Redundant-Digit Floating-Point Addition Scheme Based on a Stored Rounding Value

Ghassem Jaberipur, Behrooz Parhami, *Fellow*, *IEEE*, and Saeid Gorgin

Abstract—Due to the widespread use and inherent complexity of floating-point addition, much effort has been devoted to its speedup via algorithmic and circuit techniques. We propose a new redundant-digit representation for floating-point numbers that leads to computation speedup in two ways: 1) Reducing the per-operation latency when multiple floating-point additions are performed before result conversion to nonredundant format and 2) Removing the addition associated with rounding. While the first of these advantages is offered by other redundant representations, the second one is unique to our approach, which replaces the power- and area-intensive rounding addition by low-latency insertion of a rounding two-valued digit, or twit, in a position normally assigned to a redundant twit within the redundant-digit format. Instead of conventional sign-magnitude representation, we use a sign-embedded encoding that leads to lower hardware redundancy, and thus, reduced power dissipation. While our intermediate redundant representations remain incompatible with the IEEE 754-2008 standard, many application-specific systems, such as those in DSP and graphics domains, can benefit from our designs. Description of our radix-16 redundant representation and its addition algorithm is followed by the architecture of a floating-point adder based on this representation. Detailed circuit designs are provided for many of the adder's critical subfunctions. Simulation and synthesis based on a $0.13\ \mu\mathrm{m}$ CMOS standard process show a latency reduction of 15 percent or better, and both area and power savings of around 58 percent, compared with the best designs reported in the literature.

Index Terms—Adder/subtractor, computer arithmetic, floating point, redundant format, rounding, signed-digit number system.

✦

---

1   INTRODUCTION

$\mathbf{F}$LOATING-POINT addition is believed to be the most frequent computer arithmetic operation. Intricacies of floating-point number representation make floating-point addition inherently more complex than integer addition. Thus, methods for speeding up floating-point addition are of utmost importance. A floating-point number is convention-ally composed of a sign bit, an exponent, and a significand [1]. Since the ANSI/IEEE standard for binary floating-point arithmetic [2] (IEEE 754 for short) was introduced in 1985, virtually all implementations have adhered to its representa-tion formats, even when they do not follow the full provisions of the standard, or its revised version, IEEE 754-2008 [3]. Implementations in this category include systems for digital signal/image processing [4] and computer graphics [5]. The short (long), also known as 32-bit or single (64-bit or double), normalized standard format incorporates a sign bit s, a biased excess-127 (excess-1023) representation e for the exponent, and a significand μ composed of 23 (52) bits to the right of the binary point that has a hidden 1 to its left. A

short (long) floating-point number $f = (s, e, \mu)$ represents the real value $(-1)^s 2^{e-\mathrm{bias}}\, 1.\mu$, where $1.\mu$ stands for $1 + \mu \times 2^{-23}(2^{-52})$ and e is an unsigned 8-bit (11-bit) integer. Besides "ordinary" floating-point numbers just described, some special values (such as $\pm 0, \pm\infty$, and NaN) have unique codes assigned to them.

The operation of floating-point addition/subtraction consists of several steps, as outlined in Algorithm 1. Description of each step is followed, in square brackets, by the order of its worst-case latency, assuming a fast implementation.

Algorithm 1. Floating-point (FP) addition/subtraction
Inputs: FP operands $f_1 = (s_1, \varepsilon, \mu_1)$ and $f_2 = (s_2, \eta, \mu_2)$,
      op ($+$ or $-$), and rounding mode
Output: FP result $f_3 = f_1 \pm f_2$
  1. *Exponent difference*: Compute $\Delta = \varepsilon - \eta$ and determine the operand with smaller exponent $[\Omega(\log(\mathrm{width}(e)))]$.
  2. *Alignment shift*: Right-shift the significand of the number having the smaller exponent by an amount derived from Δ; for exponent base of 2, the right-shift amount is $\min(|\Delta|, \mathrm{width}(\mu))$ bits $[\Omega(\log(\mathrm{width}(\mu)))]$.
  3. *Add-or-subtract decision*: Determine the actual operation to be performed on the significands based on the operand signs and the specified operation ($+$ or $-$); swap the operands, if necessary [O(1), possibly overlapped with the previous steps].
  4. *Sign and significand derivation*: Perform the actual operation determined in step 3, thus obtaining the sign and significand of the result $[\Omega(\log(\mathrm{width}(\mu)))]$.
  5. *Leading 0 digits*: Detect the number of leading 0 digits in the result $[\Omega(\log(\mathrm{width}(\mu)))]$.
  6. *Postnormalization*: Normalize the result, if nonzero, via

- *G. Jaberipur is with the Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran 19839-63113, Iran, and the School of Computer Science, Institute for Research in Fundamental Sciences, Tehran, Iran.*
- *B. Parhami is with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106-9560.*
- *S. Gorgin is with the Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran 19839-63113, Iran.*

shifting, such that there is a single 1 (the new hidden 1) to the left of the binary point $[\Omega(\log(\mathrm{width}(\mu))]$.

7. *Exponent adjustment*: Adjust the result exponent to compensate for the shift in step 6 $[\Omega(\log(\mathrm{width}(e))$, but overlapped with step 6].

8. *Rounding*: Use information provided by the extra bits maintained in the internal format to round the result and adjust its exponent, if needed $[\Omega(\log(\mathrm{width}(\mu))]$.

Each of the steps above may consist of a number of simpler substeps. To minimize the addition latency, clever methods have been developed to allow concurrent execution of (sub)steps in Algorithm 1 (e.g., [6], [7], [8], [9]). Rounding, in particular, is problematic, because it could introduce a second power/area-intensive word-width addition (actually, an incrementation). Pipelining of steps in floating-point operations has also reduced the average latency per operation. Further per-addition speedup is possible with redundant representation of intermediate results, thus allowing carry-free addition, provided that a series of floating-point operations is performed before there is a need to store a result in memory or to send it to an output device. The rounding-packet forwarding scheme of Nielsen et al. [7] keeps values in binary signed-digit (BSD) form, resulting in double-size registers (in view of the 2 bits required in each radix-2 position). The latest relevant work, by Fahmy and Flynn [10], uses sign-magnitude addition with redundant high-radix signed digits, that leads to lower redundancy. In the latter work, rounding is done concurrently with the exponent comparison of the next floating-point addition, or an unrounded value is used in a subsequent operation which also receives a "rounding value" to be included in the significand addition.

In this paper, we follow the approach of Fahmy and Flynn, but with our particular redundant encoding of digits, which significantly reduces the active hardware redundancy through sign-embedding and obviates one of the two challenges cited in [10]; namely, recognition and transformation of insignificant digits in the process of leading-nonzero-digit detection. We use a redundant representation dubbed stored-unibit-transfer, or SUT [11], where the encoding provides room in the transfer part of the least-significant position of the result for the three possible rounding values -1, 0, and 1. Other features of our approach include:

- redundant-digit internal format with embedded sign,
- carry-free addition/subtraction,
- simple detection of the leading nonzero digit, and
- elimination of rounding increment operation and postrounding exponent adjustment.

Here is a roadmap for the rest of this paper. Section 2 contains an overview of the state of the art in the design of floating-point adders, including one by Fahmy [12]. In Section 3, we review the SUT encoding of a class of redundant numbers and present the associated carry-free adder/subtractor that uses only standard full/half-adders. Sections 4-7 are devoted to key design considerations of our dual-path floating-point adder: redundant internal number format, path separation, extra (guard, round, and sticky) digits, and rounding decision. Analytical and simulation

Fig. 1. Dual-path implementation of Algorithm 1. Box heights are meant to reflect circuit latencies.

results, presented in Section 8, are used to compare our designs with those of [12]. Conclusions and directions for further work appear in Section 9. Drawbacks of the work in [12] in way of possible bad rounding positions, along with the difficulty of adherence to IEEE 754-2008 standard with either SUT or maximally redundant signed-digit number representation, are discussed in the Appendix.

