Benchmarks and Performance Analysis of Decimal Floating-Point Applications

Liang-Kai Wang, Charles Tsen, Michael J. Schulte, and Divya Jhalani
University of Wisconsin - Madison, Department of Electrical and Computer Engineering
lwang@cae.wisc.edu, stsen@wisc.edu, schulte@engr.wisc.edu, jhalani@wisc.edu

Abstract

The IEEE P754 Draft Standard for Floating-point Arithmetic provides specifications for Decimal Floating-Point (DFP) formats and operations. Based on this standard, many developers will provide support for DFP calculations. We present a benchmark suite for DFP applications and use this suite to evaluate the performance of hardware and software DFP solutions. Our benchmarks include banking, commerce, risk-management, tax, and telephone billing applications organized into a suite of five macro benchmarks. In addition to developing our own applications, we leverage open-source projects and academic financial analysis applications. The benchmarks are modular, making them easy to adapt for different DFP solutions. We use the benchmarks to evaluate the performance of the decNumber DFP library and an extended version of the SimpleScalar PISA architecture with hardware and instruction set support for DFP operations. Our analysis shows that providing processor support for high-speed DFP operations significantly improves the performance of DFP applications.

1 Introduction

Decimal data permeates society, as humans most commonly use numbers in base-ten. An increasing demand for decimal real number computations across a wide range of exponents has spurred the IEEE 754R Working Group to include specifications for Decimal Floating-Point (DFP) arithmetic in the new IEEE P754 Draft Standard for Floating-point Arithmetic [1]. As companies bring DFP solutions to market, they suffer from a lack of representative workloads by which to guide their design decisions. While not a complete solution, our benchmark suite aims to help ameliorate this problem. Our workloads represent a diverse set of financial applications and serve as a framework upon which additional DFP benchmarks may be built.

DFP computations are critical for many financial and commercial applications. With trends towards globalization, many laws and standards require decimal calculations. For example, the European Union requires currency conversion to and from the euro to be calculated to six decimal places [13]. One study [14] estimates that a large telephone billing system can accumulate errors of up to $5 million per year, if using binary floating-point arithmetic, rather than decimal. Both hardware and software solutions for DFP arithmetic are being developed to remedy these problems [5].

IEEE P754 specifies two DFP encodings. The first encodes its significand using a packed decimal-radix encoding, commonly known as Densely Packed Decimal (DPD) [6]. IEEE P754 refers to this encoding as the decimal encoding of DFP numbers. The other encoding uses a binary integer based significand, and is commonly referred to as Binary Integer Decimal (BID). The standard refers to this encoding as the binary encoding of DFP numbers. The DPD format has the advantage of straightforward decimal rounding and shifting, whereas the BID format has the advantage that it may utilize existing high-speed binary hardware. Though these strengths are understood at a high level, representative workloads can help answer other questions about these encodings and can provide a better understanding of the performance tradeoffs of various DFP solutions.

Further questions that need answering include: (1) What are the most frequent operations and the instruction mix in target DFP applications? (2) What are the average cycle counts of DFP operations in software? (3) What is the potential performance improvement from DFP hardware in target applications? (4) Which DFP operations should go into hardware to improve performance? We do not answer all these questions, but we hope that this work offers an initial solution by which the community can address them.

This paper presents a benchmark suite for DFP computations and the performance analyses of DFP hardware and software solutions. Our benchmarks provide an initial framework to conduct performance analyses of DFP arithmetic, offering several advantages. First, we specifically target applications that use DFP arithmetic. Second, we make the benchmarks modular and flexible such that a variety of solutions may be easily tested. Third, we provide a DFP benchmark suite that can serve as a basis for future

1-4244-1258-7/07/$25.00 ©2007 IEEE

DFP benchmarks. Fourth, we use the benchmark suite to analyze the performance of software and hardware implementations of DFP arithmetic. Our benchmarks and performance analyses serve as a starting point by which designers can choose which operations to optimize and how to optimize these operations. The benchmark suite and simulation infrastructure described in this paper are being made publicly available.

This paper is organized as follows: Section 2 describes related benchmarks and Section 3 describes our new DFP benchmarks. Section 4 discusses our methodology for implementing our benchmarks and using them for performance analysis. Section 5 shows our performance analysis results, and Section 6 presents our conclusions. More details on the benchmarks and performance analysis are given in [25].

