

Article

An Open-Source Evaluation Framework for RISC-V Co-Design-Based Decimal Arithmetic

Riaz-ul-haque Mian ^{1,2,3,*}  and Michiko Inoue ⁴ 

¹ Graduate School of Natural Science and Technology, Shimane University, 1060 Nishikawatsu-cho, Matsue 690-8504, Japan

² Estuary Research Center, Shimane University, Matsue 690-0823, Japan

³ School of Science, Edith Cowan University, Joondalup 6027, Australia

⁴ Graduate School of Science and Technology, Nara Institute of Science and Technology, Ikoma 630-0192, Japan

* Correspondence: riaz@cis.shimane-u.ac.jp

Abstract

Hardware–software co-design is a balanced strategy for computationally intensive algorithms such as decimal computing. It can provide several Pareto points for the development of embedded systems in terms of hardware cost and performance. In this study, we propose an efficient and accurate evaluation framework for decimal computing. The framework was designed and developed for hardware–software co-design decimal arithmetic using the RISC-V ecosystem. New binary and decimal-oriented instructions supported by an accelerator were developed. The framework can perform cycle-accurate analysis for performance and assess hardware overhead for co-design-based decimal arithmetic. Unlike previous studies that focused primarily on implementing and evaluating individual co-design methods, the proposed framework enables exhaustive hardware–software partition analysis at the building-block level, facilitating systematic exploration of the design space. We also evaluated the decimal floating-point multiplication Pareto points and identified a new Pareto point for hardware–software co-design-based decimal multiplication (Method-A) through an analysis with the proposed framework.

Keywords: hardware–software co-design; RISC-V; decimal hardware accelerator; decimal floating-point arithmetic; decimal multiplication; Rocket chip

1. Introduction

Decimal arithmetic is widely used in financial and scientific applications, and most numeric data in commercial databases are stored in decimal format [1]. Traditional binary architectures introduce conversion and rounding errors that are unacceptable for such domains. To address this, IEEE 754-R was revised to include decimal floating-point (DFP) formats and operations [2], with the 2019 update [2] supporting implementations in software [3–8], hardware [9–19], or hardware–software co-design [20,21].

Decimal arithmetic is implemented in many programming languages using a flexible binary hardware unit. In such cases, there is no additional area overhead. These software languages have their own packages or libraries for decimal arithmetic such as Java [3,4,8] and C [5,6], and are used in nanoelectronic technology for future computing facilities [22–24]. Recent studies have also explored decimal arithmetic in emerging computing paradigms [25]. However, software-only implementations are often insufficient for large-scale and computationally intensive applications due to their limited execution performance.



Academic Editor: Lyudmila Mihaylova

Received: 9 June 2026

Revised: 14 July 2026

Accepted: 21 July 2026

Published: 4 August 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

IBM introduced the first decimal accelerator in its mainframe computers [26] and later implemented it in the System z [27] and zEnterprise series [28]. In [10], Fujitsu also introduced decimal floating-point (DFP) in the SPARC64 processor. However, implementing this solution in hardware requires additional area overhead and higher hardware costs. Furthermore, hardware-based decimal computing can be designed using a binary floating-point (BFP) unit via hardware sharing [19,29–31]. Such a design can reduce the area overhead; however, it increases latency toward the binary floating-point unit. This occurs because the BFP and the DFP numbers have different bit widths and encoding [30].

A balanced hardware–software co-design can overcome the limitations of software-only or hardware-only solutions, offering flexibility and optimization opportunities (see Figure 1) that have gained increasing attention in both academia and industry. In previous work [20], we introduced a hardware–software co-design approach for decimal multiplication, combining a software routine with a dedicated hardware unit. However, its evaluation required a special environment: the software part was tested on multiple platforms with the hardware replaced by a dummy function, while the hardware part was separately assessed using CAD tools, providing only an approximate estimate of overall performance.

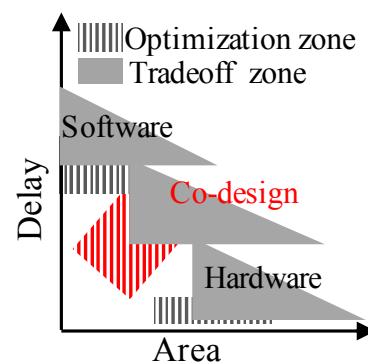


Figure 1. Area–delay trade-off with optimization scope for hardware–software co-design.

The RISC-V ecosystem, built on the open-source RISC-V ISA, provides tools for development, verification, and evaluation, including the Rocket chip and the RoCC interface. As shown in Figure 2, the accelerator communicates with the core and cache via RoCC. By integrating the accelerator into our framework, we enable comprehensive exploration of the design space for hardware–software co-design solutions [32,33].

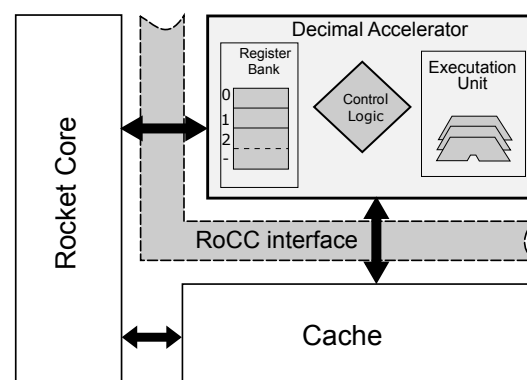


Figure 2. High-level architecture of the proposed RISC-V decimal arithmetic accelerator. The accelerator communicates with the Rocket core and cache through the RoCC interface and consists of a register bank, control logic, and an execution unit for hardware-accelerated decimal arithmetic operations.

Table 1 summarizes the evolution of our previous studies and clarifies the novelty of the present work. In our initial conference paper [20], we introduced the hardware–

software co-design concept for decimal multiplication, proposing four co-design methods and demonstrating their feasibility using software execution with dummy hardware functions together with separate hardware synthesis. Subsequently, we proposed an integrated open-source RISC-V evaluation framework [34], enabling cycle-accurate implementation and evaluation of hardware–software co-design solutions within the Rocket-chip ecosystem. The framework was then employed in our journal paper [21] to implement and comprehensively evaluate the four proposed co-design methods using actual hardware accelerators.

Table 1. Comparison of the contributions of the present work and our previous studies.

Item	[20]	[34]	[21]	This Work
Hardware–software co-design methods	Proposed	Used	Evaluated	Extended
Framework	–	Proposed	Used	Extended
HW implementation	No	Partial	Complete	Reused and Extended
Exhaustive partition analysis	–	–	–	Yes
Building-block analysis	–	–	–	Yes
New Pareto point	–	–	–	Method-A
Benchmark evaluation	No	Framework verification	Method evaluation	Extended
Software-level parallelization reference	–	–	–	Yes

In contrast, the present work builds upon these studies by extending the framework beyond method-level evaluation. Specifically, we introduce exhaustive hardware/software partition analysis through building-block-level area and cycle evaluation, identify a new Pareto-optimal hardware–software co-design solution (Method-A), and extend the benchmark-level evaluation by providing a software-level comparison between software parallelization and hardware acceleration.

The rest of this paper is organized as follows. Section 2 outlines the hardware–software co-design process, and Section 3 introduces the proposed framework. Section 4 presents evaluation results, while Sections 5 and 6 provide design analysis and a new Pareto point. Section 7 discusses overall evaluation, and Section 8 concludes the paper.

