of the residual values using additional hardware dedicated to that purpose.

2.1.2 Using the Addition Residual Register for Native-Pair Arithmetic

In general, multiple different native pairs of floating-point numbers can represent the same number. To avoid the awkwardness of non-unique value representations, all of the basic native-pair operations end with a normalization step, shown in Algorithm 1, to convert the native-pair result into a canonical normal form. We use simple C syntax with the equivalent of one instruction per statement as a portable assembler for this and all of our other algorithm listings. Given an unnormalized `hi` and `lo` native pair, the normalized `nativepair` value is created without the residual register. The resulting native-pair is equal to the original number, but normalized so that the exponents of the `hi` and `lo` components are set so that the most significant bit of `lo` has less significance than the least significant bit of `hi`. Two native numbers can be added using `nativepair_normalize` to produce a native-pair result, assuming the magnitude of the `hi` input is not smaller than that of the `lo` input. In fact, this constraint on the inputs comes from a deliberate choice made in favor of reducing computational cost; throughout the native-pair routines discussed in this paper, simplifications have been made where the accuracy would be affected only in the least significant bit.

The `nativepair_normalize` function can use the residual register to get the low component directly, as shown in Algorithm 2 – and this version needs no constraints on the magnitude of its inputs. We assume `getrr()` is an inline function that returns the residual register using a single `MOVRR` instruction. This implementation removes one instruction by storing the residual resulting from adding the high and low components. While one instruction may not seem like much, every basic operation ends with a normalization step. Saving an instruction in `nativepair_normalize` reduces the instruction count of every other operation.

Addition and subtraction can make good use of the residual register too. Adding a native floating-point number to a native-pair is common when processing input data. Algorithm 3 adds a native number to a native-pair without residual register hardware. It first adds the native number b to the high component of a, then computes the residual result, adds it to the low component, and finally normalizes the result. Algorithm 4 computes the same result using the residual register. It replaces the five instructions to compute the residual with a single instruction to get the residual.

Algorithm 3 Add a native floating point number to a native-pair without residual register hardware

```
nativepair nativepair_native_add(nativepair a, native b) {
  native hi = a.hi + b; native bhi = hi - a.hi;
  native ahi = hi - bhi; native bhierr = b - bhi;
  native ahierr = a.hi - ahi; native hierr = bhierr + ahierr;
  native lo = a.lo + hierr;
  return(nativepair_normalize(hi, lo)); }
```

Algorithm 4 Add a native floating point number to a native-pair using residual register hardware

```
nativepair nativepair_native_add(nativepair a, native b) {
  native hi = a.hi + b; native hierr = getrr();
  native lo = a.lo + hierr;
  return(nativepair_normalize(hi, lo)); }
```

Adding two native-pair numbers requires one more instruction than adding a native to a native-pair. Algorithm 5 shows a branch-free native-pair addition algorithm. The residual from adding the two hi components is stored in `ahierr` or `biherr`, depending on the values of `a` and `b`. When `a > b`, `bhierr` contains the residual and `ahierr` is zero, otherwise `ahierr` contains the residual and `bhierr` is zero. Both residual terms are computed because on modern architectures computing both `ahierr` and `bhierr` is faster than using a conditional to decide which one to compute. Algorithm 6 shows an equivalent algorithm using the residual register. It eliminates five floating-point operations to compute `hierr` by simply retrieving the residual from the residual register. Overall, addition without the residual register takes 11 instructions compared with 6 for addition with the residual register.

The algorithms for subtraction are similar to addition, except that the `hi` and `lo` components of the two operands are subtracted instead of added. Algorithm listings are omitted from this paper in the interest of space.

2.1.3 Instruction-Level Performance Implications

The multitude of different variations in instruction set architecture and floating point unit design make it difficult to determine the precise impact of the proposed residual mechanism on instruction-level performance; for example, IA32's use of two-register rather than three-register instruction formats forces insertion of an additional instruction where neither source register can be overwritten. Delays in fetching memory operands also can hide the performance of the processor [19]. Examination of the instruction data flow graphs enables the properties to be compared independent of these issues. Native-pair addition without and with addition residual support is shown in Figure 2. Note that SUBR is used to represent a subtract instruction in which the operand order shown in the graph is reversed from the usual left-to-right minuend-to-subtrahend order.

Figure 2 clearly shows that use of residual hardware yields a significantly less complex structure. To determine how dramatic the improvement is, it is necessary to obtain reasonable estimates of the pipeline timing characteristics. Detailed pipeline performance numbers have been published for the latest processors from Intel[20] and AMD[21] using the scalar SSE floating point instructions; these numbers can serve to approximate the expected pipeline characteristics independent of the instruction set. An `ADDSS` or `ADDSD` instruction has a latency of 3, 4, or 5 clock cycles on various Intel processors and 4 on AMD64 processors, thus, each `ADD` or `SUB` typically would have a latency of 4 clock cycles. The `MOVRR` operation only uses the normalization hardware at the end of the floating-point pipeline, clearly yielding a lower latency; a latency of 1 clock cycle might be practical, but let us conservatively assume a latency of 2 clock cycles. By these numbers, the complete latency of the native-pair add without use of residual hardware is the

Algorithm 5 Native-pair addition without the residual register

```
nativepair nativepair_add(nativepair a, nativepair b) {
  native hi = a.hi + b.hi; native lo = a.lo + b.lo;
  native bhi = hi - a.hi; native ahi = hi - bhi;
  native bhierr = b.hi - bhi; native ahierr = a.hi - ahi;
  native hierr = bhierr+ahierr; lo += hierr;
  return(nativepair_normalize(hi,lo)); }
```