2 FLOATING-POINT ADDITION

Design of hardware floating-point units has a long history, dating back to early digital computers that were used primarily for scientific computations [13]. Initial efforts in providing high performance in floating-point units were impeded by the exorbitant cost of hardware. This forced serialization of potentially parallel steps to allow hardware reduction and sharing, even in top-of-the-line supercomputers [14]. As hardware cost decreased, an array of innovative designs began to emerge. Once the more or less straightforward performance enhancement schemes were exhausted, replication of units and other hardware-intensive methods were employed to squeeze out incremental gains. The state of the art in floating-point adder design uses dual data paths, as depicted in Fig. 1, to separate the relatively slow alignment and postnormalization shifts (Steps 2 and 6 of Algorithm 1) into different paths. This parallelization, based on the exponent difference, is credited to Farmwald [6]. Others have refined the path separation criteria. For example, Seidel and Even [15] use both the exponent difference and the actual operation for this purpose.

The innovations cited above notwithstanding, there is still room for fine-tuning and improvements in speed, latency-area tradeoffs, and energy dissipation given the following challenges in high-speed floating-point adders/subtractors:

- The prevalent sign-magnitude encoding leads to a more complex significand addition process than 1's-or 2's-complement format. Some techniques meant to speed up the addition of sign-magnitude significands (e.g., [16]) entail additional chip-area and power overheads.

- Postnormalization via counting the leading (non)-zero digits is a log-latency operation at best. However, the count of leading 0/1 digits can be obtained concurrently with addition, perhaps by deriving an approximate count quickly and fine-tuning the result at the end [7].
- Rounding to nearest may require an incrementation and possible exponent adjustment. With some extra hardware (e.g., the parallel-prefix adder of [15]), both the normal and an incremented result can be computed in parallel, thus allowing rapid selection of the rounded value.

Many computations involve a sequence of arithmetic operations, without a need to store a result in memory. Converting an operand loaded from memory to an intermediate redundant encoding is possible in a small constant time, independent of operand width. Carry-free addition/subtraction of redundant operands reduces the latency of each operation, at the cost of wider register files to accommodate the redundant representation. However, the final result must be converted back to nonredundant form before it is stored in memory. This process is at best a logarithmic-time operation, but its latency is more than compensated for by the per-add savings, compounded over many redundant addition levels. Also, the conversion delay can be hidden by the memory store operation.

The redundant floating-point addition scheme of Nielsen et al. [7] employs double-size registers to accommodate the intermediate BSD results. Using radix-16 signed-digit representation, Fahmy and Flynn [10] introduced a floating-point adder design with fewer redundant bits. The following paragraphs highlight their approach and its challenges:

- **Redundant number system**: A maximally redundant radix-16 signed-digit (MRSD) encoding is used for intermediate values. Signed digits in $[-15, 15]$ are represented by a 5-bit 2's-complement encoding, which is not faithful because it allows the undesired value -16, though this value is not generated by the addition algorithm. The claim that narrower digit sets ($[-14, 14]$ or narrower) may complicate the design is used to justify the MRSD choice [12]. However, maximal redundancy allows for cancellation of several nonzero digits, beginning with 1 (-1) and followed by a chain of -15 (15) digits to the right. This property complicates leading nonzero digit detection. Note that for narrower digit sets, cancellation can occur only for the leading 1 (-1).
- **Path separation**: "The cancellation path is used only in the case of an effective subtraction with an exponent difference of zero or an effective subtraction with an exponent difference of one and a cancellation of some of the leading digits occurring in the result. In all other cases, the far path is used." [10].
- **Rounding:** Another challenge is the handling of the rounding increment/decrement operation. Assimilation of the increment/decrement is postponed and is performed concurrently with the exponent difference computation of the next addition. The problem here is that the rounding position (i.e., the exact binary position for inserting the rounding value),

TABLE 1
Comparison of Different Implementations of Algorithm 1

Floating-point adder design	Number of steps in Algorithm 1 with logarithmic latency on:				Additional active hardware	
	width(μ)	width(e)	$\lceil$width(μ)/$h\rceil$	h	Carry accel's	Full units replicated
Single-path	4	2	0	0	7	≥ 0
Dual-path	3	2	0	0	8	≥ 1
MRSD-FP	0	Hidden	2	2	18	14
SUT-FP	0	1	2	1	6	3

should be determined based on what the rounding position would be after converting to nonredundant IEEE 754 format. It turns out that there may be four bad rounding positions to the right of the least significant digit of the unrounded redundant result. The first of these is handled by extending the significand adder to the right, and the rest are prevented in the process of leading nonzero digit detection via PN recoding [17]. More detail is supplied in the Appendix, where we show the difficulties of handling the bad rounding positions.

Before proceeding to our redundant-digit floating-point design in Section 4, we compare some of the possible approaches to the design of floating-point addition schemes. The coarse comparison in Table 1 (to be supplemented with more detailed simulation results in Section 8), is based on latency and active hardware redundancy, with the latter also serving as an indicator of power requirements.

In column 1 of Table 1, we list actual and potential implementations of Algorithm 1. Columns 2-5 show the number of steps whose latency is proportional to the logarithm of the given column parameter (per the bracketed latency formula appearing after each step of Algorithm 1). We assume that, where applicable, carry acceleration is employed to achieve logarithmic latency, with the needed extra hardware in terms of units of width (μ) given in column 6. Full replication (e.g., SD adders to concurrently compute sum and $sum \pm 1$ in [12, Fig. C.1]) is cited in column 7. The last row of Table 1 is included for completeness; its entries will be justified once we have explained our work. Other parts are explained below, with the understanding that Steps 1 and 3 (7 and 8) in Algorithm 1 may be overlapped, with no cost penalty.

- **Single-path implementation**: In this implementation, Steps 2, 4, 5, 6, and 8 operate on the full significand (4 and 5 possibly in parallel). Hardware components used for Steps 1, 2, 4, 5, 6, 7, and 8, require seven carry acceleration mechanisms. Use of compound adders for rounding may require full hardware replication.
- **Dual-path implementation**: Here, owing to the inclusion of Steps 2 and 6 in separate paths, fewer stages in pipelined implementation and a lower latency than that of single-path implementation are both achieved, but at the cost of greater hardware redundancy. Each path needs at least one significand adder, with more copies of the adder required in some designs, depending on how signed-magnitude addition is implemented.

TABLE 2
Naming Conventions for Twits

Logical variable/constant	Description	Arithmetic worth
x	Posibit ●	$\|x\| = x$
x^-	Negabit ○	$\|x^-\| = x - 1$
$x^\pm$	Unibit ▣	$\|x^\pm\| = 2x - 1$
x' x''	Equally weighted variables	N/A
0, 1	Superscript does not change the logical value	0, 1
$0^-, 1^-$		$-1, 0$
$0^\pm, 1^\pm$		$-1, 1$

- **MRSD floating-point adder**: In Fahmy's design [12], Step 4 entails digit-length (i.e., h-bit) carry propagation. But either Step 2 or Steps 5-6 require $\lceil \text{width}(\mu)/h \rceil$-digit operations in the worst case. The delay for exponent difference computation (Step 1) is hidden by postponed rounding. The rather high hardware overhead is due to two shifters and five SD adders used (four in the normalization path and one in the alignment path), with each adder internally triplicated to also compute $sum \pm 1$. Each of these 15 adders, the two shifters, and the leading zeros predictor (18 units in all) is assumed to contain carry acceleration circuitry.

3 REDUNDANT REPRESENTATIONS

To allow the use of conventional arithmetic components for bit compression and manipulation, redundant digits are often encoded as weighted bit sets. We have previously studied weighted-bit-set (WBS) encodings for digit sets and extended them to include the use of other two-valued digits [11]. Graphically, we use an extended dot notation: ● for a bit or *posibit* in $\{0, 1\}$, ○ for a *negabit* in $\{-1, 0\}$, ▣ for a *unibit* in $\{-1, 1\}$. Symbolically, we distinguish the latter two with letters or constants (0, 1) bearing the superscripts $^-$ and $^\pm$, respectively (Table 2). Generalized two-valued digits (twits) lead to weighted twit-set (WTS) encoding of digit sets. Logically, we use 0 for the smaller and 1 for the larger of the two twit values, a convention leading to inverted encoding of negabits (0 denotes -1 and 1 denotes 0), complementary to the common usage. Such encodings (Table 2) have been shown to result in efficient, VLSI-friendly adder designs [11], [18].

As noted in Section 2, the rounding-packet forwarding scheme of Nielsen et al. [7] requires double-size registers and the MRSD adder uses three parallel 5-bit 2's-complement adders. The area and power dissipation penalties of this triplication of active hardware are significant. The maximally redundant radix-16 digit set $[-15, 15]$ also complicates the leading nonzero digit detection. Both of the latter undesirable characteristics may be mitigated through improved encoding and/or adder design. For example, clever designs [19], [20] provide MRSD adders with no hardware redundancy, but the problem of cancellation of leading digits persists.