2. Co-Design for Decimal Arithmetic

A classification of the hardware–software co-design-based decimal arithmetic process is shown in Figure 3. According to the IEEE 754-2008 standard [2,35], two types of representations (encoding) are allowed. One is the densely packed decimal (DPD) [5,36,37], and another is the binary-coded decimal (BID) [6,7,38].

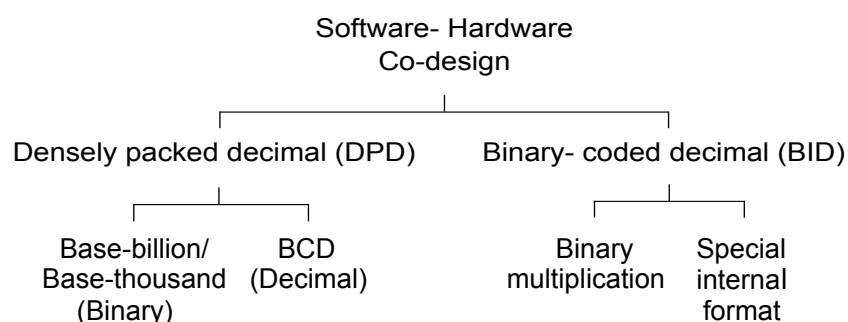


Figure 3. Decimal arithmetic and hardware–software co-design-based classification.

In the BID-based co-design implementation, the embedded binary arithmetic hardware is used with some additional supported hardware components. However, in the

DPD-based co-design implementation, both software and dedicated hardware (BCD or Binary) components are used [20,34], where some parts are calculated by embedded binary hardware while the remaining parts use dedicated decimal hardware.

One of the primary objectives of hardware–software co-design-based decimal implementation is to take certain advantages of using the existing embedded binary arithmetic hardware. A highly optimized embedded binary arithmetic already exists in the system (ALU), or in the case of a field-programmable gate array (FPGA), there are also a variety of IP cores available with binary arithmetic. A decimal co-design solution can then use this arithmetic to reduce the area overhead.

3. Proposed Evaluation Framework

Decimal arithmetic can be implemented using software, hardware, or hardware–software co-design, requiring a common platform for fair evaluation. Hardware implementations are typically assessed in terms of area, power, and delay [39–42], whereas software implementations are evaluated based on execution performance on a target processor [36]. Therefore, a unified evaluation framework is essential for objectively comparing different implementation approaches.

Several evaluation platforms have been proposed. FPGA-based decimal floating-point coprocessors [43] provide hardware acceleration, but their performance is often limited by PCIe communication overhead. Alternatively, extending an open-source processor requires dedicated decimal hardware, custom instructions, compiler support, and simulation tools [44].

An effective hardware–software co-design evaluation framework should support flexible hardware/software partitioning while enabling seamless interaction between software and hardware accelerators. The RISC-V ecosystem provides an open and extensible platform for this purpose through customizable instructions and tightly integrated accelerator interfaces [32]. Therefore, this section presents a complete open-source framework for designing and evaluating decimal arithmetic hardware–software co-design, extending our previous framework [34].

3.1. Overview of the Framework

The proposed framework uses the RISC-V ecosystem, a decimal library, a decimal benchmark application, a decimal arithmetic functional verification database, randomly generated verification test cases, and a set of test programs. The software, tools, and hardware used to develop the proposed framework are described in detail in [34].

The software part of the proposed framework can adopt certain existing processes from [5,6] by replacing certain expensive and suitable parts with hardware as described in [20]; moreover, a completely new software part with new instructions can also be designed. This design includes base-billion [5], base-thousand [5], DPD encoding [37], and binary hardware; however, the design parameters can be flexibly changed by the framework user.

In addition to the hardware instructions and corresponding software programs, the framework includes a C-based test program generator that dynamically configures its parameters. The mandatory configuration specifies the precision format (double or quad), input dataset (rounding, overflow, normal, underflow), and arithmetic operation type (addition, subtraction, multiplication, or others), while the optional configuration specifies the number of repetitions, output pattern (execution time or cycle count), and similar parameters. All configurations have framework-defined default values. The generated test program is used for functional evaluation, with a random decimal input mode additionally provided for performance evaluation.

Figure 4 depicts an overview of the proposed framework. On the hardware side, the necessary finite-state machine (FSM) and hardware description for the accelerator are designed. Furthermore, the Rocket Chip is compiled and built using the newly generated accelerator, and then an executable for the emulator is generated. On the software side, the RISC-V in-line assembly and C source code are compiled using the RISC-V GCC cross-compiler to generate the RISC-V binary of the target program with new custom instructions using the accelerator. The binaries are then simulated by the SPIKE ISA simulator for functional verification. Thereafter, the RISC-V machine code is generated. This RISC-V machine code is then executed on the emulator to obtain the evaluation output and waveform. A detailed description of the architecture (hardware) and programming model (software) is provided below.

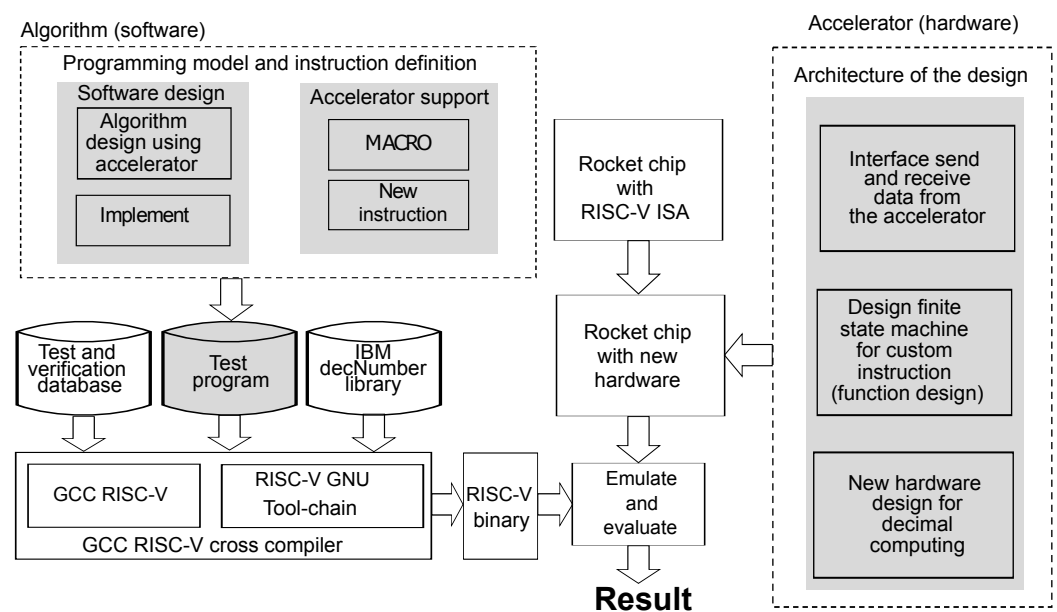


Figure 4. Overview of the proposed framework. The arrows indicate the workflow and data flow between the software design, hardware accelerator development, compilation, simulation, and evaluation stages. Gray-colored blocks denote the components introduced by the proposed framework.

3.2. Architectural Design (Hardware)

To embed decimal arithmetic into a RoCC coprocessor, two main parts, the interface and the execution unit, are required. The RoCC has three default interface signals:

1. Co-ordination between an accelerator and Rocket core (core control)
2. Exchanging data between an accelerator and Rocket core (register mode).
3. Communication between an accelerator and cache (memory mode).

The default interface signal communicates through the core and memory using the command (cmd), request (req), and response (resp) signals. In addition to the default interface, there are certain extended RoCC interfaces, such as an (FPU) interface for sending and receiving data from a floating-point unit.

