

Potential Speedup Using Decimal Floating-Point Hardware

Mark A. Erle

Michael J. Schulte

John M. Linebarger

Electrical & Computer Engr.
Lehigh University
Bethlehem, PA 18015

Electrical & Computer Engr.
Univ. of Wisconsin – Madison
Madison, WI 53706

Computer Sci. and Engr.
Lehigh University
Bethlehem, PA 18015

Abstract

This paper addresses the potential speedup achieved by using decimal floating-point hardware, instead of software routines, on a high-performance superscalar architecture. Software routines were written to perform decimal addition, subtraction, multiplication, and division. Cycle counts were then measured for each instruction using the SimpleScalar simulator. After this, new hardware algorithms were developed, existing hardware algorithms were analyzed, and cycle counts were estimated for the same set of instructions using specialized decimal floating-point hardware. This data was then used to show the potential speedup obtained for programs with different instruction mixes and a recently developed benchmark.

1 Introduction

Current floating-point hardware implementations are binary based, not decimal based, for largely two reasons. First, binary data can be stored efficiently and manipulated very quickly on today's "two-state" computers [1]. Second, binary arithmetic is better for error analysis as the relative density of floating-point numbers is proportional to the base [2]. However, since there is not an exact mapping between all decimal and binary values, binary floating-point hardware implementations may produce inaccurate results when dealing with decimal data. Further, since data is often input and stored in decimal format [3], having hardware support for decimal floating-point (DFP) arithmetic eliminates the need for time-consuming decimal-to-binary and binary-to-decimal conversions.

On current computer platforms, the alternatives available to address the aforementioned inaccuracy include decimal software libraries [4] [5] and fixed-point hardware designs that operate on scaled DFP data [6]. These alternatives are slow compared to binary floating-point hardware. With the cost of die space continually dropping, a dedicated DFP hardware im-

plementation is likely to be considered by microprocessor manufacturers. To aid in answering the question of whether the hardware development cost and effort are worthwhile, this paper addresses the potential speedup achieved by using DFP hardware, instead of software routines, on a high-performance superscalar architecture.

A straightforward and familiar approach to estimating speedup is through the use of Amdahl's Law:

$$S_{\text{overall}} = \frac{1}{(1 - F_{\text{enhanced}}) + \sum_{i=1}^n \frac{F_{\text{enhanced}_i}}{S_{\text{enhanced}_i}}} \quad (1)$$

where S is speedup and F is fraction of execution time. Thus, to determine the overall potential speedup, one must estimate or measure the fraction of execution time affected by each enhancement (prior to enhancement) and the speedup of each enhancement.

The enhancements evaluated and presented in this work include the core DFP instructions of add, subtract, multiply, and divide. A C program, described in Section 2, was written to determine the speedup from each of these enhancements, i.e. replacing each DFP software routine with a hardware routine. In Section 3, new hardware algorithms are described and existing hardware algorithms are analyzed to determine the potential speedup for each DFP instruction. Section 4 contains a description of the potential overall speedup for hypothetical instruction mixes and a range of execution time percentages. Then, for a particular benchmark, the potential overall speedup is examined in Section 5. Section 6 includes the summary and conclusions of this work.

2 Software implementation

Input operands of up to 15 binary coded decimal (BCD) digits in length are supported for both the software and hardware algorithms. This length was chosen because it is the proposed length for a single-precision number in [7]. The reader is referred to [8]

for a more detailed explanation behind some of the following algorithms as well as alternative algorithms.

2.1 DFP software add

Addition is accomplished by first significant aligning the addend and augend based on their respective exponents, biasing each digit in the addend with +6 (0x6), adding the respective digit of the augend, and then removing the bias from those digit positions that did not produce a carry. This algorithm works because in BCD, there are six unused combinations of the four binary bits. Biasing each addend digit with six removes the unused combinations in each digit and positions the digit to function as a single hexadecimal digit. Thus, one gets a carry in the upper portion of each digit exactly when the decimal addition of the two digits should produce a carry.