As another example, the stored-posibit-transfer (SPT) encoding [18] of the radix-16 digit set $[-8, 8]$, where there can
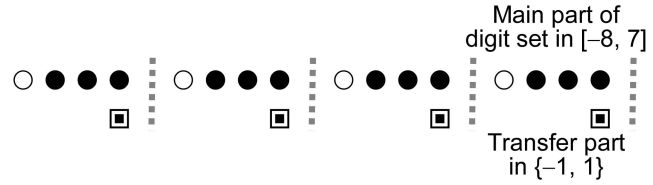

Fig. 2. SUT representation in radix 16.

be at most one leading insignificant 1 (-1) followed to the right by -8 (8), obviates the need for any complex provision to convert leading insignificant nonzero digits. But the active hardware replication factor of the SPT adder is only modestly less than that in the design of Fahmy and Flynn. Nevertheless, the SPT adder has an edge in terms of latency.

The minimally asymmetric digit set $[-9, 8]$ of this paper, which uses the SUT encoding [11], also has at most one insignificant leading 1 or -1 digit. The corresponding adder uses two rows of full-adders, which is equivalent to hardware duplication, and exhibits the same latency as the SPT adder just cited. More importantly, however, it obviates the need for rounding increment due to its ability to store the rounding value as a unibit. The cardinality of SPT or SUT digit sets is slightly more than half that of the maximally redundant radix-16 digit set. But this is not a disadvantage, because all three schemes require the same number of redundant radix-16 digits after conversion from nonredundant format (e.g., 7 for IEEE 754-2008 short format).

As shown in Fig. 2, each radix-16 SUT digit has a main 2's-complement part in $[-8, 7]$ and a transfer part in $\{-1, 1\}$. A high-level design for the ith block of a radix-16 SUT adder/subtractor, mainly composed of full-adder blocks, is shown in Fig. 3. Each input's type is symbolically denoted using our extended dot notation. Note that due to our twit encoding scheme, standard full-adders can receive and produce a variety of twit combinations, as justified in [11].

The critical path in Fig. 3 (heavy line) has a latency of five cascaded full-adders. Note, however, that the bottom full-adder row in Fig. 3 may be augmented by carry-lookahead logic for greater speed. The collective arithmetic value of the two external inputs of the lower full-adder in position 0 of the
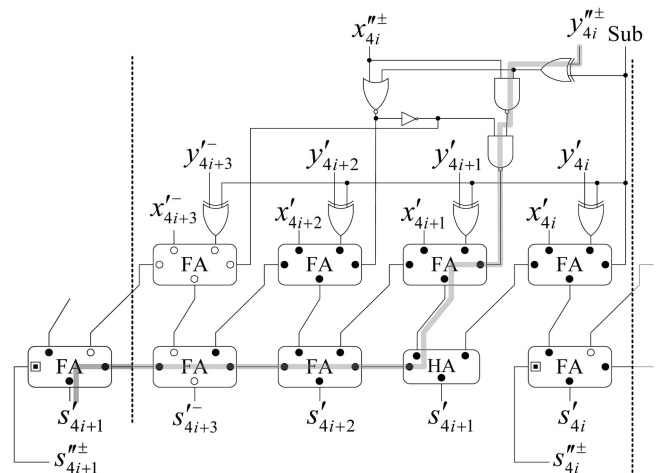

Fig. 3. The ith block of SUT adder/subtractor; the leftmost FA, rendered in gray, belongs to the next higher block. Vertical dotted lines represent radix-16 digit boundaries.

TABLE 3
Truth Table for Significand Transformation

x_{4j}	x_{4j-1}	Value	x'_{4j}	$x''^{\pm}_{4j}$	x'^{-}_{4j-1}
0	0	$0 = 2 - 2 - 0$	1	$0^{\pm}$	1^{-}
0	1	$1 = 0 + 2 - 1$	0	$1^{\pm}$	0^{-}
1	0	$2 = 0 + 2 - 0$	0	$1^{\pm}$	1^{-}
1	1	$3 = 2 + 2 - 1$	1	$1^{\pm}$	0^{-}

least-significant digit (i.e., $i = 0$) is normally zero. Therefore, these inputs can accommodate a rounding digit in $[-1, 1]$ used in lieu of rounding decrement or increment. Conversion of a 2's-complement number to an SUT number is possible in constant time (Table 3); details are given in Section 4.

4 A REDUNDANT RADIX-16 REPRESENTATION

For brevity, and without loss of generality, let the nonredundant inputs be in IEEE short format (α in Fig. 4). Note, however, that the experimental results in Section 8 are based on the long format, as are those of [12]. Our redundant binary and radix-16 representations correspond to formats β and γ in Fig. 4. Recall that we distinguish negabits and unibits with letters or constants (0, 1) bearing the superscripts $^{-}$ and $^{\pm}$ (Table 2). Also, primed and double-primed symbols in the same position denote equally weighted entities. The variables s, e_i, and γ_i ($0 \leq i \leq 7$) represent the sign, exponent twits, and radix-16-digit components of the significand.

The process of converting from IEEE 754-2008 to SUT format is described next. The explanations may be long, but the process itself is quite simple in hardware cost and latency. Conversion from other nonredundant formats is similar.

Exponent conversion. The biased input exponent $e = e_7e_6e_5e_4e_3e_2e_1e_0$ is converted to the unbiased exponent $e' = e'_7e'^{-}_6 e'^{-}_5 e'^{-}_4 e'^{-}_3 e'^{-}_2 e'^{-}_1 e'^{-}_0$ by simple copying: $e'_i = e_i$. Using $\|b\|$ to denote the arithmetic value of a bit-string b, the fact that the same exponent bit-string interpreted differently becomes the unbiased internal exponent is justified by:

$$\|e'\| = \|e'_7 e'^{-}_6 \ldots e^{-}_0\| = \|e_7(e_6 - 1) \ldots (e_0 - 1)\|$$
$$= \|e_7 e_7 \ldots e_0\| - 127 = \|e\| - 127.$$

Due to inverted encoding of negabits, the lowest (highest) possible value for e', that is, -127 (128), is represented by a string of eight 0s (1s). Therefore, ease of comparison (i.e., the rationale for conventional use of a biased exponent) is achieved here, with the unbiased exponent e'.

Significand conversion. Each bit-pair $x_{4j}x_{4j-1}(0 < j < 6)$ is independently transformed, leading also to the introduction of a transfer in position $4j$ (see Table 3). The following equations govern all transformations:

$$x'_0 = \overline{x_0}; \qquad x''^{\pm}_0 = x_0; \quad x'^{-}_{4j-1} = \overline{x_{4j-1}} \quad (1 \leq j \leq 5),$$
$$x'_{4j} = x_{4j} \odot x_{4j-1} \quad x''^{\pm}_{4j} = x_{4j} \vee x_{4j-1} \quad (1 \leq j \leq 5),$$
$$x'_{4j+1} = x_{4j+1}; \qquad x'_{4j+2} = x_{4j+2} \qquad (1 \leq j \leq 5).$$

Radix conversion for the significand. Starting at the right end of the format β in Fig. 4, every four positions, up to but not including the leftmost part of the significand, may be viewed as a redundant SUT digit γ with a 4-bit 2's-complement main part and a stored unibit in its least-significant position. The proper handling of the leftmost 4-bit group in the significand will be discussed along with the conversion of the exponent base to 16.

