Appears in the 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems
Chicago, Illinois, October 7-9, 2009

Reduced Precision Checking for a Floating Point Adder

Patrick J. Eibl Andrew D. Cook Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
{pje2, adc11, sorin}@ee.duke.edu

Abstract

We present an error detection technique for a floating point adder which uses a checker adder of reduced precision to determine if the result is correct within some error bound. Our analysis establishes a relationship between the width of the checker adder's mantissa and the worst-case magnitude of an undetected error in the primary adder's result. This relationship allows for a tradeoff between error detection capability and area overhead that is not offered by any previously developed floating point adder checking schemes. Experimental results of fault injection experiments are presented which support our analysis.

1. Introduction

Transient errors have long been a serious matter of concern to designers of systems that operate in noisy or high-radiation environments, or which require extremely high availability. Going into the future, technology trends predict that such problems will become more troublesome for commodity processor designers as well [1]. A large body of research has been directed at tolerating such faults at all levels, including that of microprocessor functional units. Although many such units have had a variety of useful error-detection schemes developed for them that can be tailored to a specific system, limited success has been achieved with the floating point adder.

In setting out to do this work, our initial goal was to develop an efficient error detection scheme that allows for a tradeoff between error detection coverage and area overhead. As an example of such a tunable scheme, consider residue checking in an integer multiplier, whereby we check the accuracy of X*Y by making sure that (X*Y) *mod m* = (X *mod m* * Y *mod m*) *mod m*. The modulus *m* can be adjusted in order to find an appropriate balance between the probability of an undetected error and the hardware cost of the redundant hardware. Unfortunately, no comparable method exists for a floating point adder. The main reason for this is that the floating point add operation is a complicated nonlinear series of sub-operations for which one cannot use a simple function of the inputs to predict the output.

One of the unique aspects of floating point computation as compared to integer operations is that there is a natural ordering of importance among the bits in which, for many applications, the exact value of the least significant bits of the mantissa is much less important than the sign, exponent, and most significant bits of the mantissa. This is much less likely to be the case with integer operations, which frequently represent values for which exactness is absolutely necessary (e.g. loop indices, pointers) and no relative importance can be assigned to any group of bits.

It was this notion of relative significance that led us to the idea of building a reduced precision adder to check the primary adder. If we have a checker adder that does the same computation with the same sign and exponent, but a reduced size mantissa, we can provide

error bounds on the full result with a modest area overhead. The rest of the paper will describe previous related work, our design and analysis of the reduced precision checking scheme, and the results of fault injection experiments.

2. Previous work

The simplest, most general solution that has been applied to floating point adders is to just duplicate the unit and compare the results. However, this comes at over a 100% area and power cost and does not take advantage of any properties of the adder itself.

Low-level error detection represents another approach, with countless schemes having been developed for both storage and computation. A good survey of such techniques is found in Sellers et al.'s book [2], including all manners of parity and residue checkers. However, limited success has been seen in attempts to use these techniques to develop an efficient way to detect errors in floating point adders. Many schemes implemented thus far either only detect errors in the adder's internal buses and storage [3] or work at the circuit level with dual-rail domino logic [4]. Pflanz [5] describes a Berger code scheme for detecting floating point add errors which works by considering the contribution of every subunit to the final Berger code. It is limited in that it cannot trade off error detection versus cost, and it only detects unidirectional errors (two errors can cancel each other). Also, the formulas presented don't account for several aspects of addition like special values and rounding modes.

Ohkubo et al. [6] present an error-detecting floating point unit that uses a combination of parity prediction and residue checks on each of the low-level components of the unit, which are much easier to design than a high-level check for the entire unit. Of course, this type of scheme needs to be redesigned for each specific implementation of a floating point unit, which is the primary disadvantage of this type of low-level checking.

3. Reduced precision checker adder

3.1. Checker adder overview

At the heart of our checking system is a fully-featured reduced precision floating point adder. We maintain the same amount of bits for the exponent, as it is most important that those bits be checked, but we truncate the mantissa to save area and power. The IEEE standard 32-bit floating point format has 1 sign bit, 8 exponent bits, and 23 mantissa bits, and we performed most of our evaluation using checker adders with mantissa widths ranging from 4 to 14 bits (i.e. the adders are 13 to 23 bits wide in total).