2 Related Benchmarks

SPECjbb2005, SPECjAppServer2004, TPC-H, and telco are benchmarks that include DFP datatypes and computations. These benchmarks are well written and representatives of industrial workloads. However, with only four benchmarks, researchers do not have a representative set of DFP applications. Also, these workloads do not cover many important financial and monetary applications; a main target market for DFP arithmetic. Furthermore, these benchmarks are not designed to support different implementations of DFP arithmetic, so comparing a variety of decimal hardware and software DFP solutions may be cumbersome.

SPECjbb2005, developed by the Standard Performance Evaluation Corporation (SPEC), models an inventory system for a wholesale company's warehouses [21]. It uses DFP classes in the Java BigDecimal library to represent financial data [22]. Since SPECjbb2005 operates on decimal data, it can be modified to use IEEE P754 DFP data types. However, this benchmark is primarily designed to measure Java server performance, so it is not ideal for measuring the performance of DFP platforms.

SPECjAPPServer2004 models server transactions for supply chain management in an automotive dealership to measure the performance/scalability of J2EE application servers [20]. It models a growing business that allows dealers to keep track of their accounts, inventories, sales, and purchases. Numerical data in SPECjAPPServer2004 is kept in the BigDecimal DFP format.

TPC-H is a query and data-manipulation based benchmark that mimics a large decision support system [24]. It uses decimal datatypes to keep track of item prices, account balances, part costs, tax rates, order totals, discounts, and order quantities. It models the data analysis portion of a large transaction and database system, and measures the number of queries per hour that a server supports.

The telco benchmark, developed at IBM, models a major telephone company's billing system [14]. It performs billing for millions of calls with correct decimal rounding. This benchmark is written specifically for DFP numbers using the decNumber library and is modified for inclusion in our benchmark suite.

3. Our DFP Benchmarks

Unlike the existing benchmarks from SPEC and TPC, our benchmark suite focuses on monetary computations and financial analyses. Currently, there are five benchmarks in our DFP benchmark suite. The following briefly introduces each benchmark, except the telco benchmark, which is discussed in the website developed by Mike Cowlishaw [14].

3.1. Banking System Benchmark

Banking is one area that requires DFP computations and rounding to guarantee correct results. Typical banking systems deal with a variety of tasks, communicate with database systems, and perform daily maintainance routines to keep account information up to date. Banking systems are usually complex and customized. To emulate daily server activity in an bank, we construct a banking system benchmark, in which we use the chained hash table by Bob Jenkins [17] to model a simple hash-table-based database. Details for each account type in this system are given next.

3.1.1 Credit Card Accounts

The credit card transactions follows the terms and conditions for the Citi® Dividend Platinum Select® MasterCard® [4]. There are three main types of transactions, namely **Purchase**, **Balance Transfer**, and **Cash Advance**, that can be posted to a credit card account. The system receives commands for opening, closing, inquiring about, and paying outstanding debts in users' accounts via file I/O and performs daily maintainance operations to accumulate interest on each credit card account and to detect if accounts are overdue.

3.1.2 Mortgage Accounts

Mortgage transactions are important but complex operations in banking systems. In our banking system benchmark, we focus on home mortgages and model the traditional Fixed Rate Mortgage (FRM) and Adjustable Rate Mortgage (ARM). Four options are provided; FRM-15 Year, FRM-30 Year, ARM-5/1 Year, and ARM-1 Year.

The banking system benchmark reads commands for opening, closing, inquiring about, and paying mortgage accounts. The mortgage rates, points, and margins increase

Table 1. Operations for Euro Conversion

From	To	Operations
Euro	Non-Euro	$Amount \times Rate[To]$
Non-Euro	Euro	$\frac{Amount}{Rate[From]}$
Non-Euro	Non-Euro	$Round(\frac{Amount}{Rate[From]}) \times Rate[To]$

when account holders do not pay their monthly payments on time. Also, premature closure of a mortgage acount may introduce penalty fees based on when the mortgage account is closed.

3.1.3 Checking and Certificate Accounts

In addition to the credit card and mortgage accounts, the banking system also supports normal deposit accounts, such as checking and certificate accounts. The operations for a deposit account are similar to a credit card account. Commands for opening, closing, inquiring about, depositing, withdrawing, and wiring money from accounts are read into the system. In addition, certificate accounts can be set to be extended and closed such that the banking system can perform proper operations when these accounts mature. Like the mortgage rates, all the currency conversion rates and certificate interest rates are from historical records available on the web to provide a more realistic benchmark framework.