Radix-16 exponent. The radix-16 exponent is actually $e'_7e'^{-}_6 e'^{-}_5 e'^{-}_4 e'^{-}_3 e'^{-}_2$, with the effect of $e'^{-}_1 e'^{-}_0$ taken into account by appropriate binary shift of the significand so as to preserve the value of the floating-point number. The process is shown in Table 4 for positive significands, where the γ variables are radix-16 SUT digits. The radix point is between γ_6 and γ_5, with $\gamma_6 \geq 1$; hence, we have a normalized radix-16 floating-point representation. To have a full radix-16 SUT digit before the radix point, we extend the significand width to 28 bits, with the following shift decisions in effect (dashed lines near the lower-right corner of Fig. 4 show the four possible alignments that might arise):

- $\|e'^{-}_1 e'^{-}_0\| = 0$: No shifting is needed, but the 23-bit significand is extended to the right by one binary position, which is filled with the effective arithmetic value 0. The appearance of x_{22} to the left of the radix point is due to the SUT transformation. The hidden 1 is accommodated as a unibit and the effective arithmetic value of the three most-significant bits is 0.
- $\|e'^{-}_1 e'^{-}_0\| = -1$: The fractional part of the original significand is right-shifted by one binary position to
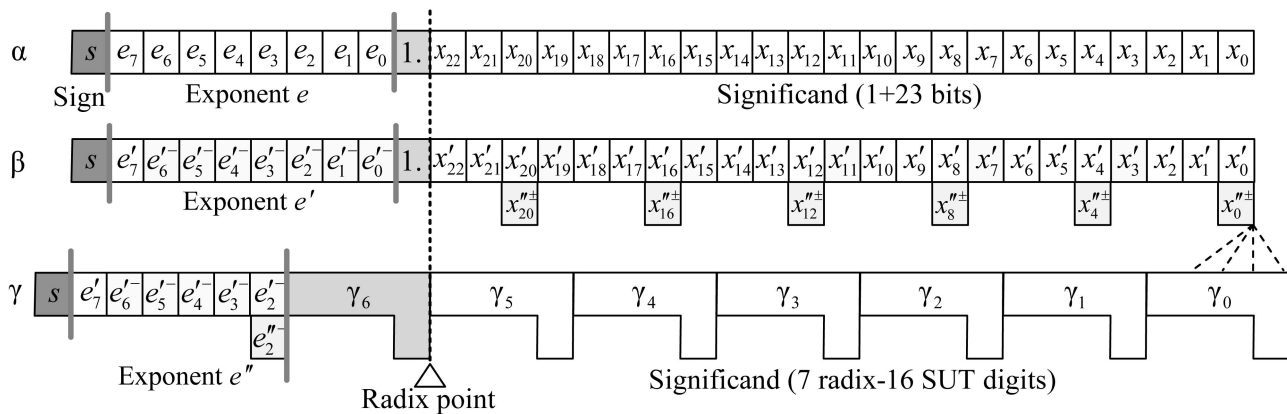

Fig. 4. Short (single) floating-point format α, with the hidden 1 exposed; an equivalent representation β, with redundant significand and unbiased exponent; and the internal radix-16 redundant format γ used in our design.

TABLE 4
Transformed Significand for Different Values of $e'^-_1 e'^-_0 =$ ($\odot$ Stands for Exclusive NOR Operator)

$e'^-_1 e'^-_0$	$\|e'^-_1 e'^-_0\|$	e''^-_2	γ_6	γ_5	$\cdots$	γ_1	γ_0
1^-1^-	0	1^-	1^- 0 0 x_{22} $1^\pm$	$\overline{x_{22}}$ x_{21} x_{20} $x_{19}\odot x_{18}$ $(x_{19} \vee x_{18})^\pm$	$\cdots$	$\overline{x_6}$ x_5 x_4 $x_3\odot x_2$ $(x_3 \vee x_2)^\pm$	$\overline{x_2}$ x_1 x_0 1 $0^\pm$
1^-0^-	-1	1^-	1^- 0 0 0 $1^\pm$	0^- x_{22} x_{21} $x_{20}\odot x_{19}$ $(x_{20} \vee x_{19})^\pm$	$\cdots$	$\overline{x_7}$ x_6 x_5 $x_4\odot x_3$ $(x_4 \vee x_3)^\pm$	$\overline{x_3}$ x_2 x_1 $\overline{x_0}$ $x_0^\pm$
0^-1^-	-2	0^-	1^- 1 x_{22} $x_{21}\odot x_{20}$ $(x_{21} \vee x_{20})^\pm$	$\overline{x_{20}}$ x_{19} x_{18} $x_{17}\odot x_{16}$ $(x_{17} \vee x_{16})^\pm$	$\cdots$	$\overline{x_4}$ x_3 x_2 $x_1\odot x_0$ $(x_1 \vee x_0)^\pm$	$\overline{x_0}$ 0 0 1 $0^\pm$
0^-0^-	-3	0^-	1^- 1 1 $x_{22}\odot x_{21}$ $(x_{22} \vee x_{21})^\pm$	$\overline{x_{21}}$ x_{20} x_{19} $x_{18}\odot x_{17}$ $(x_{18} \vee x_{17})^\pm$	$\cdots$	$\overline{x_5}$ x_4 x_3 $x_2\odot x_1$ $(x_2 \vee x_1)^\pm$	$\overline{x_1}$ x_0 0 1 $0^\pm$

compensate for ignoring the value of $\|e'^-_1 e'^-_0\|$. With this right shift, the hidden 1 moves to the right of the binary point, but to conform with SUT format, we transform the hidden 1 to a negabit 0^- (with arithmetic value -1) in the same position and a unibit $1^\pm$ (with effective arithmetic value 2) in the next higher position. The arithmetic value of the other four leftmost bits is zero.

- $\|e'^-_1 e'^-_0\| = -2$: A right shift by two binary positions to compensate for ignoring the value of $\|e'^-_1 e'^-_0\|$ may denormalize the radix-16 representation. We therefore use a 2-position left shift instead, and compensate for it by storing a 0^- in place of e''^-_2 under e'^-_2. The rationale for storing an adjustment, instead of decrementing the exponent, is to minimize the conversion latency.
- $\|e'^-_1 e'^-_0\| = -3$: An argument similar to the one just offered substantiates a binary left shift by one position.

Sign embedding. When the original floating-point number is negative, we embed the sign into the internal radix-16 representation by negating each of the seven radix-16 SUT digits of the transformed significand (Table 4) in parallel. Likewise, when the required operation is subtraction, the subtractor is negated, with the result added to the subtrahend. Therefore, the actual operation is always addition, leading to the elimination of the following provisions and corresponding reductions in the hardware complexity and latency:

- detection of the actual operation,
- possible swapping of the operands,
- postcomplementation, and
- widening of the addition circuitry to capture extra digits for rounding.

One drawback of SUT paradigm, although not problematic for subtraction (see Fig. 3), is that negation of an SUT digit involves digitwide carry propagation in general, given the asymmetry of the digit set [21]. Here, however, the value interdependence between the posibit and unibit in position 0 of each SUT digit (see Table 4), produced by the conversion outlined above, obviates the need for carry propagation in digit-by-digit negation. An SUT digit is negated by inversion of its unibit transfer and 2's-complementation of its main part.

The latter requires carry propagation in general, but for the digits γ_1 through γ_5, as well as for γ_6 in the bottom two rows of Table 4, position 0 of each digit γ holds a posibit $p =_{x_{j+1}} \odot x_j$ and a unibit $u^\pm = (x_{j+1} \vee x_j)^\pm$. Complementing all twits and adding a constant posibit 1 in position 0 of the latter digits leaves the three twits $1, \overline{p}$, and $\overline{u^\pm}$ in position 0. The first two of these may be replaced by the original $p = 1 \oplus \overline{p}$ and a carry $\overline{p}$ into the next position. Collectively, the value of this carry (i.e., $2(x_{j+1} \oplus x_j)$) and the unibit $\overline{u^\pm}$ (i.e., $-1 + 2\overline{u^\pm}$) is $2(x_{j+1} \oplus x_j) - 1 + 2\overline{x_{j+1}\vee x_j}$, which can be accommodated by a unibit $(\overline{x_{j+1}x_j})^\pm$, as justified in Table 5, where only the second column from the right holds arithmetic constants. The value-specific SUT digits γ_0 as well as instances of γ_6 in the upper two rows of Table 4 are easily negated independently.

Based on the observations above, the two-step process for accommodating the sign of a negative nonredundant floating-point number in its equivalent radix-16 SUT encoding can be reduced to a direct process (Table 6). Moreover, examination of Tables 4 and 5 shows that the overall conversion process can be summarized as follows for hardware implementation:

- Extend the nonredundant floating-point number by one radix-2 position to the right and perform no operation, 1-bit right shift, 2-bit left shift, or 1-bit left shift, for $\|e'^-_1 e'^-_0\| = 0$, -1, -2, or -3, respectively. In other words, we can use e'^-_1 to control right ($e'^-_1 = 1$) or left ($e'^-_1 = 0$) and e'^-_0 for choosing odd ($e'^-_0 = 0$, 1-bit shift) or even ($e'^-_0 = 1$, 0-bit right or 2-bit left) shift amount.
- Exclusive-or the sign bit with all bits of the significand.
- Restructure into SUT format.