Algorithm 6 Native-pair addition with the residual register

```
nativepair nativepair_add(nativepair a, nativepair b) {
  native hi = a.hi + b.hi; native hierr = getrr();
  native lo = a.lo + b.lo; lo += hierr;
  return(nativepair_normalize(hi,lo)); }
```

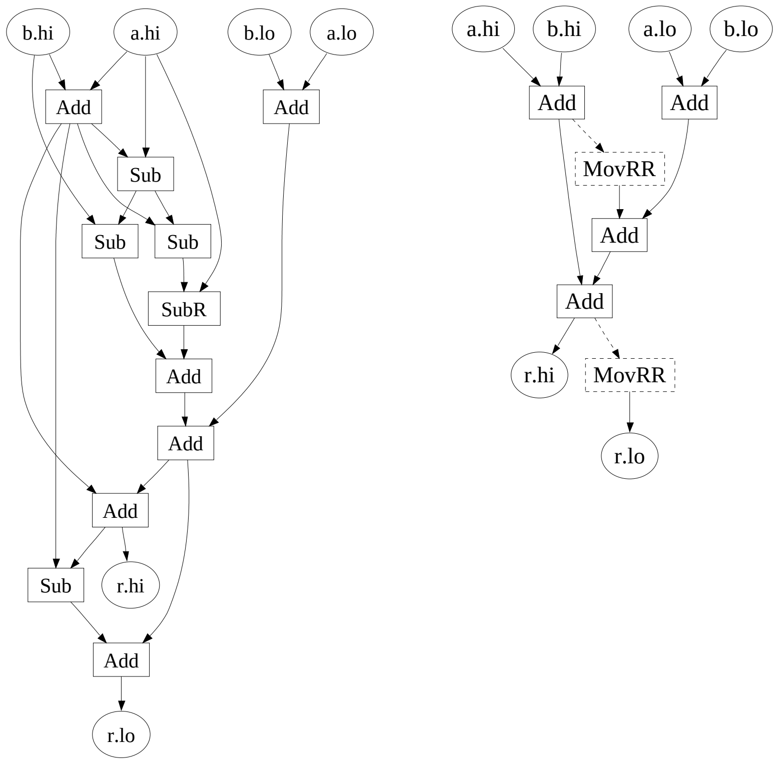


Figure 2: Native-Pair Addition Data Flow, Conventional & Residual Algorithms

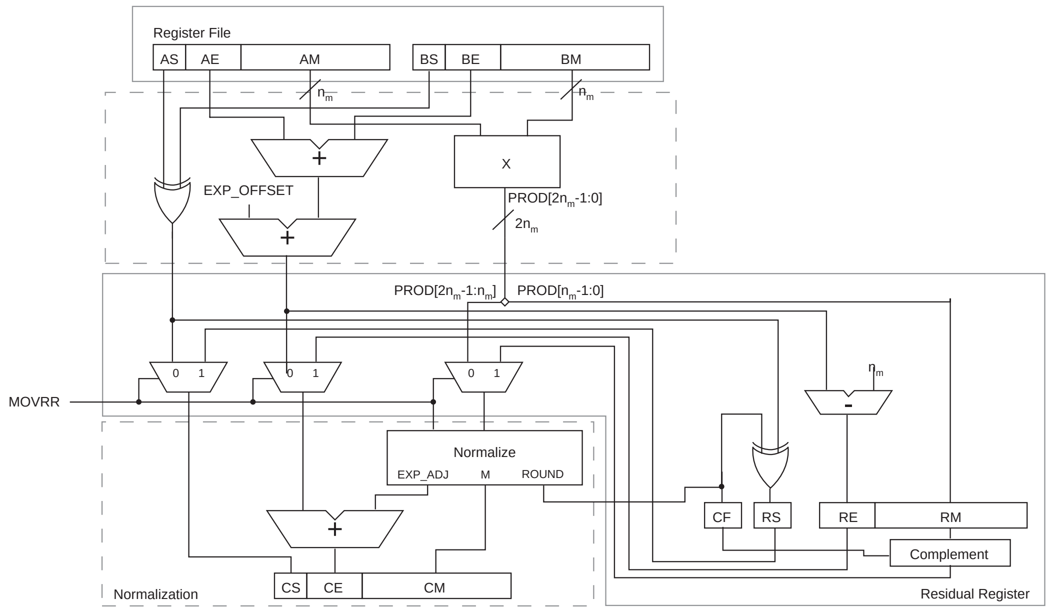


Figure 3: High-level schematic for the multiply residual register

time for a critical path containing 9 instructions: $9 \times 4 = 36$. Using the proposed residual hardware, there are just 3 conventional instructions in the critical path. However, accessing the residuals might require executing two more instructions that also are on the critical path – the MOVRR instructions discussed earlier. Thus, the residual support yields a total latency of $3 \times 4 + 2 \times 2 = 16$ cycles, for an expected $2.25\times$ speedup. Alternatively, at the significant additional expense of duplicating normalization hardware, it would be possible to absorb the MOVRR instructions into the ADDs that generated the residuals, yielding $3 \times 4 = 12$ cycle latency, for an expected $3\times$ speedup.

Due to overlap, a sequence of dependent native-pair adds does not incur the full latency for each operation. Without residual hardware, the hi portion of the result is available 8 cycles before the lo portion and, fortuitously, delaying the lo portions of an input to the algorithm by 8 cycles is not sufficient to alter the critical path; thus, the throughput per native-pair add is one new operation every $36 - 8 = 28$ cycles. Using the residual hardware, the same effect occurs, but the extra MOVRR in generating a lo result precisely matches the extra MOVRR on the path that does not use the lo inputs; this leaves $16 - 2 = 14$ cycles, or exactly $2\times$ speedup over the algorithm not using the residual hardware.