3.2 Euro Conversion Benchmark

The introduction of the euro marked the historical step of having a single currency unit for most of the European states [10]. The European Central Bank specified several regulations and monetary policies for euro currency conversion. Currency conversion between the euro and legacy currencies and between two legacy currencies requires using decimal arithmetic [13]. The regulations also require (1) having a unilateral conversion rate from the euro to each legacy currency unit, (2) not using an inverse conversion rate, (3) having six significant figures, including trailing zeros, in the conversion rates from the euro to participating currencies, (4) not rounding or truncating conversion rates, and (5) having an intermediate result in euros when converting between two legacy currency units, where the intermediate result may be rounded to no less than three decimal digits. We have built our currency conversion benchmark based on the above requirements. An array, $Rate[]$, is constructed at program startup to maintain the table for conversion rates. Conversion requests are stored in triples of the form (From, To, Amount). The operations for euro conversion are shown in Table 1.

3.3 Risk Management Benchmark

Risk management seeks to measure and manage risk to earn greater returns over a time period and is an important topic in corporate finance applications. It usually involves several statistical calculations such as variance and covariance computations. Therefore, it is compute-intensive and suitable for our benchmark suite.

We implement the risk management benchmark based on the spreadsheets from [8] using the decNumber library. The risk management benchmark computes risk parameters in the Capital Asset Pricing Model (CAPM) and predicts prices for a company against a stock market valuation. It takes monthly indices, dividends per share, split factors of a company's stock, and a market index (Standard and Poor's 500 Index) from January 1994 to December 2005 to compute several risk measures, variance statistics, and expected returns. This benchmark reads historical data and computes the risk parameters for twenty companies listed on the Standard and Poor's 500 Index. Professor Damodaran on his website [8] gives a brief introduction to CAPM.

3.4 Tax Preparation Benchmark

Our tax preparation benchmark calculates federal taxes for an individual for the 2006 tax year. This benchmark was modified from the Open Tax Solver from Sourceforge [19]. The code follows the instructions provided in IRS documents 1040, Schedules A, B, C, and D. The code was modified to incorporate DFP arithmetic. The inputs to the program are given by W2 and 1099 forms handed out by employers, universities, banks, etc. We generate benchmark inputs using directed pseudo-random numbers with constraints to model a realistic setting from [9], which presents the distribution of adjusted gross income on a sample size of about 132 million tax returns for the 2004 tax year.

In order to support the decNumber library, various changes are made to the code. Appropriate decNumber library functions are called to perform DFP arithmetic operations and comparisons. Since the inputs are already in the DFP format, reading and storing the inputs does not result in any loss of precision. Furthermore, all DFP calculations are performed with decimal rounding, making the final results more accurate. Our tax preparation benchmark reads users' tax form data from a file. After the calculations are made, results are stored in a file. Hence, the amount of file I/O is directly proportional to the number of tax returns.

4 Methodology

We use the decNumber C library to implement the benchmark suite. The benchmark suite is analyzed using the Intel VTune Performance Analyzer [16] and SimpleScalar

toolset [3]. We also modify the SimpleScalar toolset and extend the PISA architecture to model decimal instructions in hardware. DFP function calls in our benchmark suite are implemented with C macros to allow different DFP implementations to be analyzed by making modifications to a single file in the benchmark suite. In all circumstances, gcc compiles the benchmarks with **'-O3'** to enable all optimization. In addition, we provide flexible test generators to facilitate representative simulation results. In this section, we discuss the decNumber library, our modifications to the SimpleScalar toolset, and our testcase generators.

4.1 The decNumber Library

The decNumber library is a fast implementation of the General Decimal Arithmetic Specification [7] in ANSI C. It conforms to the DFP arithmetic specification given in [2] and [23], as well as the current IEEE P754 Draft Standard. The decNumber library is optimized for fast operations on common decimal values (e.g., significands with tens of digits), but supports significands with up to a billion digits of precision and 9-digit exponents. It includes functions for the DFP arithmetic operations defined in IEEE P754 and has support for conversions, rounding, exception handling, and encodings for special values including NaNs and infinity.