TABLE 5
Deriving the Unibit Transfer After Negation

Weight:		2	1	1	1
x_{j+1}	x_j	$x_{j+1}\oplus x_j$	$(\overline{x_{j+1} \vee x_j})^\pm$	$2(x_{j+1}\oplus x_j) - 1 + 2\overline{x_{j+1}} \vee x_j$	$(\overline{x_{j+1}x_j})^\pm$
0	0	0	$1^\pm$	1	$1^\pm$
0	1	1	$0^\pm$	1	$1^\pm$
1	0	1	$0^\pm$	1	$1^\pm$
1	1	0	$0^\pm$	-1	$0^\pm$

TABLE 6
Truth Table for Significand Transformation

$e_1'^- e_0'^-$	$\|e_1'^- e_0'^-\|$	$e_2''^-$	γ_6	γ_5	$\cdots$	γ_1	γ_0
1^-1^-	0	1^-	0^- 1 1 $\dfrac{x_{22}}{\overline{x_{22}}^{\pm}}$	$\overline{x_{22}}$ $\overline{x_{21}}$ $\overline{x_{20}}$ $\dfrac{x_{19}\odot x_{18}}{(\overline{x_{19}x_{18}})^{\pm}}$	$\cdots$	$\overline{x_6}$ $\overline{x_5}$ $\overline{x_4}$ $\dfrac{x_3\odot x_2}{(\overline{x_3 x_2})^{\pm}}$	$\overline{x_2}$ $\overline{x_1}$ $\overline{x_0}$ $\dfrac{1}{1^{\pm}}$
1^-0^-	-1	1^-	0^- 1 1 $\dfrac{0}{1^{\pm}}$	1^- $\overline{x_{22}}$ $\overline{x_{21}}$ $\dfrac{x_{20}\odot x_{19}}{(\overline{x_{20}x_{19}})^{\pm}}$	$\cdots$	$\overline{x_7}$ $\overline{x_6}$ $\overline{x_5}$ $\dfrac{x_4\odot x_3}{(\overline{x_4 x_3})^{\pm}}$	$\overline{x_3}$ $\overline{x_2}$ $\overline{x_1}$ $\dfrac{\overline{x_0}}{1^{\pm}}$
0^-1^-	-2	0^-	0^- 0 $\overline{x_{22}}$ $\dfrac{x_{21}\odot x_{20}}{(\overline{x_{21}x_{20}})^{\pm}}$	$\overline{x_{20}}$ $\overline{x_{19}}$ $\overline{x_{18}}$ $\dfrac{x_{17}\odot x_{16}}{(\overline{x_{17}x_{16}})^{\pm}}$	$\cdots$	$\overline{x_4}$ $\overline{x_3}$ $\overline{x_2}$ $\dfrac{x_1\odot x_0}{(\overline{x_1 x_0})^{\pm}}$	$\overline{x_0}$ 1 1 $\dfrac{1}{1^{\pm}}$
0^-0^-	-3	0^-	0^- 0 0 $\dfrac{x_{22}\odot x_{21}}{(\overline{x_{22}x_{21}})^{\pm}}$	$\overline{x_{21}}$ $\overline{x_{20}}$ $\overline{x_{19}}$ $\dfrac{x_{18}\odot x_{17}}{(\overline{x_{18}x_{17}})^{\pm}}$	$\cdots$	$\overline{x_5}$ $\overline{x_4}$ $\overline{x_3}$ $\dfrac{x_2\odot x_1}{(\overline{x_2 x_1})^{\pm}}$	$\overline{x_1}$ $\overline{x_0}$ 1 $\dfrac{1}{1^{\pm}}$

The latency for conversion from nonredundant floating-point format to SUT format equals the delay of 2 bit-shifts, an XOR gate for sign embedding, and an OR gate for the final restructuring into SUT format.

5 PATH SELECTION

With the embedded sign representation of Section 4, addition and subtraction operations do not need to be distinguished in implementation. Therefore, we focus only on the radix-16 exponent difference $\Delta'' = \varepsilon'' - \eta''$ used for path separation, where $\varepsilon'' = \varepsilon_7'\varepsilon_6'^- \varepsilon_5'^- \varepsilon_4'^- \varepsilon_3'^- [\varepsilon_2'^- \varepsilon_2''^-]$ and $\eta'' = \eta_7'\eta_6'^- \eta_5'^- \eta_4'^- \eta_3'^- [\eta_2'^- \eta_2''^-]$ represent the exponents of the two operands f_1 and f_2 according to the encoding γ of Fig. 4. The binary positions enclosed in square brackets indicate equally weighted negabits in the least-significant positions of the two exponents. As is common in high-performance floating-point units, we develop a dual-path adder, with the *alignment path* allowing word-length alignment shifts and the *normalization path* going through a postaddition normalizing shifter. Both paths are active in every add/subtract operation, but one of the results may be wrong. The correct result is obtained through multiplexing the result from the alignment and normalization paths. The selection is based on the exponent difference Δ'', as detailed in the following paragraphs.

Fig. 5 depicts our dual-path adder. The two paths have shared exponent difference and rounding logic. To avoid hardware redundancy owing to concurrent shifted and nonshifted additions, no prediction logic for exponent difference is used in the normalization path. We can now explain the entries of the last row of Table 1. The rounding block latency (to be discussed in Section 7) amounts to only three logic levels, and as such cannot hide the latency of exponent difference computation. However, the exponent difference computation for the next floating-point addition may begin at the same time as the process of rounding decision, thereby taking the latter off the critical path. Leading zero detection and the following normalization shifts contribute to the column titled $\lceil \text{width}(\mu)/h \rceil$, while the former adder latency is proportional to $\log h$. The two adders, two shifters, exponent subtractor, and detection logic (total of six) use carry acceleration circuitry, and there are three instances of hardware replication arising from adders, each with internal duplication due to two full-adder rows in Fig. 3.

Alignment path. For $\Delta'' \geq 2$, the latency of alignment shift is significant, while that of normalization shift is minimal. This is because for the operand having the larger exponent (say f_1), we have $|\gamma_6| \geq 1$ and $-9 \leq \gamma_5 \leq 8$, while for the operand with the smaller exponent (f_2), the postalignment values are $\gamma_6 = \gamma_5 = 0$. When $|\gamma_6| \geq 2$ for f_1, MSD of the result is nonzero and there is no normalization shift. When $|\gamma_6| = 1$ for f_1, the transfer value to the most-significant digit of the result may make it zero. In this case, however, the next most-significant digit cannot be zero, given the γ_5 values for the two operands. Thus, we will either have a normalized result or need a single-digit right/left shift to normalize it. The sign and magnitude of the exponent difference are derived by Algorithm 2 below. Note that inverted encoding of negabits is in effect.

Normalization path. For $\Delta'' = 0$, no exponent comparison, swapping, or postcomplementation is needed. This is due to sign-embedded representation of significands that allows the result to be negative. In case of $\Delta'' = 1$, there is only a 1-digit alignment right shift, but in both cases (i.e., $\Delta'' \leq 1$), a lengthy normalization shift may be necessary. To compute the amount of normalization postshift, we need to locate the first nonzero digit. Because the SUT-encoded
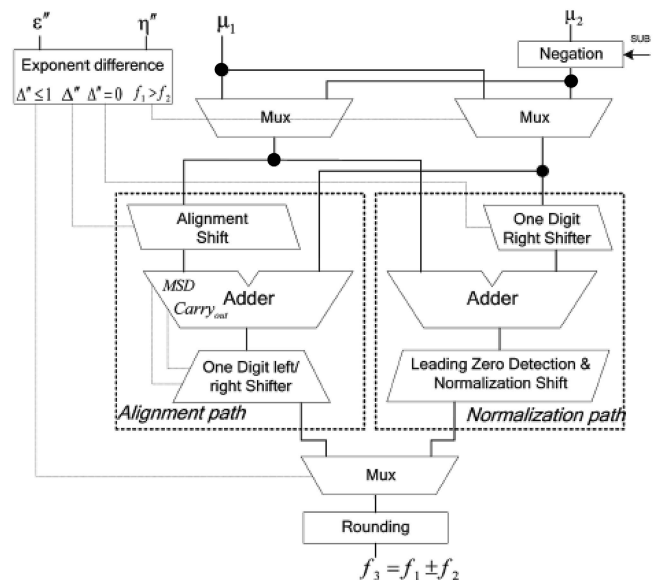

Fig. 5. Dual-path SUT floating-point adder.

result does not allow leading insignificant digits, except for a single 1 or -1, conventional leading nonzero digit detector may be used and no extra provisions, such as PN recoding circuitry [22], as used for the maximally redundant signed-digit encoding of Fahmy and Flynn [10], is required.