Figure 5 shows a RoCC 32-bit custom instruction encoding with the bit length of several parts of the instruction. The opcode is selected from custom-0 to custom-3, which are reserved for custom instructions, and each opcode realizes several functions using funct7 bits. Each instruction can have at most one destination register rd and two source registers rs1 and rs2. Three flags, xd, xs1, and xs2, indicate whether the corresponding Rocket-core register is used as a destination or source, i.e., whether synchronization between the Rocket core and accelerator is required: if xs1 = "1" or xs2 = "1", the source-register

value is transferred with the instruction, and if $xd = "1"$, the Rocket core waits for the response. Otherwise, the field specifies an accelerator register-file address or is unused.

Funct7(7) rocc instruction	rs2(5)	rs1(5)	xd (1)	xs1 (1)	xs2 (1)	rd(5)	Opcode(7) Custom 0–3
-------------------------------	--------	--------	-----------	------------	------------	-------	-------------------------

Figure 5. RoCC instruction encoding format (bit fields).

During the evolution of the proposed framework, several custom instructions were refined and additional instructions were introduced as the Chisel implementation matured. Table 2 illustrates the RoCC custom instruction encoding format, whereas Table 3 summarizes the final custom instructions implemented in the proposed framework together with the corresponding funct7 values, opcode assignments, and functionality. Table 3 serves as the definitive instruction specification used throughout the remainder of this paper.

Table 2. RoCC custom instruction encoding format used in the proposed framework.

31	25	19	15			11	6
Instruction	Source-1	Source-2	Addressing Mode			Destination	Opcode
funct7(7)	rs2(5)	rs1(5)	xd(1)	xs1(1)	xs2(1)	rd(5)	opcode(7)
WR (0000000)	00000	01011	1	0	0	00000	0010111
DEC_MUL (0000111)	01010	01011	1	1	1	01100	0010111
DEC_CNV (0000001)	01010	01011	1	1	1	01100	0110111
BIN_CNV (0000010)	01010	01011	1	1	1	01100	0110111
DEC_ADD (0000100)	01010	01011	1	1	1	01100	0010111

Table 3. Final specification of the custom RoCC instructions implemented in the proposed framework.

Instruction	Funct7	Opcode	Function
WR	0000000	custom-0	Write a value into the internal accelerator register.
DEC_MUL	0000111	custom-0	Perform decimal multiplication.
DEC_CNV	0000001	custom-1	Convert a base-billion number into the corresponding BCD representation.
BIN_CNV	0000010	custom-1	Convert a BCD number into the corresponding base-billion representation.
DEC_ADD	0000100	custom-0	Perform decimal addition.

Figure 6 shows the high-level architecture of the accelerator, whereas Figure 7 illustrates the implementation of the finite state machine (FSM) required to implement the accelerator.

For example, when the BIN_CNV instruction is required, it performs an operation that converts a BCD number to the corresponding base-billion number for a single operand and writes the result to the destination register. The pseudo-code written in Chisel for the instruction BIN_CNV (funct7 = "0000010") is as follows.

```
val BIN_CNV = funct === UInt(2) //function bit definition
//calling conversion module
when(cmd.fire() && (BIN_CNV))
//store the result after converting to base-billion single operand
regfile(addr) := CNV(cmd.bits.rs1)
```

When the accelerator receives a BIN_CNV command, it converts the BCD value stored in the source register rs1 into its corresponding base-billion representation and writes the result to the destination register xd. Here, CNV performs the BCD-to-base-billion conversion corresponding to the BIN_CNV instruction.

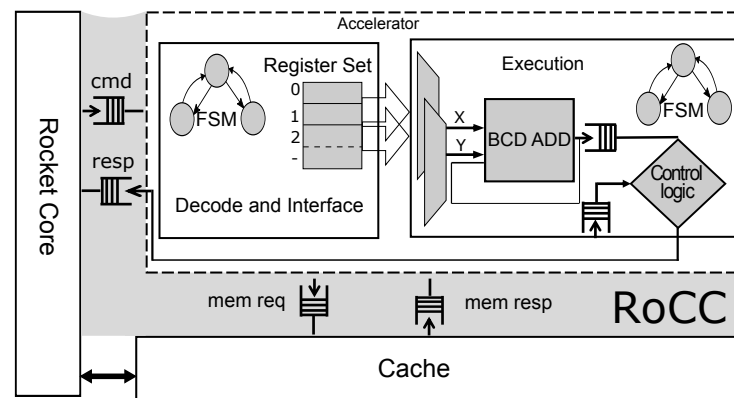


Figure 6. Basic block of Method-1 proposed in [20]. The arrows represent the communication and data flow among the Rocket core, accelerator, and cache.

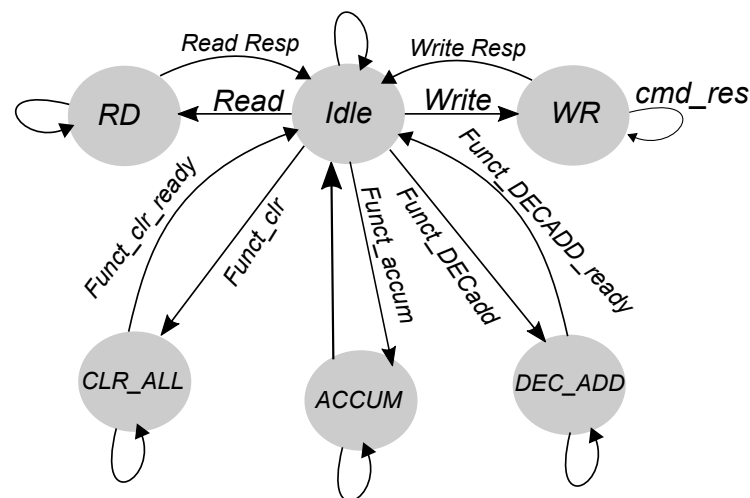


Figure 7. Finite-state machine (FSM) implementation of Method-1 proposed in [20]. The arrows represent state transitions initiated by commands and control signals, including read, write, clear, accumulate, and decimal-add operations.

3.3. Programming Model (Software)

A set of software programs is implemented for the primary algorithm, using dedicated functions to call new instructions supported by the accelerator.

The fragments of pseudo-code using the accelerator are as follows:

```
//For generating 1X-9X
M_multiple[0]=0; M_multiple[1]=X;
for (mm=1; mm<9 ; mm=mm+1){
  //high-level call for in-line assembly
  roccadd(M_multiple[i+1],M_multiple[1], M_multiple[i]);
}
```

```
//to accumulate the PPs
for (pp=0; pp < 64 ; pp=pp+4){
  roccadd(result,result,M_multiple[mcMID(Y_64,j,j+4)]);
  //mcMID is MACRO to extract the particular digit
  product << 4;
  //For next digit, formats are in BCD
}
```

The first fragment is used to generate the multiplicand multiples, referred to as partial product generation (PPG), whereas the second fragment accumulates the generated partial

products, referred to as partial product accumulation (PPA), to produce the final result. Further details can be found in [20].

roccadd is a function that calls the DEC_ADD instruction. The function roccadd takes two BCD inputs and returns the result after BCD addition. A set of MACROs and in-line assembly programs are used to define the function roccadd.

The function that invokes the DEC_ADD instruction is defined as follows.

```
int roccadd(int a, int b, int c) {
    asm __volatile__ (".word0x08A5F617\n");
    return a;
}
```

In the code, “0x08A5F617” is the hex value for instruction custom-0 (DEC_ADD), where Rocket internal integer registers 10 and 11 are used as the source registers and integer register-12 is used as the destination register.