The decNumber library we use is version 3.32, which does not include the fused multiply-add function. The decNumber library consists of several modules, which support a decimal number representation that is designed for efficient software computation. The decimal number representation is machine dependent and is optimized for speed, rather than storage efficiency. Thus, it can be viewed as an internal format for processing decimal data. The decNumber library also supports three IEEE P754 standard formats; 34-digit decimal128 , 16-digit decimal64, and 7-digit decimal32. Our benchmarks use the decimal64 format and operations. However, because of the modularity and flexibility of our benchmark suite, we can easily extend the benchmarks to support other decimal formats and other DFP libraries, such as [15].

4.2 SimpleScalar Toolset

Our benchmarks are simulated using the SimpleScalar processor simulator version 3.0 with the PISA architecture and are compiled using the GCC 2.7.2.3 cross-compiler [3]. SimpleScalar is an out-of-order cycle-accurate processor simulator that models the complete processor pipeline, including branch predictors, memory, cache hierarchy, functional units, and queues between processor stages. The configuration of the simulator for our performance analysis is shown in Table 2.

Table 2. Processor Configuration

Processor Core	8-wide Fetch/Decode/Issue/Commit 128-entry Register Update Unit 64-entry Load/Store Unit 4 ALUs, 1 Multiplier 4 BFP ALUs, 2 BFP Multipliers 1 DFP Multifunction Unit (MFU) 1 DFP Multiplier 1 DFP Convert and Bit Manipulate Unit (CBMU) 1 DFP Divider/Square Root Unit (DSRU)
Branch Prediction	Combined Branch Predictor with 16K Meta-table, 16K L1 and L2 Tables, 11-bit History-width, XORed with Address in L2 Predictor A 2K-entry, 2-way Branch Target Buffer A 16-entry Return Address Stack 11-cycle Mispredict Latency
Memory Subsystem	L1 I-Cache: 64KB, 2-way, 64B Line, 3 Cycles L1 D-Cache: 64KB, 2-way, 64B Line, 3 Cycles L2 U-Cache: 1024KB, 16-way, 64B Line, 8 Cycles Memory: 16B Wide, 120 Cycles I/D TLB: Fully Associate, 32 4-Kbyte Page Translation

Table 3. DFP Instruction Set Extensions, Functional Units, and Latencies

DFP Instruction	Functional Unit	Fast Hardware Latency	Slow Hardware Latency
Add	DFP MFU	4	150
Compare	DFP MFU	4	150
Copy	DFP CBMU	1	1
Divide	DFP DSRU	20	400
Max	DFP MFU	3	113
Min	DFP MFU	3	113
Minus	DFP CBMU	1	1
Multiply	DFP Multiplier	6	200
Rescale	DFP MFU	4	150
Square Root	DFP DSRU	20	400
Subtract	DFP MFU	4	150
RoundToIntegral	DFP MFU	4	150
Zero	DFP CBMU	1	1
SetRoundingMode	DFP CBMU	1	1

The GCC cross-compiler is modified so that it can use the binary floating-point register file to store decimal operands in the IEEE P754 64-bit decimal64 format. SimpleScalar is also extended to model DFP instructions using DFP arithmetic units and emulate these instructions using the decNumber library. The decimal operations supported in our enhanced version of SimpleScalar are shown in Table 3, along with their latencies when implemented using fast or slow DFP hardware configurations. In the fast hardware configuration, all the operations, except divide and square root, are fully pipelined and therefore can generate a DFP result every cycle. These operation laten-

Table 4. Testcase Generator Configurations

Name	Test Generation
Banking	Up to 100,000 active accounts Between 100,000 and 150,000 transactions per day for a 30-day period
Euro	100,000 conversions
Tax	One million federal tax return forms for 2006
Telco	One million calls

Table 5. Percentage of Execution Time (PET) for Decimal Functions in Each Benchmark

Function Name	Banking	Euro	Risk	Tax	Telco
Add	**10.1%**	4.5%	11.1%	**7.5%**	**19.0%**
Compare	2.8%	3.6%	N/A	1.5%	N/A
Copy	1.0%	0.9%	0.1%	1.7%	N/A
decimal64FromNumber	N/A	N/A	N/A	2.1%	N/A
decimal64ToNumber	1.3%	2.7%	0.6%	**2.8%**	5.1%
Divide	**29.9%**	**50.7%**	3.9%	1.1%	N/A
Max	0.3%	N/A	N/A	N/A	N/A
Min	N/A	N/A	N/A	0.3%	N/A
Minus	0.0%	N/A	N/A	0.0%	N/A
Multiply	**13.4%**	**12.5%**	**23.1%**	1.5%	**27.5%**
Rescale	N/A	N/A	**18.7%**	N/A	**24.1%**
SetRoundMode	0.7%	0.9%	1.3%	0.0%	2.3%
SquareRoot	N/A	N/A	0.4%	N/A	N/A
Subtract	7.0%	6.3%	**25.6%**	1.3%	N/A
RoundToInt	8.5%	**10.8%**	N/A	0.4%	N/A
Zero	0.3%	0.3%	0.2%	**13.6%**	0.9%
Overall Decimal PET	75.4%	93.1%	85.1%	33.9%	78.9%