Although the amount of parallel execution permitted by each of the two approaches is primarily a function of the pipeline structure, both methods can be executed with minimum total latency within a 2-way pipeline. However, the simpler and more regular dependence graph for the residual hardware algorithm yields a lower average number of live values. A smaller live count means fewer registers are needed to hold temporaries, which in turn implies more other work can be intermingled with the native-pair add's instructions to fill empty pipeline slots.

2.2 Native-Pair Multiplication

Setting the residual register after multiplication is much simpler than after addition or subtraction. Multiplication of two n bit numbers produces a result with up to $2n$ bits. The $mant(rr)$ 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 $mant(rr)$ with p. When the result is rounded down, the sign is set to match the sign of the product and the complement flag is cleared. If p is rounded up then $a \cdot b = p + r = p - 2^{exp(p)-n_m} + rr$, so $r = 2^{exp(p)-n_m} - rr$. That is, residual register gets the opposite sign of p and the complement flag is set. A high-level schematic to implement the multiply residual register is shown in Figure 3.

The residual is computed and stored in the residual register according to the rules outlined above. The MOVRR instruction optionally complements the mantissa before aligning and storing it. As with the addition residual register, this design adds multiplexers in the critical path. The delay added by the multiplexers may limit the clock cycle time. The implementation in Figure 3 assumes that all $2n$ product bits are available. Many current multiplier architectures only compute the carries for low order bits in the product. For these multipliers, additional hardware is required

Algorithm 7 Native-pair multiply without the residual register

```
nativepair nativepair_mul(nativepair a, nativepair b) {
  nativepair tops = native_mul(a.hi, b.hi);
  native hiloa = a.hi * b.lo;
  native hilob = b.hi * a.lo;
  native hilo = hiloa + hilob;
  tops.lo += hilo;
  return(nativepair_normalize(tops.hi, tops.lo));
}
#define NATIVEBITS 24
#define NATIVESPLIT ((1<<(NATIVEBITS-(NATIVEBITS/2)))+1.0)
nativepair native_mul(native a, native b) {
  nativepair c;
#ifdef HAS_FUSED_MULADD
  /* Actually written for fused multiply-subtract... */
  c.hi = a * b; c.lo = a * b - c.hi;
#else
  native asplit = a * NATIVESPLIT;
  native bsplit = b * NATIVESPLIT;
  native as = a - asplit;
  native bs = b - bsplit;
  native atop = as + asplit;
  native btop = bs + bsplit;
  native abot = a - atop;
  native bbot = b - btop;
  native top = atop * btop;
  native mida = atop * bbot;
  native midb = btop * abot;
  native mid = mida + midb;
  native bot = abot * bbot;
  c = nativepair_normalize(top, mid);
  c.lo += bot;
#endif
  return(c); }
```

Algorithm 8 Native-pair multiply using the residual register

```
nativepair nativepair_mul(nativepair a, nativepair b) {
  native tophi = a.hi * b.hi; native toplo = getrr();
  native hiloa = a.hi * b.lo; native hilob = b.hi * a.lo;
  native hilo = hiloa + hilob; toplo += hilo;
  return(nativepair_normalize(tophi, toplo)); }
```

to compute the bottom n bits of the product. Though adding support for the low order bits adds complexity to the multiplier, no more hardware is required than is needed to implement a fused `MADD` instruction.

2.2.1 Using the Multiply Residual Register for Native-Pair Arithmetic

Algorithm 7 shows how to multiply two native-pair numbers without residual register hardware. The `nativepair_mul` function begins by using `native_mul` to multiply the two high components, which produces a native-pair result from the multiplication of a native value with a native-pair. If the processor has a fused multiply-subtract instruction that preserves the full precision of the product before adding, `native_mul` can be implemented in just two instructions. The first computes the product in native precision and the second subtracts the rounded product from the full product to get the residual. If fused multiply add is available, but not fused multiply subtract, a negate instruction is required to compute `-c.hi`.

Some CPUs have no multiply-add instruction and some processors that do, like GPUs, are not guaranteed to preserve precision with their multiply-add instruction [22]. For these processors the code in the `#else` clause of `native_mul` is used. This code splits the two factors into high and low components and does component-wise multiplication of the components. The residual register greatly simplifies native-pair multiplication when a fused multiply-add is not available.