2.2 DFP software subtract

Subtraction is accomplished by first taking the 9's complement of either the subtrahend or the minuend and then performing BCD addition with a carry in (to realize the 10's complement). If there is a carry out from the addition, it is ignored, which essentially subtracts $10^{(n-1)}$ from the result, where n is the number of digits. This is justified because the 10's complement function introduced this value. If there's no carry out from the addition process, the result is a negative number in 10's complement form (and the value introduced from the initial 10's complement function must still be removed). To convert the 10's complement form to sign magnitude form, one simply takes the 10's complement of the result (by taking the 9's complement and then adding zero with a carry in.)

2.3 DFP software multiply

Multiplication is accomplished via N-tupling; iterating over every digit in the multiplier from the least significant position and adding the appropriate multiple of the multiplicand to the partial product (initially 0). With each iteration, the next partial product is added into the next higher digit position (as must be done due to the increasing significance of each digit position in the multiplier).

2.4 DFP software division

Division is accomplished by iterating over each digit position in the dividend, from the most significant position, and successively subtracting the dividend from the remainder (initially the original dividend). A carry out is obtained from the addition (part of the subtraction process) when the difference has become negative,

due to one too many divisors being subtracted. The algorithm is initially started with a zero vector for the quotient equal in length to the dividend. With each subtraction that does not generate a carry out from the addition portion of the subtraction procedure, the quotient digit is incremented in the current position. When a carry out is generated, the quotient digit is unaltered and the algorithm moves to the next lower significant digit position in the quotient and repeats the subtraction process.

2.5 Cycle count measurement

A complete program was developed that iteratively parses instruction types and operand values (in BCD form) and executes these instructions using the aforementioned algorithms. The program was compiled into SimpleScalar [9] assembly code. To obtain cycle counts, each instruction was simulated using a best-case test. For example, a best-case test for multiplication is a value in only one digit position of the multiplier because this results in only a single addition. In contrast, a worst-case test for multiplication is a value in each position of the multiplier. The motivation behind obtaining pessimistic cycle counts for the software implementations is such that when these numbers are compared to the hardware estimates, a conservative estimate results. This is reasonable in light of the fact that 1) the software algorithms may be less than optimal, 2) the C language implementation may consume more cycles than an optimized assembly language implementation, and 3) the hardware cycle count estimates may be optimistic. Table 1 below lists the cycle counts obtained for the core DFP instructions implemented in software.

3 Hardware implementation

The cycle counts for each DFP instruction were estimated based on our experience and on recently published work ([6]). For all the estimates, a fixed worst-case latency was assumed. Establishing a fixed worst-case latency (i.e. no early exit) for each instruction eases the impact of adding DFP logic on the completion unit. This is because the tracking of instruction completion is much less complex.

3.1 DFP hardware add and subtract

Addition is accomplished by generating the BCD equivalent of a 4-bit binary generate and propagate signal for each digit. These signals can then be used in carry look-ahead circuitry to select the appropriate preliminary sum values. This work can be performed in a single clock cycle. However, before the operands

can be added, they must be significant aligned (one cycle). And after the addition, the result may be rounded (one cycle). Subtraction is similar to addition and estimated to also take three cycles.

3.2 DFP hardware multiply

Multiplication is accomplished by first generating the 2-tuple and 3-tuple of the multiplicand and storing these values in temporary registers, and then iterating over each multiplier digit and either adding or subtracting combinations of the stored multiples. Generating the multiples takes two cycles. The decision to add or subtract is based on the value of the current multiplier digit. If the digit is 1-5, the stored multiples are added, and if the digit is 6-9, the stored multiples are subtracted and the next higher significant multiplier digit is incremented. Also with each iteration, the partial product is shifted right and new combinations of stored multiples are added starting at the next higher significant digit position. The worst-case number of additions and subtractions for each multiplier digit is two (e.g. if the multiplier digit is 5, the 2x and 3x multiples need to be added). Thus, for a 15 BCD digit multiplier, 30 cycles may be needed. After the final add/subtract, the product may need to be rounded (one cycle). Note: this algorithm is an adaptation of the algorithm used on the IBM z900 microprocessor [6].