4. Evaluation of the Proposed Framework

In previous work [20], evaluation was conducted by replacing the hardware part with a static dummy function that always returns a fixed value according to the method’s algorithm, called repeatedly from the software as per the method’s design; this allows approximate estimation of the hardware–software interfacing time. However, this approach cannot capture actual computing time, which is highly input-dependent—for example, rounding operations take substantially longer than normal arithmetic, a variation the dummy function cannot represent. Cycle time also depends on hardware design, since arithmetic operations may differ with and without the new hardware.

Method-1 [20] was evaluated using the proposed framework in [21]; here, we instead use this method to evaluate the framework itself. Method-1 with dummy functions is examined on three different platforms using the same 8000-sample dataset for functional verification, covering overflow, underflow, normal, and rounding cases as detailed in [21]. The decNumber C library [5] serves as the software-based baseline. All estimated times and cycle counts are obtained on a Linux 64-bit workstation with an Intel Core i7 processor and 8 GB RAM, the same environment used for all other experiments in this paper. Table 4 shows that Method-1 with dummy functions is 2.15 times faster than the software.

Table 4. Evaluation of Method-1 [20] using the proposed framework.

Process	Avg. Cycles	Speedup
Method-1 using dummy function	1712.68	2.15×
Method-1 with accelerator	2071.85	1.78×
Software [5]	3687.91	1

To assess this result further, Table 5 reports real implementations of the two methods executed in the same environment, with each process repeated 500,000 times and the resulting time averaged over the 8000 samples. A second evaluation, using the Gem-5 [45] simulator with AtomicSimpleCPU in System Call Emulation (SE) mode and RISC-V as the target ISA [32], is summarized in Table 6.

From Tables 4–6, the three independent evaluation approaches yield comparable speedups (2.15×, 1.90×, and 2.06×), while the actual hardware implementation evaluated using the proposed framework achieves a speedup of 1.78×. Although no identical reference implementation is available for direct comparison, all evaluations consistently exhibit an approximate two-fold performance improvement over the software baseline. This con-

sistency supports the validity of the proposed framework for estimating the performance of hardware–software co-designed decimal arithmetic.

Table 5. Evaluation of Method-1 [20] by real implementation.

Process	Avg. Time (s)	Speedup
Method-1 with dummy function	0.03021	$1.90\times$
Software [5]	0.05742	1

Table 6. Evaluation of Method-1 [20] using Gem-5 targeting RISC-V ISA.

Process	Avg. Cycles	Speedup
Method-1 using dummy function	1074.61	$2.06\times$
Software [5]	2213.21	1

We then implement Method-1 as proposed in [20,21] using the actual hardware accelerator and evaluate its performance using the RISC-V RDCYCLE instruction. The hardware–software co-designed implementation achieves a $1.78\times$ speedup over the software-only implementation. Compared with the approximate evaluations, the slightly lower speedup is primarily attributed to the RoCC command-dispatch and command/response (cmd/resp) synchronization overheads incurred by the actual accelerator interface, which are not modeled in the dummy-function or Gem5 evaluations. A detailed cycle-level decomposition of these overheads is beyond the scope of this work and is left for future extension of the framework.

5. Analysis to Design Decimal Multiplication Using the Proposed Framework

Figure 8, adapted from our previous work [21], shows the existing Pareto points for hardware–software co-designed decimal multiplication. To identify additional solutions in the unexplored region, the proposed framework evaluates the individual computational building blocks using 1000 randomly generated decimal inputs. Hardware area is estimated using a commercial synthesis tool [46] and a 90 nm standard-cell library, consistent with the remaining hardware evaluations in this paper.

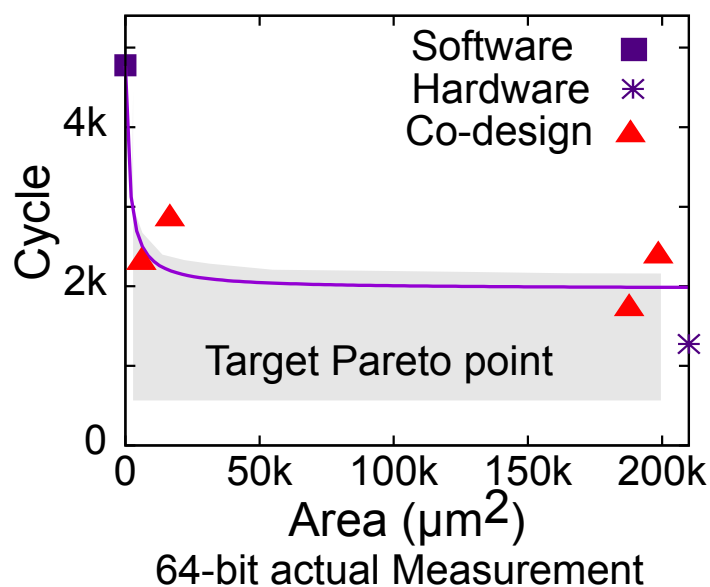


Figure 8. Target Pareto-point region for decimal multiplication. A similar trend is expected for other decimal arithmetic operations supported by the framework.

As illustrated in Figure 9, decimal multiplication may use either binary-based or BCD-based coefficient processing. Both approaches consist primarily of conversion, coefficient multiplication, carry or partial-product processing, and rounding. The following analysis evaluates these building blocks individually to support systematic hardware/software partitioning.

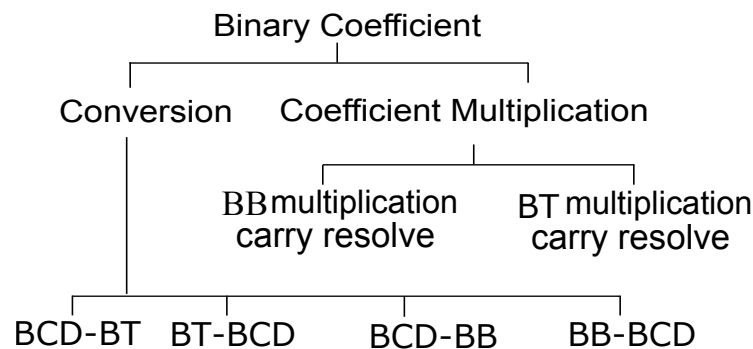


Figure 9. Hardware–software partitioning options for decimal multiplication using binary coefficients. BT and BB denote base-thousand and base-billion, respectively.

5.1. Conversion

Table 7 summarizes the software cycle counts, hardware cycle counts, and synthesized area of the conversion modules supported by the framework. DPD–BCD and BCD–DPD conversions require relatively few cycles because they involve direct encoding transformations; software implementations generally use lookup tables, while the corresponding hardware requires only shallow combinational logic. Similar unpacking and packing units are employed in commercial decimal processors [26,27,47,48].

Table 7. Average cycle counts and hardware area for the supported conversion processes.

Process	Software Cycles	Hardware Cycles	Area (μm^2)
DPD $\rightarrow$ BCD	211.7	36.2	1186.4
BCD $\rightarrow$ DPD	198.2	32.8	697.6
BCD $\rightarrow$ BT	484.9	97.1	5412.2
BCD $\rightarrow$ BB	522.1	78.2	10,752.2
BB $\rightarrow$ BCD	1168.0	43.1	15,911.6
BT $\rightarrow$ BCD	646.4	61.1	8106.0

Abbreviations: BB = base-billion; BT = base-thousand.