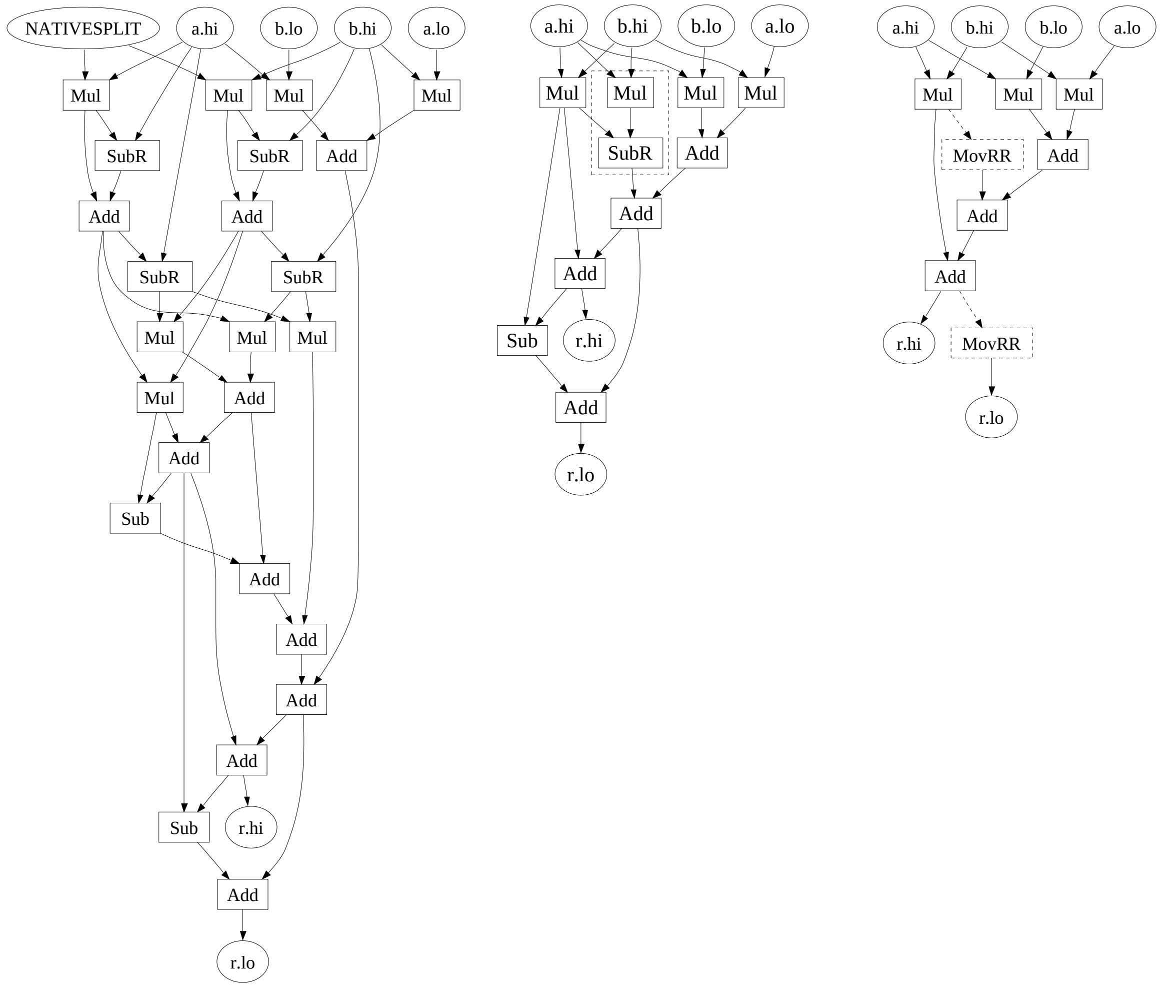


Figure 4: Native-Pair Multiplication Data Flow; Conventional, Fused Operation, & Residual

Algorithm 8 shows a `nativepair_mul` implementation using the residual register. It requires the same number of floating-point instructions as the fused multiply-add version, counting both `MOVRR` and `MADD` instructions as single floating-point operations. When fused multiply-add is not available, the residual register replaces 17 instructions in `native_mul` with two instructions, reducing the number of FLOPs for `nativepair_mul` from 24 instructions to 8 instructions. When a fused-multiply add instruction is available, the residual register does not reduce the number of instructions. However, fused multiply-add requires a wider adder.

2.2.2 Instruction-Level Performance Implications

To better understand the instruction-level performance of the native-pair multiply alternatives, it is useful to again apply the methodology used in Section 2.1.3. According to the pipeline performance numbers published for the latest processors from Intel [20] and AMD [21] using the scalar SSE floating point instructions, a `MULSS` or `MULSD` instruction has a latency of 6 or 7 on various Intel processors and 4 on AMD64 processors. Because this is a relatively large difference, we will compute the timing twice, once using 4 and again using 6.

Figure 4 shows the data flow graphs for the three alternative native-pair multiply algorithms. The conventionally coded native-pair multiply critical path consists of 12 `ADD` or `SUB` operations and 2 `MUL`, which produce a total latency of either $14 \times 4 = 56$ or $12 \times 4 + 2 \times 6 = 60$ cycles. Using an expensive-to-implement fused multiply-subtract instruction dramatically improves the total critical path latency, yielding $6 \times 4 = 24$ or $5 \times 4 + 1 \times 6 = 26$ cycles for a speedup of about $2.3\times$ in either case. The use of residual hardware is even more effective. The five-instruction critical path using the proposed residual hardware costs either $4 \times 4 + 1 \times 2 = 18$ or $3 \times 4 + 1 \times 6 + 1 \times 2 = 20$ cycles to

Algorithm 9 Native-pair division without a residual register

```
nativepair nativepair_div(nativepair a, nativepair b) {
  native qhi = a.hi / b.hi;
  nativepair d = native_mul(qhi, b.hi);
  native ahierr = a.hi - d.hi; native aerr = ahierr - d.lo;
  native anew = aerr + a.lo; native divb = qhi * b.lo;
  native lo = anew - divb; native qlo = lo / b.hi;
  return(nativepair_normalize(qhi, qlo)); }
```

execute, producing $3.1\times$ or $3\times$ speedup over the conventional code and about $1.3\times$ speedup over the fused multiply add code.

While the critical path length serves as a lower limit on the code's latency, it is appropriate to also consider the level of parallel execution that must be sustained so that performance is not limited by pipeline width. Here, a 2-way pipeline structure is not sufficient to complete in the critical path time for either the conventional or fused multiply-subtract implementations. In contrast, the residual algorithm has only one spot that might appear to need a 3-way pipeline structure, and the latency of the `MOVRR` instruction is low enough so that this can be avoided by careful instruction scheduling. If throughput of the `MUL` operation is at least one operation every two clock cycles (both Intel [20] and AMD [21] quote one operation every cycle), scheduling the `MUL` of `a.hi` and `b.hi` to come after the other two `MUL` operations have been initiated allows the residual algorithm to achieve the minimum possible latency using a 2-way pipeline structure.