3.3 DFP hardware division

Division is accomplished by first generating the 2-tuple and 4-tuple of the divisor, and then iterating over each divisor digit and successively subtracting combinations of the stored multiples. Generating the multiples takes two cycles. As with the software implementation of divide, the respective quotient digit is incremented until the subtraction, via addition, produces a carry out (indicating that one too many subtractions were performed). The series of subtractions for each divisor digit is shown in Figure 1.

As can be seen, the most subtractions that can occur for a given digit position is four. Thus, for a 15 BCD digit divisor, 60 cycles may be needed. After the final subtraction for the final digit, the quotient may need to be rounded (one cycle). As an aside, this algorithm can be implemented with a 4-bit state machine, the values of which would control the subtract operations for each dividend digit and also contain the value of the respective quotient digit.

Table 1 lists the estimated cycle counts for the core DFP instructions were they to be implemented in hardware.

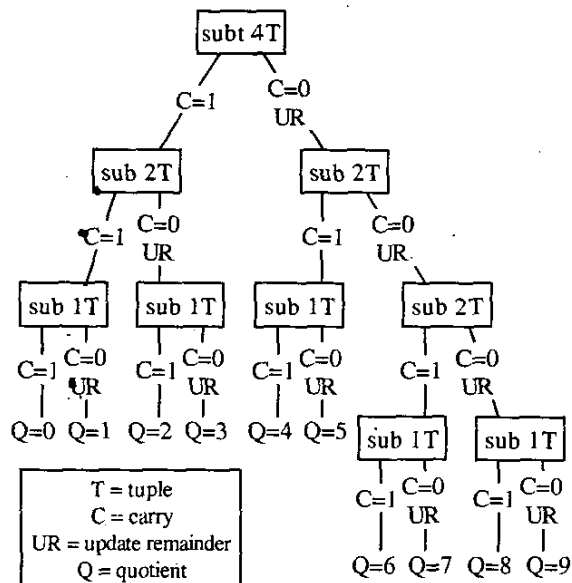


Figure 1: DFP Division Hardware Algorithm

DFP Instr'n	Software (cycles)	Hardware (cycles)	Speedup
Add	652	3	217.33
Sub.	1060	3	353.33
Mul.	4285	33	129.85
Div.	3617	63	57.41

Table 1: Cycle Counts and Speedups for Software and Hardware Decimal Floating-Point Instructions

4 Speedup versus instruction mix

In order to assess the potential speedup to be gained from replacing DFP software routines with application-specific enhancements to a typical microprocessor, three pieces of information are needed. The first piece is the speedup of each instruction, which is given in the rightmost column of Table 1. The second piece is the percentage of CPU execution time that is to be enhanced. Since this is dependent on the application, the potential speedup was calculated for the range 10% to 100%. The last necessary piece of information is the instruction mix. The authors have chosen the three hypothetical instruction mixes shown

in Table 2. The first mix is pessimistic as it minimizes

Mix	Add	Sub.	Mul.	Div.
Pessimistic	24%	8%	60%	8%
SPECFP92	29%	21%	46%	4%
Optimistic	25%	42%	32%	1%

Table 2: Instruction Mixes for Hypothetical Software Programs

subtraction (the instruction that exhibited the greatest speedup). The third mix is optimistic as it favors subtraction. The second mix uses the average relative frequencies for add, subtract, multiply, and divide as they appear in the SPECFP92 benchmark suite.

Figure 2 shows the potential speedup using the data from Table 1 and the instructions mixes in Table 2. From this figure, it can be seen that a halving of execution time is realizable for software spending 50% of its time in decimal routines, and an order of magnitude reduction of execution time is realizable for software spending 90% of its time in decimal routines. Figure 2 further shows that the instruction mix does not significantly affect the potential speedup. For applications that use DFP arithmetic software routines, the percent of execution time spent in these routines should be quite high, since each decimal floating-point operation consumes a large number of cycles.