cies are from [12, 18, 26, 27]. The slow hardware configuration is used to estimate the performance from DFP hardware if DFP operations have longer latencies, such as the DFP instructions in the IBM z9 architecture, which are implemented in millicode routines [11]. For the slow hardware configuration, the latencies of Add, Subtract, Multiply, and Divide are taken from the latencies provided in Duale's paper [11]. The initiation interval for the slow configuration is equal to the operation latency minus one.

4.3 Test Generators

Most of our benchmarks are equipped with test generators to produce reasonable pseudo-random input data. To make the benchmarks realistic, the range of decimal numbers may be defined by the user based on historical data. In addition, the test generator for the banking benchmark requires specifications of the maximum transactions per day and the period. These and other specifications used for our performance analysis are shown in Table 4.

Table 6. Average Latency per Function Call (ALPFC) in Cycles for Each Benchmark

Function Name	Banking	Euro	Risk	Tax	Telco	Average
Add	302	224	124	121	130	180
Compare	94	105	NA	168	NA	122
Copy	94	49	35	36	NA	54
Divide	1374	2046	3194	1345	NA	1990
Max	397	NA	NA	NA	NA	397
Min	NA	NA	NA	272	NA	272
Minus	186	NA	NA	814	NA	500
Multiply	460	353	316	372	344	369
Rescale	NA	NA	165	NA	226	196
SquareRoot	NA	NA	22347	NA	NA	22347
SetRoundMode	16	15	19	34	15	20
Subtract	254	294	154	256	NA	240
RoundToInt	249	237	NA	242	NA	242
Zero	20	18	14	28	14	19

5 Results

5.1 Performance Analysis

Table 5 presents the percentage of execution time (PET) spent in decNumber functions based on data obtained using Intel's VTune Performance Analyzer and executing the benchmarks on an Intel Pentium 4 Processor. The three decimal functions with the highest PET in each benchmark are shown in bold and underlined. The table also shows in *italic* the conversion functions, decimal64FromNumber and decimal64ToNumber, which can be removed if decimal operations are supported in hardware. Based on Table 5, the DFP Add, Subtract, Multiply, Divide, Rescale, and RoundToIntegral functions account for a significant percentage of the overall execution time in most of the DFP benchmarks. Thus, they are good candidates for instruction set support and hardware acceleration. As shown in the last row of the table, most of the benchmarks spend more than 75% their execution time in DFP functions. The tax preparation benchmark spends about 34% of its execution time in DFP functions due to a large amount of file I/O and string processing.

We compute the average latency per function call (ALPFC) using SimpleScalar. As can be seen from Table 6, the ALPFC of each decimal function in different benchmarks varies significantly, but most of the decimal functions require hundreds of cycles per call. Compared to the latencies of DFP instructions with fast DFP hardware support, shown in Table 3, most arithmetic operations using the decNumber libray run tens to hundreds of times slower.

To evaluate the performance improvement from dedicated DFP hardware, all the decimal arithmetic function calls are mapped to the decimal instructions shown in Table 3. SimpleScalar and GCC are modified such that DFP instructions are recognized by the tool suite. Table 7 shows the improvement of many statistics of the different hardware

Table 7. Improvements for Benchmarks

SW/HW	Banking	Euro	Risk	Tax	Telco
Cycles (Fast)	6.1	31.2	7.5	1.7	5.3
Cycles (Slow)	2.6	3.3	2.1	1.3	1.8
Instruction Count	6.1	21.8	7.9	1.7	5.0
Loads/Stores	7.5	17.4	8.8	1.9	5.5
Branch Lookup	7.3	22.8	7.9	1.5	4.6
IL1 Accesses	6.9	26.8	8.2	1.6	5.2
DL1 Accesses	7.7	20.7	9.0	1.8	5.7