As for native-pair add, the complete latency is not experienced when a dependent series of native-pair multiply operations is executed using the conventional algorithm; 8 cycles can be saved by overlapping generation of `r.lo` with the start of the next computation. Although both the other algorithms also share the characteristic that `r.lo` is completed later than `r.hi`, the `a.lo` and `b.lo` inputs are on the critical path, so no speedup by overlap is possible. A sequence of native-pair operations interleaving add and multiply will allow overlap using these algorithms, but only between the end of multiply and start of add. This improves the multiply throughput to one operation every 16 or 18 cycles using the fused multiply-subtract or one every 17 or 19 cycles using the residual algorithm. While this sounds better for the fused operation, in fact the pipeline width needed to handle the native-pair multiply in that many cycles would actually allow *two* such operations to be performed every 17 or 19 cycles using the residual algorithm.

2.3 Native-Pair Division and Square Root

Without any additional hardware, Algorithm 9 implements native-pair division by computing an approximation of the quotient and then multiplying the approximation by the divisor to obtain a refined quotient term. This error term is essentially the remainder of the first division.

Compared to addition, subtraction, and multiplication, floating-point divide instructions typically have a very high latency. Current processors from Intel [20] take 23 or 32 cycles for `DIVSS` and 32, 38, or 39 for `DIVSD`; those from AMD [21] take 16 for `DIVSS` and 20 for `DIVSD`. Although it is possible to implement a residual register for divide, the savings in cycles per native-pair operation is not sufficient to justify the circuit complexity because the execution time still would be dominated by the two divide instructions needed. The native-pair square root algorithm has similar issues.

A modest speedup, typically less than $1.2\times$, can be obtained by using the multiply and add residual registers for `native_mul` and `nativepair_normalize`.

2.4 Native Fused Multiply-Add

A fused multiply-add is advantageous when computing $a \times b + c$ if $a \times b$ has the opposite sign as c, and a magnitude close to c. 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, it can be simulated in software using the `native_mul` function listed in Algorithm 7. The `native_mul` function takes 17 instructions to multiply $a \times b$ and two more instructions to add c to the high and low components of the product. Algorithm 10 shows how residual register hardware can implement a fused multiply-add using six instructions and approximately 18 to 20 cycle latency from the multiply inputs; the latency from the add input is 12 to 14 cycles. The product $a \times b$ is stored in `prod_hi`

Algorithm 10 Simulated Fused Multiply-Add using a residual register

```
native fused_madd(native a, native b, native c) {
  native prod_hi = a * b; native prod_lo = getrr();
  native sum = prod_lo + c; native sum_res = getrr();
  sum = sum + prod_hi; return sum + sum_res; }
```

and `prod_lo`, the sum is accumulated in `sum`, and the residual bits are stored in `sum_res`. The high component of the product is then added, followed by the `sum_res`, in case adding `prod_hi` causes heavy cancellation.

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. Moreover, completing both the multiply and the add within the same cycle can force a longer clock period.

3 Residual Register Algorithm Validation

Pairs of numbers in pseudo-random sequences based on two of those described by McNamee [23] are used to validate the residual register computations for both add and multiply. Numbers in the first sequence are randomly generated with a Gaussian distribution having a zero mean and unit standard deviation. The second sequence has pseudo-random numbers of the form $\pm 10^{x_i}$. The sign is randomly determined with each sign equally likely. The x_i's are pseudo-random numbers with Gaussian distribution having zero mean and standard deviation of σ, but values greater than $+\sigma$ are limited to $+\sigma$, and values less than $-\sigma$ are limited to $-\sigma$. The addition test uses σ =35 to exercise large portion of the floating point range. The multiplication test uses $\sigma = 17$ to avoid a large number of the results being out of the range representable with an 8-bit exponent.

The adder simulation program computes the `nativepair` sum of two `native` numbers using both Algorithm 3 and Algorithm 4. Since both numbers are native `a.lo` is set to zero in both algorithms. Algorithm 3 is implemented straightforwardly as listed. The residual result for Algorithm 4 is computed by decomposing the input floating-point numbers a and b into their sign, exponent, and mantissa components. The simulation program emulates the hardware described in Section 2.1 to compute the primary sum and residual results of adding a and b. The results of the two algorithms are compared and an error is flagged if they differ. For each test sequence, one billion pairs of numbers were evaluated and no errors were found.

The multiplier testing program computes the `nativepair` product of two `native` numbers using one algorithm that does not rely on a residual register and one that does use a simulated residual register. The software-only algorithm is similar to Algorithm 7, but optimized for two `native` multiplicands, rather than two `nativepair` multiplicands. Likewise the simulated residual register algorithm is based on Algorithm 8, but with `native` multiplicands. As with the addition simulation, the residual is computed by decomposing the input floating-point numbers into integers representing their sign, exponent, and mantissa components. The simulated floating-point numbers are manipulated as described in Section 2.1 to compute the primary and residual results. The results of the two algorithms are compared and an error is flagged if they differ. For each sequence, one billion pairs of numbers were evaluated and no errors were found.

4 Speculation Support

With or without a residual register, native-pair arithmetic requires multiple native floating-point operations for every native-pair operation. The only reason to use native-pair floating-point is if the native result is not *accurate* enough. With speculative precision [18], an application optimistically tries parts of its computation at a low precision. If an algorithm-dependent accuracy check fails, the computation can be re-executed using higher-precision arithmetic [18], or the low-accuracy results can be refined using higher-precision arithmetic [24],[25].

This paper introduces inexpensive hardware extensions to dramatically decrease the computation required to track two common causes of summation errors: cancellation and absorption. *Cancellation* occurs primarily in summing long sequences. When numbers with opposite signs and similar magnitudes are added, the most significant bits cancel, leaving fewer significant bits in the result. Mantissa bits from smaller numbers added before or between the canceling numbers are lost. *Absorption* happens when the magnitudes of two numbers being added or subtracted differ by enough so that the smaller is treated as zero.