Algorithm 2. Computing the exponent difference
$\Delta'' = \varepsilon'' - \eta''$

1. *Negating the 2nd exponent*: Invert all bits of η''. Since the encoding has a symmetric range $[-32, 32]$, simple parallel inversion is sufficient (see Theorem 1 in [21]).
2. *Deriving the exponent difference*: Use a 6-bit ripple-carry adder (or an equivalent fast adder) to derive
$\Delta'' = d_6' d_5'^- d_4'^- d_3'^- d_2'^- d_1'^- [d_0'^- d_0''^-]$, where all full-adders, except the one in the leftmost position, receive three negabits. The fourth negabit in the rightmost position is kept intact, so that $d_0''^- = \eta_2''^-$. Therefore,
$\|\Delta''\| = 2^6 d_6' + \sum_{i=1}^{5} 2^i (d_i'^- - 1) + d_0'^- - 1 + d_0''^- - 1$.
The latter can be reduced to $\|\Delta''\| = 2^6(d_6' - 1) + \delta$, where $\delta = \sum_{i=1}^{5} 2_i d_i'^- + d_0'^- + d_0''^- \le 2^6$.
3. *Sign of the difference*: Complement the most-significant posibit to obtain the sign. The reason is that $d_6' = 0$ (1) leads to $\|\Delta''\| \le 0$ ($\|\Delta''\| \ge 0$).
4. *Magnitude of the difference*: The absolute value of $\|\Delta''\|$ is $2^6 - \delta$, for $d_6' = 0$, and δ, for $d_6' = 1$. The latter is represented by $d_5'^- d_4'^- d_3'^- d_2'^- d_1'^- [d_0'^- d_0''^-]$ and the former by $\overline{d_5'^-} \ \overline{d_4'^-} \ \overline{d_3'^-} \ \overline{d_2'^-} \ \overline{d_1'^-} \ [\overline{d_0'^-} \ \overline{d_0''^-}]$. Therefore, the negabits of Δ'' or their complements, which represent the magnitude of the exponent difference, can be used to control a parallel (e.g., barrel) shifter directly.

When $\|\Delta''\| > 6$ ($\|\Delta''\| < -6$), all the digits of the operand with the smaller exponent will be shifted out. Therefore, only $d_2'^- d_1'^- [d_0'^- d_0''^-]$ ($\overline{d_2'^- d_1'^-} \ [\overline{d_0'^- d_0''^-}]$) are needed to control the parallel shifter. Shifting is not required when $d_5'^- d_4'^- d_3'^- \ne 000$ ($\overline{d_5'^-} \ \overline{d_4'^-} \ \overline{d_3'^-} \ne 000$); in this case, no addition is necessary, but the shifted-out digits may contribute to the rounding decision.

6 GUARD, ROUND, AND STICKY DIGITS

In the alignment path, two or more radix-16 digits (i.e., SUT digits in $[-9, 8]$) of the operand with the smaller exponent are shifted out. As discussed in Section 5, normalization postshift in this path is limited to one digit. Therefore, one needs only to save as a guard digit the most significant one of the digits that were shifted out. This guard digit may then be shifted back during the postnormalization process. The other shifted-out digits are only needed to the extent that they affect the rounding decision.

For conventional floating-point units, an extra round bit and a sticky bit (logical OR of all subsequent bits) suffice for correct rounding, where the sticky bit being 0 signals the need for applying the halfway rule of the round-to-nearest-even mode. For redundant-digit floating-point units, however, round and sticky information should carry information about the range of the shifted-out digits, at least indicating whether the fractional value represented by the shifted-out digits is negative, zero, or positive.

In our radix-16 SUT floating-point adder, we recognize three (four) possibilities for sticky (guard and round) digit
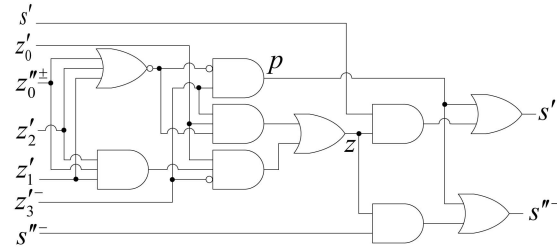

Fig. 6. The logic for deriving a new sticky digit from the old one and the shifted-out digit.

values. We use the equally weighted posibit s' and negabit s''^- to represent sticky information such that $s's''^- = 00^-$, 01^-, and 11^- (10^- not used) represent a negative, zero, and positive value passed into and through the sticky position, respectively. Rounding information, on the other hand, is encoded as $r_1^- r_0 0^- 0$, $0^- 1$, $1^- 0$, and $1^- 1$ representing the following values/ranges -9, -8, $[-7, 7]$, and 8, respectively. A zero-valued shifted-out digit does not change the sticky digit, but a positive (negative) one makes the sticky digit also positive (negative).

Fig. 6 depicts the logic required for deriving the new sticky digit from the old one, where z and p indicate that the shifted-out digit, represented by input twits $z_3'^- z_2' z_1' \ (z_0' z_0''^\pm)$, is zero or positive, respectively. The output bits s' and s''^- at the right edge of Fig. 6 are to be latched back to inputs with the same names appearing at the left, with the initial value $s's''^- = 01^-$ (i.e., zero) and inverted encoding of negabits in effect. Note that in case of no normalization shift, the guard digit will serve as the round digit and the original round digit should be fed in to the logic of Fig. 6 for updating the sticky digit. As in nonredundant floating-point addition, in the alignment path of SUT floating-point addition, a one-digit normalization right shift may also occur, in which case the postaddition shifted-out digit serves as the round digit. The original guard and round digits should then modify the sticky digit through the logic of Fig. 6.

In the normalization path, where alignment shift is limited to one right shift, maintaining the guard digit is still required. With one alignment shift and no postshift (a one-digit postshift to the right), the guard (shifted-out) digit serves the same purpose as the round digit and sticky digit is zero (is derived from the guard digit). But when there are one or more shifts to the left, no rounding action is necessary, given that the round and sticky digits are zero.

7 ROUNDING DECISION

In this section, we describe how the round-to-nearest-even mode might be implemented with our SUT floating-point addition scheme. The main idea is to compute a rounding value R_v to be stored as the transfer part of the least-significant digit in the final sum. Note that the SUT addition scheme (described in Section 3) does not generate a value for the transfer part of the least-significant digit, thus leaving it available for our rounding scheme. Unfortunately, however, this simple rounding decision does not always work correctly. The challenges include the possible shifting of the transferless LSD and bad rounding positions. We discuss the former below, and deal with the latter in the
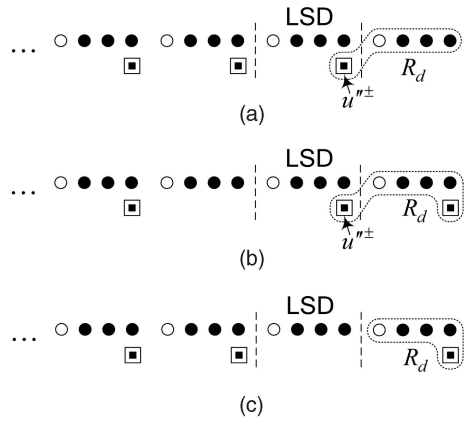
Fig. 7. The rounding digit illustrated.