Base-thousand and base-billion representations reduce the number of coefficient-processing steps in binary-based multiplication [5]. Base-thousand conversion requires less hardware area, whereas base-billion conversion generally requires fewer hardware cycles because it uses fewer digits. For the reverse conversion to BCD, the hardware modules employ the shift-and-add-3 algorithm, while software implementations use arithmetic decomposition and lookup tables. As shown in Table 7, base-billion-to-BCD conversion is the most expensive software conversion but remains substantially faster when implemented in hardware.

5.2. Coefficient Multiplication

Binary-based multiplication consists of coefficient multiplication followed by carry resolution. As shown in Table 8, the base-billion approach requires fewer cycles because it operates on fewer, wider digits, whereas base-thousand multiplication uses more digits and therefore requires additional multiplication and carry-resolution steps. Base-thousand arithmetic may nevertheless be useful when the available embedded multiplier is narrower than the 32-bit multiplier required by the base-billion implementation.

Table 8. Average cycle counts for software-based binary coefficient multiplication.

Base	Coefficient Multiplication	Carry Resolution	Area	Total Cycles
Billion	261.87	275.94	0	537.82
Thousand	541.28	667.56	0	1208.13

Note: No additional hardware area is required, because both stages are executed in software.

In the base-billion procedure, 32-bit coefficient digits are multiplied to generate higher-precision partial products, which are subsequently normalized into lower-precision base-billion digits [5]. In the base-thousand procedure, each partial product is compared with 999 and divided by 1000 when a carry is present. The larger number of base-thousand digits accounts for its higher cycle count.

BCD-Based Format

BCD-based multiplication consists of partial product generation (PPG) and partial product accumulation (PPA). Table 9 compares four PPG alternatives. Sequential base-billion addition is faster than base-thousand addition because it processes fewer digits, while direct digit multiplication in base-billion requires the highest cycle count. The BCD-CLA implementation provides a competitive cycle count with moderate hardware area and corresponds to the approach used in Method-1 [20].

Table 9. Average cycle counts and area for partial product generation (PPG).

Process	Average Cycles	Area (μm^2)
Base-billion addition	442.17 +	NA
Base-thousand addition	1146.56 +	NA
Digit multiplication in BB	1650.79 +	NA
Using BCD-CLA	461.60	5459.00

+ Conversion cycles are not included.

Table 10 compares sequential and parallel accumulation alternatives. Software accumulation avoids hardware overhead but requires more cycles, whereas parallel accumulation substantially reduces latency at the cost of a large hardware area. The framework implements the area-efficient adder-tree-based parallel structure for evaluating this trade-off.

Table 10. Average cycle counts and area for partial product accumulation (PPA).

Process	Average Cycles	Area (μm^2)
Parallel selection and accumulation	103	87,729
Software using CLA	581	0
Parallel selection with CSA [41]	NA	94,268
Sign-digit adder tree [49]	NA	93,794

5.3. Rounding and Special-Case Handling

The final stages of decimal multiplication are rounding and special-case handling. The framework provides both a software rounding routine and an IEEE-compliant hardware rounding module. The software routine currently accepts BCD inputs, whereas the hardware module supports BCD, base-thousand, and base-billion representations. The hardware rounding module occupies $11,288.8 \mu\text{m}^2$ and requires an average of 39.7 cycles.

Special-case handling checks conditions such as NaN, infinity, invalid operands, infinity multiplied by zero, and related IEEE 754 exceptions. In the current framework, these checks are performed using software macros, while a fully hardware-based implementation would require operation-specific checking logic. Because the same software-based rounding

and special-case stages are applied to all compared partitioning methods, they are analyzed separately from the partition-dependent conversion and coefficient-processing stages. Their costs are incorporated when reporting end-to-end results.

6. Design Pareto Point for Decimal Multiplication Using the Proposed Framework

This section integrates the results from Section 5 to achieve the most desirable hardware–software co-design Pareto point for decimal multiplication. First, the framework evaluates existing software-based multiplication [5] to determine which functions or operations should have dedicated hardware and which should remain in software to reduce hardware overhead and computation time.

The primary steps of a software-based multiplication of [5] are as follows:

The multiplication process involves checking operand specialties, determining the sign and initial exponent, and converting coefficients to base-billion format. The base-billion digits are multiplied to produce partial products with carries, which are then resolved to obtain the final base-billion product. This result is converted back to BCD for rounding, and finally verified for exceptions, special cases, and rounding correctness.

Figure 10 shows the average cycles for software-based decimal multiplication, where coefficient multiplication dominates (gray), including DPD to base-billion conversion, coefficient multiplication, carry resolution, and base-billion to BCD conversion. The two conversion steps, along with rounding, require the most cycles. Tables 11 and 12 summarize trade-offs between binary- and BCD-based approaches. In the binary case, conversions are replaced by hardware components, and instruction cycles are added to compute totals. For BCD-based multiplication, two Pareto points are observed, Method-1 [21] and Method-3 (parallel BCD) [21], while binary-based multiplication also yields two Pareto points, shown in Table 12.

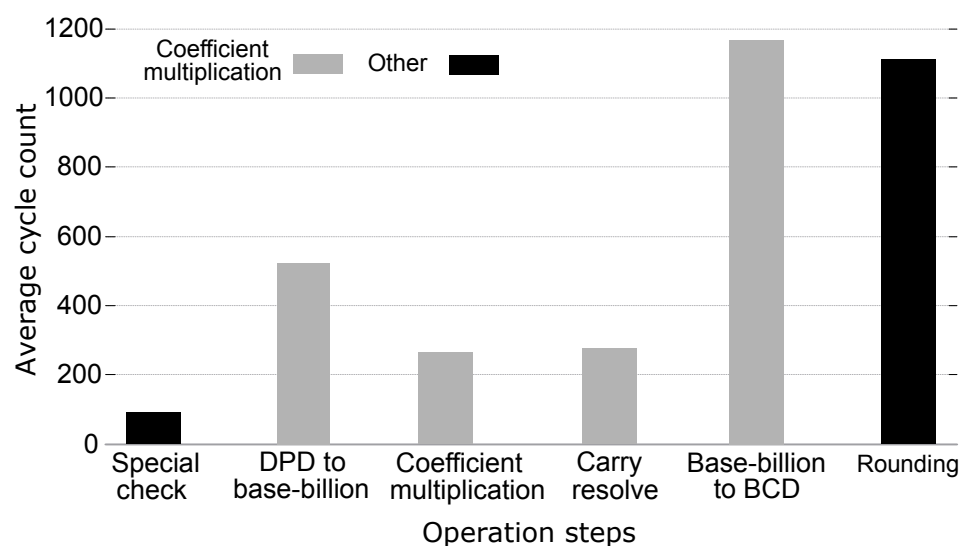


Figure 10. Average number of cycles required for the individual stages of software-based decimal multiplication.

Table 11. Overall trade-off for binary-based hardware–software co-design.

Operation	Avg. Cycles	Area (μm^2)
Base-thousand	1366 +	13,519
Base-billion	658 +	26,663

+ Rounding and special check are not included.

Table 12. Overall trade-off for BCD-based co-design.

Operation	Avg. Cycles	Area (μm^2)
Software with BCD-CLA	1042 +	5459
Software with parallel accumulation	564 +	87,729

+ Rounding and special check are not included.

New Pareto Point (Method-A)

Figure 11 summarizes BCD and binary-based multiplication Pareto points. Along with Method-1 and Method-3 [20], we identify a new base-billion solution, Method-A, where multiplication and carry resolution run in software, while DPD BCD conversions use hardware. The detailed flow is shown in Figure 12, with hardware in gray and software in white.