Algorithm 11 Using `max_sum` to detect heavy cancellation

```
native array_sum(native ar[], int len, native *sum) {
  native lsum = ar[0]; native max_sum = lsum; int i;
  for (i=1 ; i<len; ++i) max_sum = max(max_sum, lsum += ar[i]);
  return(((max_sum - (*sum = lsum)) > THRESH) ? SPEC_FAIL : SUCCESS); }
```

The IEEE 754 standard requires a flag to be set, and provides the option of an exception occurring, whenever a floating-point operation generates an inexact result. Most software does not enable the inexact result exception because handling an interrupt every time a result is rounded up or down would seriously impact performance. We propose two additional hardware features to help software decide whether recomputing at higher precision is warranted.

The first is the *absorption counter.* Many processor architectures have performance counters to track CPU events like cache misses, floating-point operations, and pipeline stalls. The absorption counter is a performance counter that is incremented whenever absorption occurs during an addition or subtraction. An application can use it to gauge how much error may be accumulating due to absorption. For example, an application using round-to-nearest rounding may sum a long sequence of numbers. If at the end of the sequence, the absorption counter is above a threshold, then the sequence should be recomputed. The setting of the threshold depends on the application; in general, the error directly caused by 2^n absorption events is expected to be no more than n bit positions worth of accuracy. Thus, it might also be useful to allow triggering an interrupt based on a specific count being reached.

To help reduce error due to cancellation, we propose adding a *peak exponent register*. The concept of measuring cancellation also was used by Dumas and Matula; they proposed hardware that sums separate positive and negative lists of numbers using "un-normal addition" and counting leading zero bits in an extended summation mantissa [26]. In contrast, our peak exponent register works with a conventional adder and normalization, without requiring an extended mantissa (although it also works with native-pairs). Whenever two numbers are added or subtracted, the exponent of the larger magnitude number is compared with the peak exponent register. If an operand has a larger exponent than is stored in the register, the register is updated with the current operand's exponent. A program can test for cancellation by clearing the exponent register before beginning the sum and comparing it with the exponent of the result at the end of the sum. The difference between the peak exponent and the sum exponent indicates how much cancellation has occurred.

The simplest equivalent software technique would be to keep track of the maximum intermediate sum, as shown in Algorithm 11. Compared with simply summing the numbers, this technique would require dedicating a register and executing a maximum operation per element summed. At the end of the sum, the exponent of `max_sum` is the same as the information in the peak exponent register.

4.1 Speculation Experiments

To test the effectiveness of speculation, we used two data sets taken from those McNamee [23] used to evaluate summation accuracy. The first data set, we call "$N(0,1)$", is a sequence of $4,096$ Gaussian distributed pseudo-random numbers with zero mean and unit standard deviation. The second data set is called "Random Heavy Cancellation". It is a sequence of $4,096$ pseudo-random numbers of the form $\pm 10^{x_i}$. The x_i's are pseudo-random numbers with Gaussian distribution having zero mean and standard deviation of 35, but values greater than $+35$ are limited to $+35$, and values less than -35 are limited to -35. The sign is also randomly determined with each sign equally likely. All the test data values were mapped into exact representations in the IEEE 754 32-bit format so that the inputs are exactly the same for all precisions tested.

A simulation program computed the sum of $1,000,000$ sequences from each data set using 32-bit floating point, 64-bit floating point, and 32-bit native-pair and compared them with 304-bit floating-point sum generated with the GNU Multi-Precision library (GMP) [14]. For each sequence, we computed the number of bits equivalent to the GMP answer.

Figure 5 shows the number of bits in the 32-bit sum equivalent to the GMP reference sum as a function of the difference between the peak and sum exponents. For both data sets, the lower bound on number of bits equivalent decreases linearly as the difference between the peak exponent and the sum exponent increases. The number of bits equivalent varies widely at each exponent difference, indicating that cancellation often does not cause the worst case behavior.

The difference between peak exponent and sum exponent can be used as a speculation threshold. Figure 6 shows