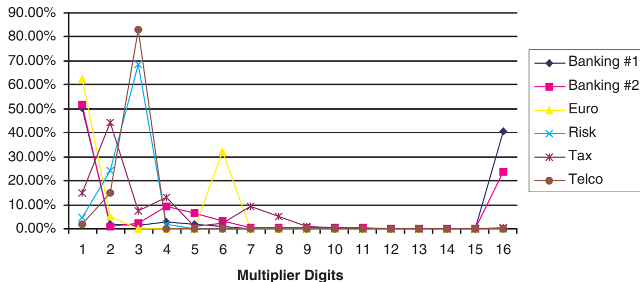

Figure 1. Average number of multiplier digits in our benchmark suite

configurations over DFP software functions. For example in the risk management benchmark, providing instruction set support for DFP operations reduces the total number of instructions executed by a factor of 7.9 and the total number of loads and stores by a factor of 8.8. From the table, the improvements in cycle count range from a factor of 1.7 for the tax preparation benchmark to 31.2 for the euro benchmark if they are simulated using the fast DFP hardware configuration.

The benchmark implementations using the slow DFP hardware configurations are 1.3 to 3.3 times faster than the DFP software implementation. For these benchmarks, the fast DFP hardware configuration is 1.3 to 12.8 times faster than the slow hardware configurations, illustrating the potential performance benefits of fast DFP hardware. It is important to note that these speedups are only estimates. Many compiler optimizations, such as constant folding, constant propagation, and common subexpression elimination, are not executed during the compilation phase for DFP numbers. Furthermore, SimpleScalar does not accurately model system calls and file I/O. In some benchmarks, this may lead to overestimating the potential speedup from DFP instructions and hardware.

5.2 Characteristics of DFP Operands

Table 8 shows operand characteristics for DFP Add, Subtract, Min, Max, Rescale, RoundToIntegral, Multiply, and Divide. Analysis of operand characteristics may lead to insights that can improve DFP hardware designs. From the table, the second row shows the percentage of DFP Add and Subtract operations for which the input operands have the same exponent. In this case, the operation can use a fast path that does not require significand alignment and rarely requires rounding. The third row shows the percentage of Min and Max operations that have the same exponent, and do not require significand alignment.

The remaining rows of Table 8 show how often various DFP operations require rounding in our benchmarks. This data shows that for most of our benchmarks, results from Add and Subtract operations typically do not need to be rounded, whereas the Rescale and RoundToIntegral operations typically require rounding. Much of the input data in the tax preparation benchmark is zero. This leads to a significant percentage of Rescale and RoundToIntegral operations that do not require rounding. For Multiply and Divide, the need for rounding varies greatly between benchmarks. For several DFP operations, it may be useful to design hardware that detects the need for rounding and has the operation take a faster path when rounding cannot occur.

Figure 1 shows the average number of multiplier operand digits in our benchmark suite when performing DFP multiplication. As can be seen from the figure, most of the benchmarks have multiplier operands with less than six significant digits, except for the banking benchmark, which maintains a large number of digits in intermediate results. This indicates a sequential DFP multiplier or a millicode DFP multiplication routine may benefit from exiting early once all the multiplier digits are processed, as proposed in [11, 12].

6 Conclusions

This paper presents a benchmark suite of financial DFP applications. The benchmark suite includes a banking benchmark, a euro conversion benchmark, a risk management benchmark, a tax preparation benchmark, and a telephone billing benchmark. The benchmark suite is being made publicly available.

In addition to describing the benchmark suite, we evaluate these benchmarks using Intel's VTune Performance Analyzer and the SimpleScalar toolset, and extend SimpleScalar and the GCC cross-compiler to model hardware support for DFP instructions. Our results show that when using the decNumber library for DFP arithmetic, most of our benchmarks spend more than 75% of their execution time in DFP functions. We also show that providing fast hardware support for DFP instructions results in speedups

Table 8. Characteristics of Decimal Operands and Results

Operations	Characteristic	Banking	Euro	Risk	Tax	Telco
Add & Subtract	Same Exponent	16.36%	6.71%	98.57%	50.88%	100.00%
Min & Max	Same Exponent	55.22%	NA	NA	51.90%	NA
Add & Subtract	Rounded	3.93%	0.04%	0.02%	0.02%	0.00%
Rescale & RoundToIntegral	Rounded	94.79%	97.24%	99.85%	61.26%	99.57%
Divide	Rounded	6.44%	32.81%	97.81%	2.86%	NA
Multiply	Rounded	25.04%	11.05%	0.19%	0.26%	0.00%

for our benchmarks ranging from 1.3 for the tax preparation benchmark to 31.2 for the euro conversion benchmark.