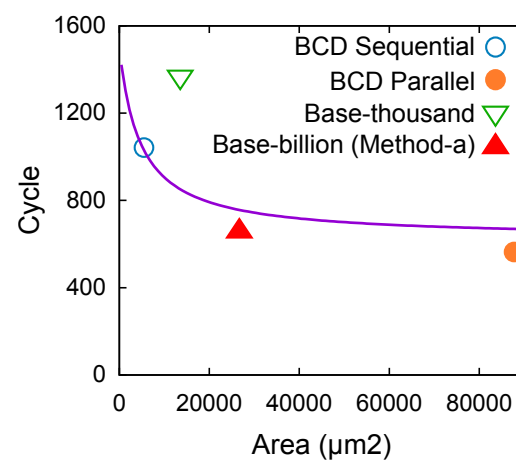


Figure 11. Overall analysis for identifying the Pareto-optimal decimal multiplication hardware–software co-design solution.

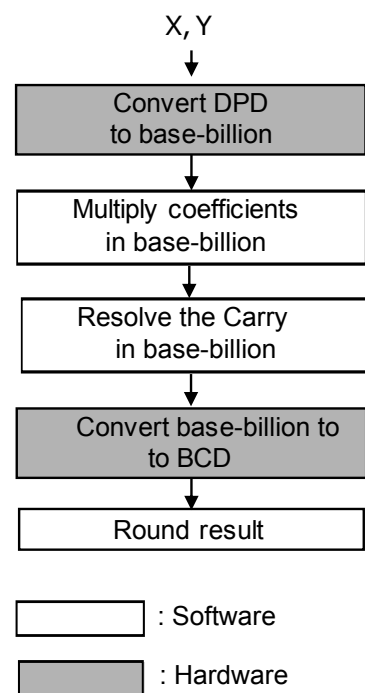


Figure 12. Method-A: Newly identified Pareto-optimal solution for decimal multiplication hardware–software co-design.

For base-thousand multiplication, conversions require less area but higher cycles. Although not included as a Pareto point in Figure 11, it remains suitable for cases where embedded multipliers are below 32-bit, as in many FPGAs.

7. Evaluation of Decimal Multiplication Hardware–Software Co-Design

This section evaluates the decimal multiplication methods proposed in the previous work [21], where the decimal multiplication hardware–software co-design process is not evaluated in the context of multiple cores with benchmark decimal application. To compare the overall evaluation of the methods, the area of the accelerator is first calculated and then compared to the area of the entire Rocket chip [33]. Table 13 shows the overall area overhead, where the accelerator area is compared with the total chip area. Table 14 shows the breakdown of the chip area. The area of the components of the Rocket chip (cache, core and FPU) is evaluated. The cache area overhead is evaluated by CACTI [50], while the area overhead of the other units is evaluated by CAD tools [46]. The developed CHISEL [51] code is converted to a corresponding Verilog HDL code to evaluate the area overhead ratio. In Table 13, the area for the proposed methods represents the total accelerator area including the components of the corresponding methods. The “ratio” column of the table compares the ratio of the area of the proposed methods with the total chip area (cache, core, and FPU).

Table 13. Overall area overhead comparison of the proposed methods.

Method	Area (μm^2)	Ratio
Method-1 [21]	28,095	2.09
Method-2 [21]	213,519	15.88
Method-3 [21]	226,526	16.85
Method-4 [21]	44,366	3.30
Rocket Chip from Table 14	1,344,428	-

Table 14. Area overhead of Rocket chip components.

Chip Components	Area (μm^2)
I\$	309,214
D\$	452,452
FPU	213,519
Rocket Core	364,321
Total	1,344,428

After that, the overall speed improvement is calculated using a benchmark application. The benchmark application is the Telco benchmark (a telephone company billing application) [52] developed by IBM. The benchmark provides an example of an IEEE 754-2008-compliant set of DFP operations and it can be executed on any computer. The benchmark is based on the decNumber package, [5], an implementation of the IEEE 754 decimal specifications in ANSI C. This application uses call duration, rates, and taxes to calculate the bill. To perform these calculations, all basic decimal operations are performed with rounding. In this evaluation environment, only decimal multiplication operations are used, following a hardware–software co-design approach. The decimal software library performs the rest of the operation [5]. Since the benchmark application is not only based on decimal multiplication, the results in Tables 15–18 reflect only the speedup due to the

hardware–software co-design-based multiplication operation using the proposed methods. Table 15 shows that the speed improvement follows a similar trend as the performance of the proposed methods. Finally, these two tables are summed up in Table 16 to evaluate the trade-off comparison. Table 16 also shows that the result of the evaluation follows a similar trend as the performance of the proposed methods.

Table 15. Overall delay overhead using Telco benchmark application.

Method	Number of Cycles	Speedup Ratio
Software [5]	194,592	1.00
Method-1 [21]	149,232	1.30
Method-2 [21]	111,217	1.75
Method-3 [21]	156,195	1.25
Method-4 [21]	162,160	1.20

Table 16. Overall area–delay trade-off of the proposed methods.

Proposed Method	Area Overhead	Speed Improvement
Method-1 [21]	2.09	1.30
Method-2 [21]	15.88	1.75
Method-3 [21]	16.85	1.25
Method-4 [21]	3.30	1.20

Table 17. Performance of the Telco benchmark using software parallelization with one and two CPU threads.

Threads	Execution Time (ms)	Speedup
1 thread	6.67	1
2 threads	5.79	1.15

Table 18. Comparison of software parallelization and hardware acceleration for the IBM Telco benchmark.

Method	Speedup	Area Overhead (%)
Software parallelization (2 threads)	1.15	100
Method-1 [21]	1.30	2.09
Method-2 [21]	1.75	15.88
Method-3 [21]	1.24	16.85
Method-4 [21]	1.20	3.30

Note: The 100% area overhead for software parallelization represents the additional processor core assumed for two-thread software execution. It is included only as a software-level reference for comparison with the proposed hardware–software co-design-based methods and should not be interpreted as a direct architectural comparison.

Finally, we compare two alternative approaches for accelerating decimal applications: software parallelization using multiple CPU cores and hardware acceleration using the proposed decimal accelerator. It should be noted that these two evaluations are performed on different experimental platforms. The proposed hardware accelerator is evaluated using the single-core RISC-V Rocket-chip framework developed in this work, whereas the software parallelization experiment is performed using OpenMP on a multicore x86 platform. Therefore, this comparison is intended only as a software-level reference for the same decimal benchmark rather than a direct platform-to-platform performance comparison.

The proposed evaluation framework currently targets a single-core RISC-V Rocket processor and therefore does not support multicore hardware evaluation. Consequently, rather than evaluating the proposed framework itself in a multicore environment, we execute the IBM Telco benchmark using OpenMP on a multicore x86 platform as a software-level reference. Because the hardware accelerator and software parallelization results are obtained on different processor architectures, the comparison should be interpreted only as an indirect assessment of the potential benefits of software parallelization versus hardware acceleration for the same decimal workload.

Table 17 summarizes the execution times obtained using one and two CPU threads. The results show that the Telco benchmark achieves a $1.15\times$ speedup when executed with two threads compared with single-thread execution.

The results are summarized in Table 18. The proposed hardware–software co-design methods achieve speedups ranging from $1.20\times$ to $1.75\times$ with only 2–17% accelerator area overhead, whereas software parallelization using two CPU threads achieves a $1.15\times$ speedup on the multicore x86 platform for the same benchmark workload. Since these results are obtained on different hardware platforms, they should not be interpreted as a direct quantitative comparison. Instead, they provide an indirect assessment of the trade-off between software parallelization and hardware acceleration for decimal applications. The proposed framework itself remains a single-core RISC-V evaluation framework and no claim regarding multicore capability or multicore efficiency is made.