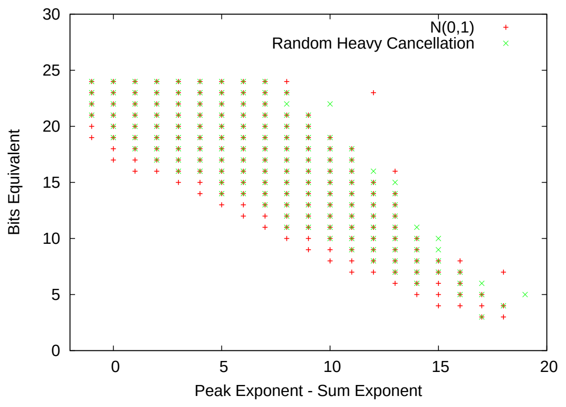


Figure 5: Number of bits in the 32-bit sum equivalent to the GMP reference sum

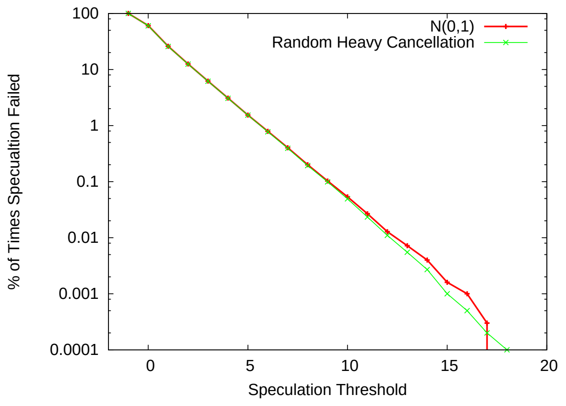
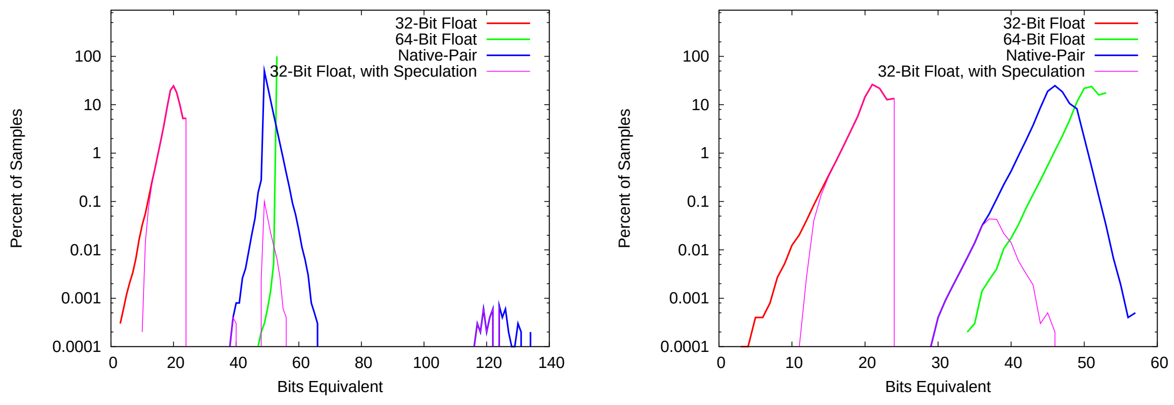


Figure 6: Speculation failure rate as a function of speculation threshold

what percentage of the time speculation will fail when the difference between peak exponent and sum exponent being greater than the threshold triggers speculation failure. The y-axis is plotted in log scale to show the exponential decrease in speculation failure as more sum bits are allowed to be in error. For example, in Figure 5, the minimum number of bits equivalent for a difference of five is 13 bits, and for a difference of eight the minimum bits equivalent is 10. With a threshold of five we would expect a speculation failure rate of about $1.5\,\%$, and about $0.2\,\%$ at a threshold of eight. Previous work has shown native-pair arithmetic to be about 11 times slower on several different architectures [18] (without hardware support.) Thus on average computing a sum of $4,096$ number with speculation would take about $17\,\%$ longer with a threshold of five or about $2.2\,\%$ longer with a threshold of 8.

Figure 7 shows log scale histograms of the percentage of bits in each sum that are equivalent to the GMP reference sum. In both data sets 64-bit data preserves precision significantly better than 32-bit floating point. In fact, for the worst case in Figure 7(a), the 32-bit sum had only three bits equivalent to the reference, compared to 47-bits for 64-bit floating point. For over $99\,\%$ of cases, 32-bit floating point had 15 bits or more equivalent to the reference as opposed to 53 bits for 64-bit floating point. Native-pair had a worst case of 38 bits equivalent with over $99\,\%$ of all cases having 49 bits or more equivalent. Native-pair getting over 48 bits equivalent is somewhat surprising since there are only 48 bits available in the two 24-bit mantissas. However, when zeros follow the high component of a native-pair, the exponent of the low component is decreased, allowing more bits to be represented. In a few cases, the native-pair numbers had more than 100 bits equivalent to the reference answer, but this happened in only 52 out of $1,000,000$ sums.

The curve labeled “32-Bit Float with Speculation”, was generated by setting a threshold difference between peak and sum exponent of eight. When the difference falls below eight, the sum is recomputed with native-pair. As a result the “32-Bit Float with Speculation” curve matches the “32-Bit Float” curve down to 10 bits equivalent. Below 10 bits equivalent, the difference between peak exponent and the sum exponent is greater than eight, speculation fails, and



(a) $N(0,1)$ (b) Random Heavy Cancellation

Figure 7: A histogram showing the percentage of bits equivalent to the reference for each of the 32-bit floating point, 64-bit floating point, 32-bit native-pair, and 32-bit floating point with speculation

native-pair is used. For the $N(0,1)$ data set, speculation failed $2,002$ out of $1,000,000$ times, and for Random Heavy Cancellation it failed $1,933$ out of $1,000,000$ times. In other words, 32-bit precision was accurate enough for about $98\,\%$ of the sums, and the performance penalty of native-pair only had to be paid for $2\,\%$ of the sums.

The execution time for these summations can be estimated from the speculation failure statistics. The summation of n native-pair numbers using a native-pair accumulator will take $t_{np,sum} = nt_{nnp,add}$, where $t_{nnp,add}$ is the time required to add a native number to a native-pair number. The summation of the same sequence will take nt_{add} time when speculation succeeds, where t_{add} is the time required to perform a native addition. When speculation fails, the summation will take $nt_{add} + nt_{nnp,add}$ time. Thus, the expected summation time for a given sequence of numbers is $t_{sum} = P(speculation\,succeeds)(nt_{add}) + (1 - P(speculation\,succeeds))(nt_{add} + nt_{nnp,add})$, and the expected speedup compared with native-pair summation is:

$$speedup = \frac{t_{np,sum}}{t_{sum}} = \frac{t_{nnp,add}}{t_{add} + t_{nnp,add}(1 - P(speculation\,succeeds))}$$

Assuming $t_{add} = 4$ and $t_{nnp,add} = 16$ as in Section 2.1.3, and a speculation success rate of 98%, the expected speedup using speculation is $3.7\times$ that of using a native pair summation.

The expected gain from successful speculation is balanced by the cost of having to recompute the sum at a higher precision when the expected speedup is 1. Using the above latencies and solving for $P(speculation\,succeeds)$, we get $P(speculation\,succeeds) = 0.25$ when speedup is 1. Thus, for the given addition latencies, speculation will be faster on average as long it succeeds at least 25% of the time.