TABLE 7
Derivation of Rounding Value and Adjustment of LSB

Case No.	$u''\pm$	$r_1^- r_0$	$s' s''^-$	l_b	R_v	Adjusted $r''\pm$	l_a
1	$0^\pm$	0^-0	X X	1	-2	$0^\pm$	0
2	$0^\pm$	0^-1	00^-	1	-2	$0^\pm$	0
3	$0^\pm$	0^-1	01^-	1	-1	$0^\pm$	1
4	$0^\pm$	0^-1	11^-	1	-1	$0^\pm$	1
5	$0^\pm$	1^-0	X X	X	-1	$0^\pm$	l_b
6	$0^\pm$	1^-1	00^-	X	-1	$0^\pm$	l_b
7	$0^\pm$	1^-1	01^-	0	0	$0^\pm$	1
8	$0^\pm$	1^-1	01^-	1	-1	$0^\pm$	1
9	$0^\pm$	1^-1	11^-	0	0	$0^\pm$	1
10	$0^\pm$	1^-1	11^-	1	0	$1^\pm$	0
11	$1^\pm$	0^-0	X X	0	0	$0^\pm$	1
12	$1^\pm$	0^-0	X X	1	0	$1^\pm$	0
13	$1^\pm$	0^-1	00^-	0	0	$0^\pm$	1
14	$1^\pm$	0^-1	00^-	1	0	$1^\pm$	0
15	$1^\pm$	0^-1	01^-	0	0	$0^\pm$	1
16	$1^\pm$	0^-1	01^-	1	1	$1^\pm$	1
17	$1^\pm$	0^-1	11^-	X	1	$1^\pm$	l_b
18	$1^\pm$	10	X X	X	1	$1^\pm$	l_b
19	$1^\pm$	1^-1	00^-	0	1	$1^\pm$	0
20	$1^\pm$	1^-1	01^-	0	2	$1^\pm$	1
21	$1^\pm$	1^-1	11^-	0	2	$1^\pm$	1

Appendix, where we identify two bad rounding positions, to the right of the LSD, that may lead to a redundant value different from that obtained by nonredundant floating-point operations in accordance with IEEE 754-2008 standard. For the sake of a comprehensive comparison, we also show that a simple rounding decision scheme in case of [7] fails in four bad positions.

Unfortunately, there seems to be no simple solution for correct rounding in the case of bad rounding positions such that the requirements of IEEE 754-2008 are met. This is true for both our SUT representation and for Fahmy and Flynn's scheme. The reason is that the combined contribution of the rounding value and the other bits at position R_p or higher is less than 1 ulp, and thus, cannot be stored with the LSD. However, no problem arises for rounding after conversion to nonredundant format. To solve the problem, Fahmy and Flynn introduce additional complexity to overcome the problem of bad positions, as noted in the Appendix. Alternate redundant number systems are under investigation by the authors to alleviate this problem. Meanwhile, the SUT floating-point scheme, primarily owing to the advantages arising from "sign embedding" (refer to Section 4), may find applications in special-purpose processors and embedded systems, where full compliance with IEEE 754-2008 is not a requirement.

7.1 Shifting of the Transferless LSD

Due to normalization shifts, the transferless LSD may be shifted right or left, and replaced by the next more significant digit or by the guard digit, respectively. Both replacements lead to a new LSD with a transfer digit, and thus, no room for storing the rounding value. We consider the cases of right, left, and no shift separately (Figs. 7a, 7b, and 7c, respectively), and summarize the results in Table 7.

7.1.1 Normalization Right Shift

The LSD of the right-shifted result (i.e., the new LSD) has a nonempty unibit transfer $u''\pm$. Let R_d (for rounding digit) denote the collective value of $u''\pm$ and the last shifted-out digit (i.e., the old transferless LSD). The twits that constitute R_d have actually been generated as the sum of two radix-16 SUT digits in LSD positions of the two operands and then

shifted one radix-16 position to the right. The following interval equation summarizes the process:

$$[-9\ ulp,\ 8\ ulp]\ +\ [-9\ ulp,\ 8\ ulp]\ [-18\ ulp,\ 16\ ulp]$$
$$\overset{shift}{\Longrightarrow}\ R_d\ \in\ [-(18/16)\ ulp,\ ulp].$$

We can now extract R_v in $\{-ulp, 0, ulp\}$ for different subranges of R_d as follows:

$$-\frac{18}{16}ulp \le R_d < -\frac{ulp}{2} \Longrightarrow R_v = -1,$$
$$-\frac{ulp}{2} < R_d < \frac{ulp}{2} \Longrightarrow R_v = 0,$$
$$\frac{ulp}{2} < R_d \le ulp \Longrightarrow R_v = 1.$$

In case of the two singular boundary cases (i.e., $-ulp/2$ and $ulp/2$), R_v again in $\{-ulp, 0, ulp\}$ is decided by the sticky digit, and parity of the new LSD.

7.1.2 Normalization Left Shift

In this case, a nonzero R_v occurs only in the alignment path, where a shifted-out digit may be placed back in the LSD position. The transfer part of this digit and the main part of the next digit to the right, if any (i.e., R_d as defined in Section 7.1.1), have been generated either as the sum of two radix-16 SUT digits from the previous addition, or through conversion of a nonredundant operand to SUT format. The former may be treated exactly as Section 7.1.1. For the latter,
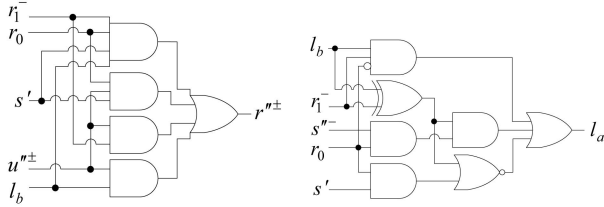
Fig. 8. Rounding logic.

observe that by significand conversion method, explained in Section 4, R_d lies in $[-(17/16)\ ulp,\ (24/16)\ ulp]$. R_v can now be extracted as follows:

$$-\frac{17}{16}ulp \leq R_d < -\frac{ulp}{2} \implies R_v = -1,$$

$$-\frac{ulp}{2} < R_d < -\frac{ulp}{2} \implies R_v = 0,$$

$$\frac{ulp}{2} < Rd < \frac{24}{16}\ ulp \implies R_v = 1.$$

For the singular cases of $\pm ulp/2$, the rounding decision is as in Section 7.1.1. Note that depending on the sticky digit, $R_d = 24/16\ ulp$ could lead to $R_v = 2$. However, given the significand conversion equations of Section 4, the latter situation occurs only for $l_b = 0$, where l_b is the least-significant posibit of LSD; hence, the possibility of storing R_v in lieu of the two least-significant twits of LSD (#20 in Table 7).

7.1.3 No Normalization Shift

The LSD is transferless in this case. For the sake of uniformity in the rounding decision, however, we assume a unibit $u''^{\pm}$ in the least-significant end of LSD. This unibit may be produced by placing a second full-adder in the low end of the SUT digit adder handling the LSD position (similar to the lower right full-adder of Fig. 3), and setting its input negabit to 1 and the right input posibit to 0, collectively representing the arithmetic value 0. Entries #1 to #4 in Table 7 occur particularly in this case when the collective arithmetic value of l_b and $u''^{\pm}$ is 0. Other entries for the no-shift case are shared by the cases of normalization shift.

Table 7 shows 21 combinations in deriving the rounding value; the other 43 possible combinations cannot occur and constitute *don't-care* conditions. Here, $u''^{\pm}$, $r_1^- r_0$, $s's''^-$, and R_v represent the unibit transfer of LSD, round, and sticky digits, as defined in Section 6, and the rounding value. The rightmost posibits of the LSD before and after storing R_v are indicated as l_b and l_a, and $r''^{\pm}$ is the stored rounding unibit. Note that the rounding value computation is done after normalization shifts have taken place, and the first digit after LSD is viewed as the round digit. The rounding value -2 (2) is taken care of by $l_a = 0$ (1) and $r''^{\pm} = 0$ (1), since they can only occur when l_b is adjusted to 1 ($u''^{\pm} = 0$) and 0 ($u''^{\pm} = 1$), respectively. Fig. 8 depicts the simple logic implementing the following equations for $r''^{\pm}$ and l_a, based on Table 7:

$$r''^{\pm} = r_1^- l_b r_0 s' \vee u''^{\pm}(r_1^- \vee l_b \vee r_0 s'),$$

$$l_a = \overline{(r_1^- \oplus l_b)} \vee r_0 s' \vee (r_1^- \oplus l_b) r_0 s''^- \vee (r_1^- l_b \overline{r_0}).$$

TABLE 8
FO4 Delays Used in Our Analytical Model

Component	Associated FO4 delay
Full-adder, 1-bit	2
Fast adder, k-bit	$5 + 2\lceil \log_{f-1}(\lceil k/f \rceil - 1)\rceil$
Multiplexer, m-to-1	1 : inputs-to-output $1 + \lceil \log_4 m \rceil$: selects-to-output
Shifter, n-way	$\lceil \log_2 n \rceil$