5 Speedup versus percent execution

Due to increasing industry interest in decimal floating-point, a benchmark has been developed to reflect a typical use of DFP operations. The 'telco' benchmark [10], which calculates the taxes on a million telephone calls, was compiled using gcc, executed on an UltraSPARC II, and analyzed with gprof. Table 3 shows the percent of execution time used by the four DFP instructions examined in this work. Inserting the

DFP Instruction	Execution Time
Add	9.6%
Sub.	0.0%
Mul.	21.1%
Div.	0.0%

Table 3: DFP Instruction Profile for 'telco' Benchmark

percentages of CPU execution time shown in Table 3

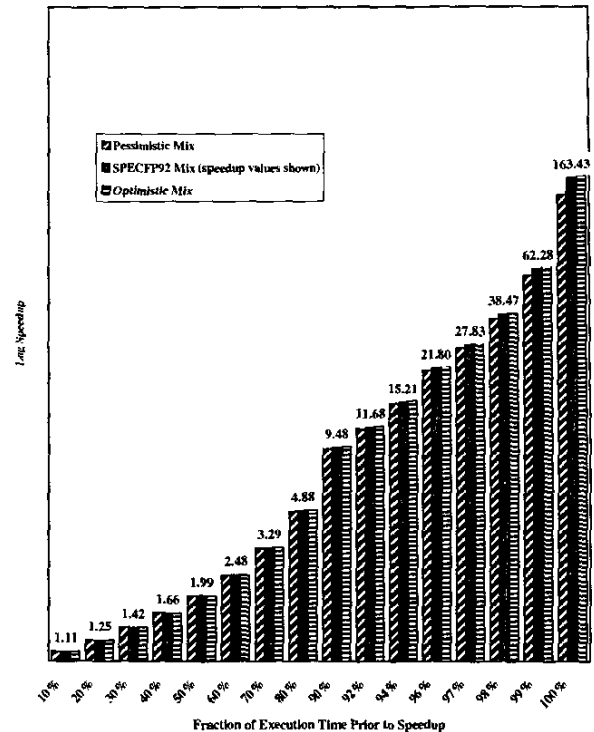


Figure 2: Potential Speedup Using Decimal Floating-Point Hardware with Hypothetical Instruction Mixes

and the respective speedup values shown in Table 1 into Equation 1 yields a potential speedup of 1.44 (or 44%).

6 Summary and conclusions

As a result of converting the DFP add, subtract, multiply, and divide instruction from software routines into hardware, it has been shown applications can realize performance improvements ranging from about 10% (for applications whose respective DFP routines consume 10% of the execution time) to nearly 1000% (for applications whose respective DFP routines consume 90% of the execution time). Thus, the overall speedup approaches the theoretical maximum due to the significant speedup achieved from implementing software routines in hardware.

Acknowledgments

This work is sponsored in part by International Business Machines and Sandia National Laboratories.

References

- [1] W. S. Brown and P. L. Richman, "The Choice of Base", *Comm. of the ACM*, vol. 12, no. 10, pp. 560–561, Oct 1969.
- [2] R. W. Hamming, "On the Distribution of Numbers", *Bell Syst. Tech. J.*, vol. 49, pp. 1609–1625, Oct 1970.
- [3] A. Tsang and M. Olschanowsky, "A Study of Database 2 Customer Queries", IBM Technical Report 03.413, IBM, San Jose, CA, Apr 1991.
- [4] Sun Microsystems, "BigDecimal Java Class", Java 2 Platform, Standard Edition, <http://java.sun.com/j2se/1.3/docs/api/java/math/BigDecimal.html>.
- [5] IBM, "decNumber ANSI C Library", World Wide Web, <http://www2.hursley.ibm.com/decimal/decnumber.html>.
- [6] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM z900 Decimal Arithmetic Unit", in *Conference Record of the Asilomar Conference on Signals, Systems and Computers*, Nov 2001, vol. 2, pp. 1335–1339.
- [7] M. F. Cowlshaw, E. M. Schwarz, R. M. Smith, and C. F. Webb, "A Decimal Floating-Point Specification", in *Proceedings 15th IEEE Symposium on Computer Arithmetic*. IEEE, Jul 2001, pp. 147–154, IEEE Computer Society Press.
- [8] R. K. Richards, *Arithmetic Operations in Digital Computers*, D. Van Nostrand Company, Inc., New Jersey, 1955.
- [9] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.
- [10] M. F. Cowlshaw, "The 'telco' Benchmark", World Wide Web, draft v0.52 2002, <http://www2.hursley.ibm.com/decimal/telcoSpec.html>.