A block diagram for the case of a 17-bit checker adder is given in Figure 1. The full computation is done in parallel with the redundant one, with the checker adder only taking the most significant bits of the operands. The rest of the hardware is responsible for comparing the results and determining whether there is an error, and will be described in the remainder of this section. Handling of reported errors (which can include false positives due to faults in the checker adder) is left to higher-level mechanisms, and may include retrying the operation or marking the adder as faulty.

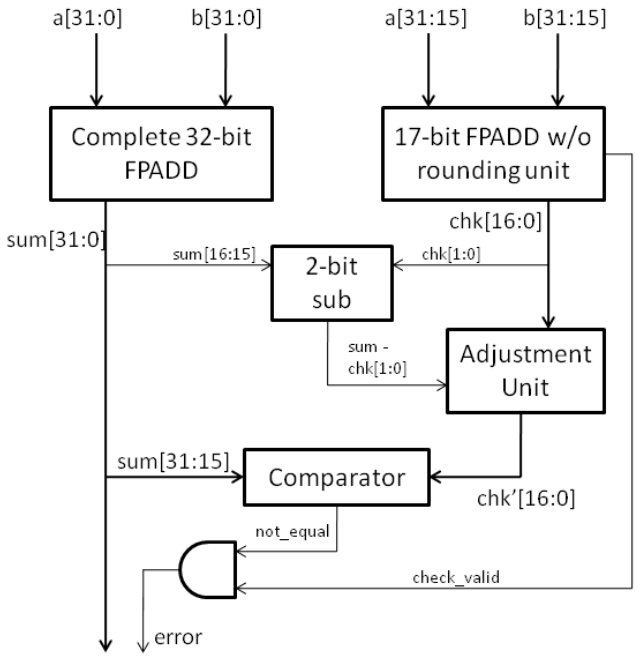

Figure 1: Block diagram of checker

3.2. Adjustment unit

The original idea behind our design was that in the absence of errors, the result of a reduced precision computation should match the full result in the sign, exponent, and most significant bits. Figure 2(a) illustrates this case, in which floating point operands *a* and *b* (expressed in binary) are added to produce *sum*, while a 17-bit checker adder produces *chk*. In the case of operands with like signs, it is also possible that a carry or round will propagate from the truncated LSB's of the original operands and cause the result to be off by one unit in the last place (known as a "ulp") of the reduced precision result, as in Figures 2(b) and (c).

```
a:    1 10011111 01101001110010000100011
b:    1 10001010 10100000100001101110010
sum:  1 10011111 01101001110010000101010
chk:  1 10011111 01101001
```

a) Checker result matches MSB's of full result

```
a:    0 01011101 00100010000100100010000
b:    0 01100100 10110111111100000101011
sum:  0 01100100 10111010001101000111101
chk:  0 01100100 10111001
```

b) Checker result is less than MSB's by one ulp

```
a:    0 01111010 01100111000011000001111
b:    0 01111011 11010100000111000100010
sum:  0 01111100 01000011111010001001010 1
chk:  0 01111100 01000100
```

c) Checker result is greater than MSB's by one ulp

```
a:    1 10010001 00101110110101100110100
b:    0 10011000 00000001110111111101010
sum:  0 10010111 11111111000001000100111
chk:  0 10010111 11111101
```

d) Checker result is less than MSB's by two ulp

```
a:    1 01011011 01110001110011111011100
b:    0 01011011 01111110010010110101011
sum:  0 01010110 10001111011100111100000
chk:  0 01010110 10100000
```

e) Subtraction of like operands; check is useless

Figure 2: Possible relationships between checker result and full result. Bits are separated into sign, exponent, and mantissa fields

Similarly for operands with unlike signs whose exponents differ by more than one, the result can be off-by-one, or in some rare cases, off-by-two, seen in Figure 2(d). Finally, in the case of operands with unlike signs and the same exponent, the result often loses so much precision that the redundant computation is worthless as a predictor of the full result. This is shown in Figure 2(e).

In designing our error detection, we decided that dealing with addition of nearly equal operands with unlike signs is not worth its cost; it is a rare case, and most well-written programs should not be critically dependent on the exact value of such an operation. The *check_valid* signal in Figure 1 is low when the difference between the exponents is zero or one and the signs differ so that an error will not be reported. The off-by-one and -two cases are rather common, and we choose to allow these cases at the cost of a reduction in the guaranteed result precision. The alternative was to predict, given the inputs, exactly whether the MSB's of the output wouldn't match the checker result, but this is a complicated function of the round mode, the truncated bits of the original operands, and the rest of the operand bits. We decided this was not nearly worth the added complexity.