We use native-pair for the summation for simplicity. Similar techniques can preserve residual terms during the sum [23],[27]. Others get better accuracy by reordering the input data set [23],[27],[28] but sorting is often impractical either for memory consumption or performance reasons. The type of speculative execution at lower precision described above is equally applicable independent of the mechanism used to obtain higher accuracy when such is required.

5 Conclusion

Although 32-bit floating-point hardware is now widely available in DSP, SWAR, and GPU processors, and is relatively cheap to implement even in an FPGA, a significant number of potential applications require higher accuracy results than 32-bit intermediate calculations directly provide. Because the primary applications targeted by these processors do not need higher precision arithmetic, it is not economically justifiable to implement 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; usually slightly poorer, on rare occasions markedly better. This enables applications requiring higher accuracy to be run on these machines, but native-pair arithmetic typically carries an order of magnitude performance penalty that cancels much of the price/performance advantage enjoyed by 32-bit floating point systems. The main hurdle to using native-pair is the high cost of computing residual terms using standard floating-point instructions.

The cost of using native-pair can be reduced by adding several simple microarchitectural features, without requiring that higher-precision function units be implemented. The primary change is the augmentation of addition, subtraction, and multiplication hardware with residual registers: a modest hardware enhancement, changing the instruction set only in that a new instruction is added to access the residual value. Using this modification, a typical floating-point processor's native-pair latency is reduced by $2.25\times$ for add or subtract and $3\times$ for multiply, with additional benefits in terms of improved instruction-level parallelism.

Still greater savings can be obtained by providing simple hardware support allowing speculative use of native precision. It is not easy to statically predict when a particular precision will yield sufficient accuracy, but it can be inexpensive to detect when a result might not have the desired accuracy. Without detrimental impact on latency or instruction-level parallelism, the peak exponent register and absorption counter implement dynamic detection of common cases where accuracy might be lost; experiments found that native sufficed more than 98% of the time and speculation was worthwhile if it succeeded at least 25% of the time.

References

[1] R. Rojas, "Konrad Zuse's legacy: the architecture of the Z1 and Z3," *Annals of the History of Computing*, vol. 19, pp. 5–16, 1997.

[2] North Star Computers Inc., *Hardware Floating Point Board FPB-A Manual, 25015B*, 1977.

[3] T. Blank, "The MasPar MP-1 architecture," *35th IEEE Computer Society International Conference (COMPCON)*, February 1990.

[4] H. G. Dietz and R. J. Fisher, "Compiling for simd within a register," in *Languages and Compilers for Parallel Computing* (S. Chatterjee, J. F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, and P.-C. Yew, eds.), pp. 290–304, Springer-Verlag, 1999.

[5] Texas Instruments, *TMS320C3x User's Guide, Texas Instruments Literature Number: SPRU031E*, July 1997.

[6] Advanced Micro Devices, *3DNow! Technology Manual, 21928*, March 2000.

[7] A. Klimovitski, "Using SSE and SSE2: Misconceptions and reality," *Intel Developer UPDATE Magazine*, March 2001.

[8] Freescale Semiconductor, *AltiVec Technology Programming Interface Manual*, June 1999.

[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] C. Severance, "IEEE 754: An interview will William Kahan," *IEEE Computer Magazine*, vol. 31, pp. 114–115, March 1998.

[12] J. Turley, "The two percent solution," *Embedded Systems Design*, December 2002.

[13] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary, Version 5.1*, May 1989.

[14] "The GNU MP bignum library." http://www.swox.com/gmp/.

[15] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, pp. 224–242, 1971.

[16] S. Linnainmaa, "Software for doubled-precision floating-point computations," *ACM Trans. Math. Softw.*, vol. 7, no. 3, pp. 272–283, 1981.

[17] D. H. Bailey, Y. Hida, K. Jeyabalan, X. S. Li, and B. Thompson, “Multiprecision software directory,” *http://crd.lbl.gov/˜dhbailey/mpdist/*.

[18] H. G. Dietz, W. R. Dieter, R. Fisher, and K. Chang, “Floating-point computation with just enough accuracy,” *Lecture Notes in Computer Science*, vol. 3991, pp. 226 – 233, Apr 2006.

[19] K. Fatahalian, J. Sugerman, and P. Hanrahan, “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS)*, (New York, New York), pp. 133–137, ACM Press, 2004.

[20] Intel, *IA-32 Intel Architecture Optimization Reference Manual, Order Number: 248966-013US*, April 2006.

[21] AMD, *Software Optimization Guide for AMD64 Processors, Pub. # 25112*, September 2005.

[22] B. Lipchak, B. Beretta, P. Brown, M. Craighead, C. Everitt, E. Hart, J. Leech, B. Licea-Kane, B. Poddar, J. Sandmel, J. P. Schelter, A. Seetharamaiah, and N. Triantos, “ARB_fragment_program,” *OpenGL Extension Registry*, Aug. 2002.

[23] J. M. McNamee, “A comparison of methods for accurate summation,” *SIGSAM Bull.*, vol. 38, no. 1, pp. 1–7, 2004.

[24] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, “Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy,” Computer Science Tech Report UT-CS-06-574, University of Tennessee, 2006.

[25] K. O. Geddes and W. W. Zheng, “Exploiting fast hardware floating point in high precision computation,” in *ISSAC ’03: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 111–118, ACM Press, 2003.

[26] M. Daumas and D. W. Matula, “Validated roundings of dot products by sticky accumulation,” *IEEE Trans. Comput.*, vol. 46, no. 5, pp. 623–629, 1997.

[27] N. J. Higham, “The accuracy of floating point summation,” *SIAM Journal on Scientific Computing*, vol. 14, no. 4, pp. 783–799, 1993.

[28] K. A. Klein, “A generalized Kahan-Babuska summation algorithm,” *Computing*, vol. 76, pp. 279–293, January 2006.