8. Conclusions

In this study, we proposed an integrated evaluation framework for hardware–software co-design of decimal arithmetic using the RISC-V ecosystem, a decimal library, and a verification database. The framework supports both functional and performance verification, validated through developed test programs. The proposed framework extends our previous work by enabling exhaustive hardware/software partition analysis at the building-block level, allowing systematic exploration of the decimal arithmetic design space. A hardware–software co-design-based decimal multiplier was implemented to validate prior work [20], and the proposed analysis identified a new Pareto point (Method-A) for decimal multiplication. Furthermore, a software-level comparison between software parallelization and hardware acceleration was conducted using the IBM Telco benchmark to provide additional insight into alternative acceleration strategies. This comparison serves only as a software-level reference and does not represent multicore evaluation of the proposed single-core RISC-V framework.

Author Contributions: Conceptualization, R.-u.-h.M. and M.I.; methodology, R.-u.-h.M. and M.I.; software, R.-u.-h.M.; validation, R.-u.-h.M. and M.I.; formal analysis, R.-u.-h.M.; investigation, R.-u.-h.M.; resources, M.I.; data curation, R.-u.-h.M.; writing—original draft preparation, R.-u.-h.M.; writing—review and editing, R.-u.-h.M. and M.I.; visualization, R.-u.-h.M.; supervision, M.I.; project administration, R.-u.-h.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All data are publicly available. The proposed framework is fully open source. The source code, hardware designs, software implementations, interfaces, and continuously updated project resources are publicly available in the following GitHub repositories: decimal multiplication (128-bit implementation): <https://github.com/riazcseiu/DecimalMultiplication>; decimal multiplication hardware accelerator implementation: <https://github.com/riazcseiu/decimalmultiplication128-bit>; modified Rocket-Chip platform: <https://github.com/riazcseiu/rocket-chip>; decimal multiplication accelerator integration for Rocket-Chip: <https://github.com/riazcseiu/decimalMultiplicationAccelerator>; hardware–software co-design framework (64-bit decimal arithmetic): <https://github.com/riazcseiu/DecimalHardwareSoftware> (accessed on 5 February 2025).

The repositories are actively maintained and updated to reflect the latest development of the proposed framework.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Cowlishaw, M.F. Decimal Floating-Point: Algorithm for Computers. In Proceedings of the IEEE International Symposium on Computer Arithmetic, Santiago de Compostela, Spain, 15–18 June 2003; pp. 104–111. [\[CrossRef\]](#)
2. *IEEE Std 754-2008*; IEEE Standard for Floating-Point Arithmetic. IEEE: New York, NY, USA, 2008. [\[CrossRef\]](#)
3. Deitel, P.; Deitel, H.M. *Java for Programmers: With Generative AI*; Pearson Education: Hoboken, NJ, USA, 2025.
4. Oracle America, Inc. Java Class BigDecimal. 2015. Available online: <https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html> (accessed on 1 June 2024).
5. Cowlishaw, M. The decNumber C Library, Version 3.68. 2010. Available online: <http://speleotrove.com/decimal/decnumber.html> (accessed on 11 May 2024).
6. Intel Decimal Floating-Point Math Library for IEEE 754-2008. 2011. Available online: https://bit.ly/INTEL_DFP (accessed on 16 May 2024).
7. Cornea, M.; Harrison, J.; Anderson, C.; Tak, P.; Tang, P. A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format. *IEEE Trans. Comput.* **2009**, *58*, 148–162. [\[CrossRef\]](#)
8. Libe, G. Financial Software Development Lifecycle: Junior Developer Perspectives in Java Using Vaadin Framework. 2025. Available online: <https://urn.fi/URN:NBN:fi:amk-2025092325005> (accessed on 5 February 2025).
9. Schulte, M.J.; Lindberg, N.; Laxminarain, A. Performance Evaluation of Decimal Floating-Point Arithmetic. In Proceedings of the 6th IBM Austin Center for Advanced Studies Conference, Austin, TX, USA, 18–20 October 2005; pp. 1–4.
10. Yoshida, T.; Maruyama, T.; Akizuki, Y.; Kan, R.; Kiyota, N.; Ikenishi, K.; Itou, S.; Watahiki, T.; Okano, H. SPARC64 X: Fujitsu's New-Generation 16-Core Processor for Unix servers. *IEEE Micro* **2013**, *33*, 16–24. [\[CrossRef\]](#)
11. Zhu, M.; Jiang, Y.; Yang, M.; Chen, T. On high-performance parallel decimal fixed-point multiplier designs. *Comput. Electr. Eng.* **2014**, *40*, 2126–2138. [\[CrossRef\]](#)
12. Gorgin, S.; Jaberipur, G. Fully redundant decimal arithmetic. In Proceedings of the IEEE International Symposium on Computer Arithmetic, Portland, OR, USA, 9–10 June 2009; pp. 145–152. [\[CrossRef\]](#)
13. Gorgin, S.; Jaberipur, G.; Asl, R.H. Efficient ASIC and FPGA Implementation of Binary-Coded Decimal Digit Multipliers. *Circuits Syst. Signal Process.* **2014**, *33*, 3883–3899. [\[CrossRef\]](#)
14. Erle, M.A.; Schulte, M.J.; Linebarger, J.M. Potential speedup using decimal floating-point hardware. In Proceedings of the Asilomar Conference on Signals Systems and Computers, Pacific Grove, CA, USA, 3–6 November 2002; pp. 1073–1077. [\[CrossRef\]](#)
15. Erle, M.A.; Schulte, M.J.; Hickmann, B.J. Decimal Floating-Point Multiplication Via Carry-Save Addition. In Proceedings of the IEEE International Symposium on Computer Arithmetic, Montpellier, France, 25–27 June 2007; pp. 46–55. [\[CrossRef\]](#)
16. Riaz-ul-haque, M.; Ali, M. Design of low power decimal logarithmic converter. In Proceedings of the International Conference on Electrical Computer and Communication Engineering, Cox's Bazar, Bangladesh, 16–18 February 2017; pp. 152–157. [\[CrossRef\]](#)
17. Erle, M.A.; Hickmann, B.J.; Schulte, M.J. Decimal Floating-Point Multiplication. *IEEE Trans. Comput.* **2009**, *58*, 902–916. [\[CrossRef\]](#)
18. Wang, L.K.; Erle, M.A.; Tsen, C.; Schwarz, E.M.; Schulte, M.J. A survey of hardware designs for decimal arithmetic. *IBM J. Res. Dev.* **2010**, *54*, 8.1–8.15. [\[CrossRef\]](#)
19. Mashayekhi, N.; Jaberipur, G.; Reshadinezhad, M.R.; Moghimi, S. Design of energy-efficient and high-speed hybrid decimal adder. *J. Supercomput.* **2025**, *81*, 482. [\[CrossRef\]](#)
20. Riaz-ul-haque, M.; Shintani, M.; Inoue, M. Decimal Multiplication Using Combination of Software and Hardware. In Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems, Chengdu, China, 26–30 October 2018; pp. 239–242. [\[CrossRef\]](#)
21. Riaz-ul-haque, M.; Michihiro, S.; Inoue, M. Hardware–Software Co-Design for Decimal Multiplication. *Computers* **2021**, *10*, 17. [\[CrossRef\]](#)
22. Gladshstein, M. Quantum-dot cellular automata serial decimal digit multiplier. *J. Comput. Electron.* **2025**, *24*, 54. [\[CrossRef\]](#)
23. Telagam, N.; Ajitha, D.; Prabin, S.; Srikanth, B.; Kandasamy, N. A Novel Area Efficient and High-Speed BCD Adder Using CFA and Multiplexer in Quantum-Dot Cellular Automata. *IEEJ Trans. Electr. Electron. Eng.* **2025**, *21*, 226–234. [\[CrossRef\]](#)
24. Asawalertsak, W.; Sarochawikasit, R.; Prechaprapranwong, P. Quantum Arithmetic for Real-Number 2's Complement Multiplication. In Proceedings of the 2025 International Conference on Quantum Communications, Networking, and Computing (QCNC); IEEE: Piscataway, NJ, USA, 2025; pp. 195–199. [\[CrossRef\]](#)
25. Akhtar, M.S. Quantum-Inspired Neural Radiative Transfer (QINRT): A Multi-Scale Computational Framework for Next-Generation Climate Intelligence. *AppliedMath* **2025**, *5*, 145. [\[CrossRef\]](#)