Our solution to the problem of allowing slightly-off cases is a logic block that functions with the truth table in Table 1, which we refer to as the *adjustment unit* in Figure 1. It takes as its input the two-bit difference *sum-chk* between the two LSB's of the redundant computation and the corresponding bits of the full result, and adjusts the checker result *chk* accordingly to produce *chk'* (these labels correspond to those in Figure 1). The only off-by-two errors involve the redundant result being two less, so in that case two is added to the input. The exponent and mantissa can be grouped together as the input to this module, as a carry that propagates past the MSB of the mantissa should increment the exponent. This adjustment logic can be implemented efficiently with a specially configured adder.

Table 1: Truth table for adjustment logic

sum-chk	00	01	10	11
chk'	chk	chk+1	chk+2	chk-1

Having this adjustment unit allows us to make a small optimization in the reduced precision adder. The rounder, which decides whether or not to add 1 ulp to the result, no longer has an effect on the error detection because these small errors are ignored anyway. Therefore, the rounder can be elided, resulting in modest area savings in the redundant adder.

4. Evaluation

4.1. Area overhead

We synthesized gate-level implementations of adders with a range of mantissa sizes using Synopsys Design Compiler. Size varies approximately linearly with mantissa width when speed constraints are constant, which is the case here as the checker adder doesn't need to operate any faster than the full one. The comparison hardware was found to be small relative to the adders (4.5%-6.5% for the widths used), and is offset by the optimization of removing the rounder from the checker adder. Specific overhead percentages for the checker adder will be presented with our experimental results.

4.2. Performance analysis

Although the checker adder can operate in parallel with the primary adder, the extra checking steps that require the results of both imply that error detection will not be as fast as the computation itself. In some designs, the floating point unit output may not be on the critical path, and error detection may fit in the same cycle as the addition.

When this is not the case, we appeal to the concept of lazy error detection [7]. As long as the result of the addition is not committed in the same cycle, the error can be detected a cycle later, still in time to stop the result from committing.

4.3. Error detection capability

4.3.1. Analytical evaluation

This scheme will not be useful without a quantitative evaluation of the bounds on undetected errors. The way in which the adjustment unit accounts for inexactness in the LSB's of the reduced mantissa determines the error bound on the result. Figure 3 demonstrates a worst-case undetected error situation when the checker hardware has no errors.

```
a:     0 00000001 00000000000000000000000
b:     1 00000000 11111111111111111111111
sum:   0 00000000 00000000000000000000001
chk:   0 00000000 00000001
sum':  0 00000001111111111111111
```

Figure 3: Worst case fault scenario – fault causes *sum'* to be produced instead of correct result *sum*, but no error is reported

Without errors, the first m bits of the primary adder's mantissa would have been 1 unit in the last place less than the checker adder's result, which the adjustment unit would correct to be equal. In the case that an error causes the full result to be nearly 3 ulp greater than the check result (the two bits of interest are 2 ulp greater), the adjustment unit will still alter the check result to be equal. Therefore, no error will be reported, yet the full result will be 4 ulp greater than the reduced mantissa. Expressed as a fraction, this means for an m-bit check mantissa, the maximum error is less than $4 \times 2^{-m} = 2^{-m+2}$.

For small m, the bounds may seem somewhat large (e.g. 50% for m=3), but for computations that span several orders of magnitude, there may be situations in which a guarantee of a result within 50% may be useful. The amount of error that is tolerable will be highly dependent on the application, as some programs may have output that is the result of a long chain of these operations without any iterative feedback, for which the maximum possible error will be much greater than that for a single add.

As mentioned previously, the case of addition with unlike signs where the exponents of the operands match is ignored in our current design. These cases are much more difficult to check as they are the only edge cases where the exponent of the sum can vary significantly from the exponents of the operands. For addition of operands with like signs, the result exponent will be either the exponent of the largest operand or one greater; likewise, with unlike signs and sufficiently different exponents, the exponent of

the sum will be the operand exponent of greatest magnitude or one less. However, in cases where the signs differ but the exponents are close, the result exponent will often depend entirely on the least significant bits of the operands that get truncated in the checker adder. Including a simple check for these cases is left for future work, but we also argue that such calculations have limited use anyway, as they result in significant loss of precision.