Acknowledgment

The authors are grateful to Mike Cowlishaw, Aston Roberts, Bob Jenkins, and Professor Aswath Damodaran for supplying their code or spreadsheets, and for providing suggestions for the paper. This work is supported by an IBM Faculty Award and a grant from Intel Corporation.

References

[1] 754 Working Group. Draft standard for floating-point arithmetic P754. http://754r.ucbtest.org/drafts/, October 2006.

[2] American National Standards Institute. REXX: ANSI X3.274-1996. Information Technology - Programming Language. http://www.rexxla.org/Standards/standards.html, 1996.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[4] Citi Bank. Terms and conditions for the Citi® dividend platinum select® mastercard® account. http://www.citicards.com, 2007.

[5] M. F. Cowlishaw. Bibliography of material on decimal arithmetic. http://www2.hursley.ibm.com/decimal/decbibindex.html.

[6] M. F. Cowlishaw. Densely packed decimal encoding. In *IEE Proceedings - Computers and Digital Techniques*, volume 149, pages 102–104, May 2002.

[7] M. F. Cowlishaw. General decimal arithmetic specification. http://www2.hursley.ibm.com/decimal/decarith.html, Oct 2003.

[8] A. Damodaran. Corporate finance. http://pages.stern.nyu.edu/~adamodar/, 2006.

[9] Department of the Treasury, Internal Revenue Service. All returns in tax year 2004. http://www.irs.gov/pub/irs-soi/04in01ar.xls.

[10] Directorate-General for Economic and Financial Affairs. Communication from the commission to the European council. Review of the introduction of euro notes and coins. *EURO PAPERS*, 44:24, April 2002.

[11] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic. Decimal floating-point in z9: An implementation and testing perspective. *IBM Journal of Research and Development*, 51(1/2), 2007.

[12] M. A. Erle and M. J. Schulte. Decimal multiplication via carry-save addition. In *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 337–347, June 2003.

[13] European Commission. The introduction of the euro and the rounding of currency amounts. http://europa.eu.int/comm/economy_finance/publications/euro_papers/2001%/eup22en.pdf, March 1998.

[14] IBM. The 'telco' benchmark. http://www2.hursley.ibm.com/decimal/telcoSpec.html, March 2005.

[15] Intel Corp. IEEE 754r decimal floating-point BID library. http://www.intel.com/cd/software/products/asmo-na/eng/343803.htm.

[16] Intel Corp. Intel VTune performance analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm.

[17] B. Jenkins. An in-memory hash table. http://www.burtleburtle.net/bob/hash/hashtab.html.

[18] H. Nikmehr, B. Phillips, and C.-C. Lim. Fast decimal floating-point division. *IEEE Transactions on VLSI Systems*, 14(9):951–961, 2006.

[19] A. Roberts. Open tax solver. http://opentaxsolver.sourceforge.net/, Feb 2007.

[20] Standard Performance Evaluation Corporation (SPEC). SPECjAppServer2004 design document, Jan. 28 2004.

[21] Standard Performance Evaluation Corporation (SPEC). SPECjbb2005 user's guide, August 10 2005.

[22] Sun Microsystem. BigDecimal class, Java 2 platform standard edition 5.0, API specification. http://java.sun.com/j2se/1.3/docs/api/, 2004.

[23] The Institute of Electrical and Electronics Engineers, Inc. IEEE standard for radix-independent floating-point arithmetic. New York, 1987.

[24] Transaction Processing Performance Council (TPC). TPC benchmark^{TM} H (decision support) standard specification. v.2.3, 2005.

[25] L.-K. Wang. *Processor Support for Decimal Floating-point Arithmetic*. PhD thesis, Department of Electrical and Computer Engineering, University of Wisconsin-Madison, 2007.

[26] L.-K. Wang and M. J. Schulte. Software and hardware support for decimal floating-point arithmetic. *To be submitted to the IEEE Transactions on Computers*.

[27] L.-K. Wang and M. J. Schulte. A decimal floating-point divider using Newton-Raphson iteration. *The Journal of VLSI Signal Processing*, pages 727–739, 2007.