26. Schwarz, E.M.; Kaepernick, J.S.; Cowlshaw, M.F. The IBM z900 Decimal Arithmetic Unit. In Proceedings of the Asilomar Conference on Signals Systems and Computers, Pacific Grove, CA, USA, 4–7 November 2001; pp. 1335–1339. [\[CrossRef\]](#)
27. Schwarz, E.M.; Kapernick, J.S.; Cowlshaw, M.F. Decimal floating-point support on the IBM System z10 processor. *IBM J. Res. Dev.* **2009**, *53*, 4.1–4.10. [\[CrossRef\]](#)
28. Carlough, S.; Collura, A.; Mueller, S.; Kroener, M. The IBM zEnterprise-196 Decimal Floating-Point Accelerator. In Proceedings of the IEEE International Symposium on Computer Arithmetic, Tübingen, Germany, 25–27 July 2011; pp. 139–146. [\[CrossRef\]](#)
29. Brisebarre, N.; Ercegovac, M.D. Implementing Decimal Floating Point Arithmetic Through Binary: Some Suggestions. In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, Rennes, France, 7–9 July 2010; pp. 317–320. [\[CrossRef\]](#)
30. Tsen, N.; Gonzalez-Navarro, S.; Schulte, M.; Hickmann, B.; Compton, K. A Combined Decimal and Binary Floating-Point Multiplier. In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, Boston, MA, USA, 7–9 July 2009; pp. 8–15. [\[CrossRef\]](#)
31. Tsen, C.; Schulte, M.; Gonzalez-Navarro, S. Hardware Design of a Binary Integer Decimal-Based IEEE 754 Rounding Unit. In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, Montréal, QC, Canada, 9–11 July 2007; pp. 115–121. [\[CrossRef\]](#)
32. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. 2011. Available online: <https://riscv.org/> (accessed on 16 May 2024).
33. The Rocket Chip Generator. 2011. Available online: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html> (accessed on 16 May 2024).
34. Riaz-ul-haque, M.; Shintani, M.; Inoue, M. Cycle-Accurate Evaluation of Software-Hardware Co-Design of Decimal Computation in RISC-V Ecosystem. In Proceedings of the IEEE International System on Chip Conference, Singapore, 3–6 September 2019; pp. 412–417. [\[CrossRef\]](#)
35. *Standard 754-2019*; IEEE Standard for Floating-Point Arithmetic; Revision of IEEE 754-2008. IEEE: New York, NY, USA, 2019; pp. 1–84.
36. Anderson, M.J.; Tsen, C. Performance analysis of decimal floating-point libraries and its impact on decimal hardware and software solutions. In Proceedings of the IEEE International Conference on Computer Design, Lake Tahoe, CA, USA, 4–7 October 2009; pp. 465–471. [\[CrossRef\]](#)
37. Cowlshaw, M. Densely packed decimal encoding. *IEEE Proc.-Comput. Digit. Tech.* **2002**, *149*, 102–104. [\[CrossRef\]](#)
38. Cornea, M. IEEE 754-2008 Decimal Floating-Point for Intel Architecture Processors. In Proceedings of the IEEE International Symposium on Computer Arithmetic, Kyoto, Japan, 10–12 June 2019; pp. 325–328. [\[CrossRef\]](#)
39. Vazquez, A.; Antelo, E.; Montusshi, P. Improved design of high-performance parallel decimal multipliers. *IEEE Trans. Comput.* **2010**, *59*, 679–693. [\[CrossRef\]](#)
40. Cui, X.; Lombardi, F. A Parallel Decimal Multiplier Using Hybrid Binary Coded Decimal (BCD) Codes. In Proceedings of the IEEE International Symposium on Computer Arithmetic, Silicon Valley, CA, USA, 10–13 July 2016; pp. 150–155. [\[CrossRef\]](#)
41. Cui, X.; Dong, W.; Liu, W.; Swartzlander, E.E.; Lombardi, F. High Performance Parallel Decimal Multipliers Using Hybrid BCD Codes. *IEEE Trans. Comput.* **2017**, *66*, 1994–2004. [\[CrossRef\]](#)
42. Vazquez, A.; Antelo, E.; Bruguera, J.D. Fast Radix-10 Multiplication Using Redundant BCD Codes. *IEEE Trans. Comput.* **2014**, *63*, 1902–1914. [\[CrossRef\]](#)
43. Fahmy, H.A.H.; ElDeeb, T.; Hassan, M.Y.; Farouk, Y.; Eissa, R.R. Decimal Floating Point for future processors. In Proceedings of the International Conference on Microelectronics, Cairo, Egypt, 19–22 December 2010; pp. 1–4. [\[CrossRef\]](#)
44. Rigge, P.; Nikolić, B. Designing Digital Signal Processors with RocketChip. In Proceedings of the Workshop on Computer Architecture Research with RISC-V, Los Angeles, CA, USA, 2 June 2018; pp. 1–6.
45. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The Gem5 Simulator. *ACM SIGARCH Comput. Archit. News* **2011**, *39*, 1–7. [\[CrossRef\]](#)
46. Synopsys, Inc. *Design Compiler User Guide*, version I-2013.06; Synopsys, Inc.: Sunnyvale, CA, USA, 2013.
47. Eisen, L.; Ward, J.W.; Test, H.; Mading, N.; Leenstra, J.; Mueller, S.M.; Jacobi, C.; Preiss, J.; Schwarz, E.M.; Carlough, S.R. IBM POWER6 accelerators: VMX and DFU. *IBM J. Res. Dev.* **2007**, *51*, 663–683. [\[CrossRef\]](#)
48. Duale, A.Y.; Decker, M.H.; Zipperer, H.G.; Aharoni, M.; Bohizic, T.J. Decimal floating-point in z9: An implementation and testing perspective. *IBM J. Res. Dev.* **2007**, *51*, 217–227. [\[CrossRef\]](#)
49. Gorgin, S.; Jaberipur, G. Sign-Magnitude Encoding for Efficient VLSI Realization of Decimal Multiplication. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 75–86. [\[CrossRef\]](#)
50. Muralimanohar, N.; Balasubramonian, R.; Jouppi, N.P. CACTI 6.0: A tool to model large caches. *HP Lab.* **2009**, *1*, 1–24.

51. Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A. Chisel: Constructing hardware in a Scala embedded language. In Proceedings of the IEEE/ACM Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012; pp. 465–471. [[CrossRef](#)]
52. IBM Corporation. The Telco Benchmark. 2008. Available online: <http://speleotrove.com/decimal/telco.html> (accessed on 24 January 2024).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.