4.3.2. Experimental evaluation

Methodology. Our goal in these experiments is to model how our scheme would respond to single stuck-at faults injected anywhere in the design, including the primary adder, checker adder, and the adjustment/comparison hardware. The specific adder used in our evaluation is a freely available, entirely combinational design specified in Verilog [8]. Our first attempt at fault injection involved simply forcing wires at the RTL level, but these experiments were strongly biased toward errors in the checkers, and did not appear to come close to modeling real faults. This motivated the methodology that we present here.

For each width we evaluate, we generate flattened gate-level netlists of the adders and comparison hardware. We then insert a "wire breaker" at every combinational net in the design (a technique adapted from work on Argus [9]). When a wire breaker is activated, it inverts the wire's value (simulating an activated fault), rather than letting it pass through unchanged. We choose to invert the value, rather than stick it to zero or one, because our circuit is entirely combinational. For any specific wire, given some input, one of the stuck-at faults on that wire is not activated and must be masked. Therefore, we need only simulate the activated half of fault possibilities, as the result for the other half is known.

The netlists, patched with wire breakers, are then instantiated in a testbench which applies a set of 50,000 input patterns for every fault, and determines whether the output of the full adder is correct, and whether an error was reported. In the case that the result is incorrect and no error is reported, the difference between the actual and expected results is logged.

Error detection results. The results of approximately one billion experiments from combinations of 50,000 inputs, six checker adder mantissa widths, and all associated faults are summarized in Figure 4. The data are categorized based on whether the primary adder output was correct and whether the comparison hardware reported an error. Keep in mind that this chart only includes activated faults. The effect of including all possible faults would be that of vertically compressing the existing chart down to the 50% mark, and filling in the rest with completely masked faults (the top category).

As a starting point for analyzing this chart, it will be helpful to understand the underlying reasons why results will fall into the various categories in Figure 4. The top category occurs when a fault is completely masked, and doesn't affect any of the outputs. As examples of where this may occur, consider a fault on an input to a multiplexor that isn't selected, or on an input to a greater-than comparison that still maintains the same relationship between the inputs. Remember that these categorizations are being done for each individual combination of an input and a fault, so the fact that a fault is masked for one input does not imply any redundant logic. Faults will fall into the second category from the top (false positives) when they strike the checker adder or comparator, as the primary result will be unaffected, but the fault will cause the check to be failed and an error to be reported. The third category is for

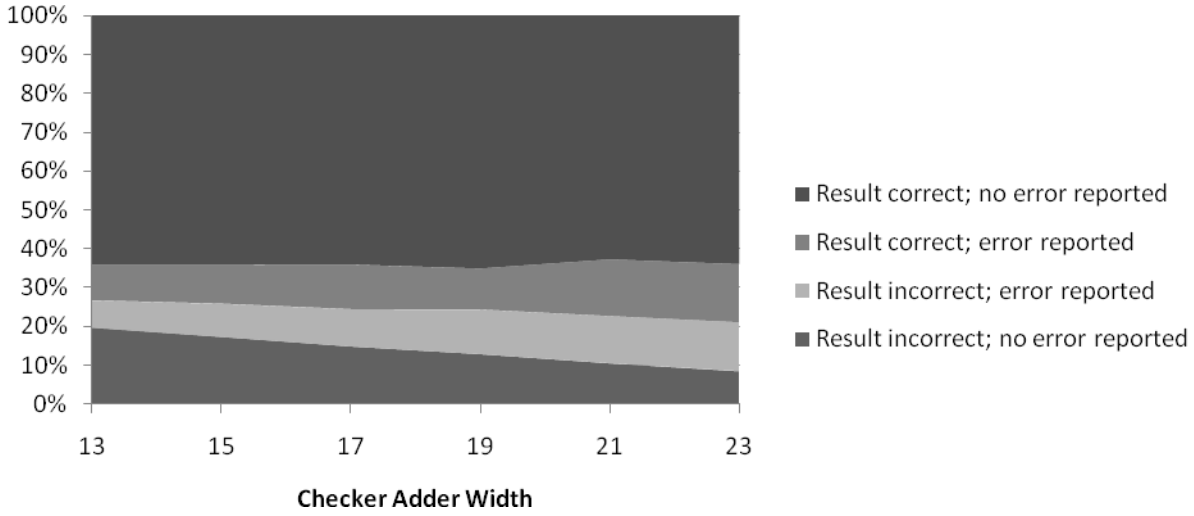

Figure 4: Fault injection experimental results

faults that strike the primary adder and cause enough of an error in the result that the comparator catches it and reports it. The bottom category (false negatives) is for those faults which cause an error whose magnitude is too small to be detected by the checker.

The plot in Figure 4 is largely what we would expect, as we see that the incidence of undetected errors (bottom area) consistently decreases with increasing width of the checker adder, while the number of detected errors increases. The proportion of false positives increases with checker width due to the greater relative size of the checker adder. The figure appears to suggest only a modest tradeoff between error detection capability and area overhead, but this relationship is explored in more detail in the next section.

Undetected error breakdown. For results which were incorrect where no error was reported, the difference from the expected result was logged. The maximum, mean, and standard deviation of the absolute value of these errors all decrease exponentially with increases in the checker width. The shapes of the error distributions are rather irregular, with a very large cluster close to zero and various smaller clusters around certain values in the distribution (mostly powers of two).

Table 2 presents the maximum errors seen in our fault injection experiments for each mantissa width along with the worst-case bound predicted in Section 4.3.1 and area overheads determined as described in Section 4.1. Recall that previously developed error detection schemes for floating point adders have area overheads close to or over 100%. The experimental data support our initial error analysis, as it fits within the worst-case error bounds that were derived. We can see that although the percentage of undetected errors sees only a modest decrease with increasing checker adder width, the size of those errors decreases by three orders of magnitude.

Table 2: Undetected error magnitude and area overhead vs. mantissa width

Mantissa width	4	6	8	10	12	14
Worst-case error bound	25.0%	6.25%	1.56%	0.39%	0.10%	0.02%
Max. experimental error	21.9%	5.27%	1.30%	0.33%	0.08%	0.02%
Area overhead	30%	35%	41%	46%	51%	62%

5. Conclusions

We have presented a best-effort scheme for cheaply detecting errors in a floating point adder which, although not comprehensive, is able to place definite bounds on the magnitude of undetected errors due to single stuck-at faults (excepting cases of operands with unlike signs and equal exponents). Our approach is unique in that it considers the relative importance of the bits in a floating point representation in its design. Based on the constraints of the application in which this scheme will be used, an implementer is free to make the necessary tradeoff between the maximum magnitude of an undetected error and the area and power overhead associated with the checker hardware.

6. Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCR-0444516, Toyota InfoTechnology Center, and an equipment donation from Intel Corporation.

7. References

[1] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (Nov. 2005), pp. 10-16.

[2] F. F. Sellers et al. *Error Detecting Logic for Digital Computers*. McGraw Hill Book Company, 1968.

[3] J. Gaisler. Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications. *Twenty-Fourth International Symposium on Fault-Tolerant Computing.* June 1994.

[4] Naini, A., Dhablania, A., James, W., and Sarma, D. D. 2001. 1-GHz HAL SPARC64® Dual Floating Point Unit with RAS Features. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (June 11-13, 2001). ARITH. IEEE Computer Society.

[5] M. Pflanz, K. Walther, H.T. Vierhaus. On-line Error Detection Techniques for Dependable Embedded Processors with High Complexity. Int. On-line Test Workshop (IOLTW'01). July, 2001.

[6] Ohkubo, N.; Kawashimo, T.; Suzuki, M.; Suzuki, Y.; Kikuchi, J.; Tokoro, M.; Yamagata, R.; Kamada, E.; Yamashita, T.; Shimizu, T.; Hashimoto, T.; Isobe, T. A fault-detecting 400 MHz floating-point unit for a massively-parallel computer. *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*, pp.368-369, 1999.

[7] M. Yilmaz, A. Meixner, S. Ozev, and D. J. Sorin. Lazy Error Detection for Microprocessor Functional Units. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Sept. 2007.

[8] J. Schauer. Opensource Floating Point Adder. http://www.hmc.edu/chips/fpadddocs.pdf

[9] A . Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low- Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.