8 COMPARATIVE EVALUATIONS

The redundant digit floating-point addition scheme of Fahmy [12] and the one proposed in this paper are both based on radix-16 signed-digit number representation. We now show that the coarse comparison of Table 1 is supported by detailed analytical evaluation of both schemes. For a fair comparison, we follow the analytical model of [12]. In this model, the FO4 delays of a full-adder, a k-bit fast adder, an m-to-1 multiplexer, and an n-way shifter are as in Table 8, where f (fan-in) is the maximum number of inputs for a gate in the design. Based on the component delays of Table 8, the overall FO4 delays (the unit being an inverter delay with fan-out of 4) of the redundant-digit floating-point adders of [12] and [7] are represented by the following equations (adapted from [12]), where $h = 1$ in [7]. Recall that μ, e, and h refer to significand, exponent, and digit width ($h = \log_2 r$), respectively:

$$\tau_{\text{Ref.}[12]} = 16 + 2\lceil \log_{f-1}(\lceil \text{width } (e)/f \rceil - 1\rceil$$
$$+ 2\lceil \log_4(\lceil \text{width } (\mu)/h \rceil(h+1) + \lceil \log_2(\lceil \text{width}(\mu)/h \rceil$$
$$+ 2\lceil \log_{f-1}(\lceil h+1)/f \rceil - 1)\rceil + \lceil \log_4(h+1) \rceil,$$
$$\tau_{\text{Ref.}[7]} = 15 + 2\lceil \log_{f-1}(\lceil \text{width } (e)/f \rceil - 1)\rceil$$
$$+ 2\lceil \log_4(\text{width } (\mu)) \rceil + \lceil \log_2(\text{width}(\mu)) \rceil.$$

Using the same component delays for the critical path of our design (Fig. 5) yields the following equation, where the five variable terms that follow the fixed latency of 16 are due to exponent-difference, shifter, adder, and leading zero detector (two terms) units, respectively:

$$\tau_{\text{Proposed}} = 16 + 2\lceil \log_{f-1}(\lceil (\text{width}(e) - \log_2 h)/f \rceil - 1)\rceil$$
$$+ \lceil \log_2(\lceil \text{width}(\mu)/h \rceil) \rceil + 2\lceil \log_{f-1}(\lceil h/f \rceil - 1)\rceil$$
$$+ \lceil \log_2(\lceil \text{width}(\mu)/h \rceil) \rceil + \lceil \log_f(h+1) \rceil.$$

The preceding analysis yields 34 FO4 gate delays for the design of [12], versus 28 for our design (using $f = 3$, $h = 4$). The delay values for width (μ) ranging from 8 to 120 are plotted in Fig. 9.

For more realistic results, we produced VHDL code for both schemes and ran simulations and synthesis using the Synopsis Design Compiler. The target library is based on TSMC 0.13 μm standard CMOS technology. For dynamic and leakage power, we have used the Synopsis Power Compiler. The same design environment (e.g., operating conditions and wire model) and design constraints (e.g., maximum path delay and area consumption) are assumed for both floating-point adders being compared. The results, as depicted in Table 9, show that our proposed floating-point adder is both
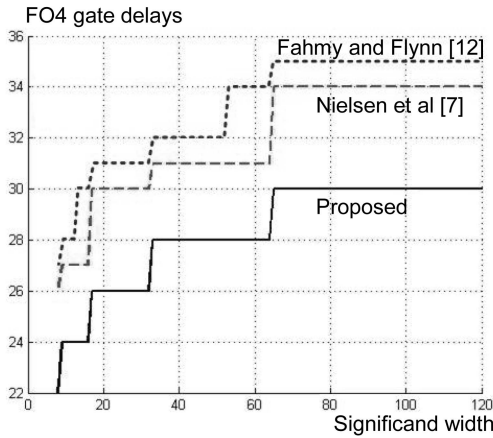
Fig. 9. Delays of three redundant-digit floating-point adders in units of FO4 gate delays.

faster and significantly outperforms the design of [12] in terms of power and area. The following differences are responsible for the improved performance, power consumption, and layout area:

1. Less hardware redundancy in the main signed-digit adder. In reference [12], three signed-digit adders are used to form sum, $sum + 1$, and $sum - 1$ simultaneously.

2. Removal of the need for postcomplementing the significand in case of equal exponents due to the sign-embedded representation. To avoid added latency, the design of [12] includes extensive hardware redundancy, in the form of multiple adders to compute $A - B$, $B - A$, $A - B_{\text{shifted}}$, and $B - A_{\text{shifted}}$. It also uses two shifters in the normalization path and a 5-to-1 final selector (versus 2-to-1 in our design).

3. Elimination of rounding increment; [12, Fig. 4.2] includes four active rounding increment/decrement modules.

4. No need for PN recoding.

9   CONCLUSION

We have described a new redundant-digit representation for floating-point numbers that leads to computation speedup as well as reduced layout area and power dissipation. These benefits have been confirmed by approximate analyses and through more detailed simulation results.

● Speedup:

-   Smaller per-operation latency for multiple floating-point additions performed before result conversion,

-   Faster rounding decision, and

-   Removal of the digit-width addition during rounding.

● Area and power:

-   Removal of swapping, postcomplementation, and out-of-word-boundary adder logic of conventional designs,

-   Drastic reduction in hardware redundancy, and

-   Simpler logic for rounding decision.

By describing algorithms and circuit implementations for a floating-point adder based on a redundant-digit representation, we have shown that the new representation offers the unique advantage of replacing the slow full-width addition required for rounding by the insertion of a rounding value, derived by a simple three-level logic circuit, in a position normally assigned to a redundant "twit" within the redundant format. We have also demonstrated that our scheme leads to a simpler rounding decision and immediate incorporation of the rounding value so that, in case of conversion to nonredundant format, no additional time beyond that of redundant-to-binary conversion is required. The cost paid for these advantages is a one-time, two-multiplexer latency for converting IEEE 754-2008 floating-point numbers to our internal format.

Although our scheme does not provide a simple solution to the problem of lack of full IEEE 754-2008 compliance with redundant representations, it does reduce the number of bad rounding positions from four, in the existing scheme of [23], to only two. Intuitively, this improvement is, in part, due to the lower redundancy of the SUT representation: digit set $[-9, 8]$, redundancy index $\rho = 2$, compared with $[-15, 15]$, $\rho = 15$. Research is now in progress on seeking suitable redundant number systems without bad rounding positions or, alternatively, proving that no such representation can exist.

APPENDIX

FLOATING ROUNDING POSITION

IEEE 754-2008 requires that in any compound operation corresponding to a sequence of basic operations, the final result be the same as what would have been obtained if intermediate results were correctly rounded after each step. For this property to hold, it must be the case that converting the unrounded result to its nonredundant equivalent and then rounding it leads to the same result as rounding the redundant result first and then converting to its nonredundant equivalent.

To deal with this challenge, we follow Fahmy's approach [12] in predicting the position of the leading 1 in the converted (i.e., nonredundant) result and accordingly

TABLE 9
Synthesis Results Based on 0.13 Micrometer Standard CMOS Technology

	Performance		Power Consumption				Complexity	
	Delay (ns)	%	Dynamic (mW)	%	Leakage (nW)	%	Area (μm^2)	%
Reference [12]	4.04	100	52.92	100	5.27	100	116 182	100
Proposed	2.99	74	22.24	42	1.79	34	48 564	42

locating the "rounding position" within the LSD of the redundant result. The rounding position within the LSD corresponds to the leading 1 in the MSD, except that conversion to nonredundant format may shift the leading 1 to the right by one position. There would be no propagating 1 during the conversion process, given that the radix-16 digits of the converted result can accommodate the SUT digits in [0, 8]. However, negative SUT digits in $[-9, -1]$ generate a propagating -1, leaving behind a radix-16 digit in [7, 15]. Therefore, if a propagating -1 reaches the MSD, it may turn the latter into 0, with the next digit to the right being in [6, 15]. In this case, the leading 1 after conversion will be 1-2 binary positions to the right of the MSD, implying that the rounding position, before conversion, may be 1-2 binary positions to the right of the LSD. With maximally redundant digits in $[-15, 15]$, as used in [23], however, a propagating -1 may leave behind a digit in [0, 15]. This leads to the rounding position falling as far as four binary positions to the right of the LSD. The most-significant one of these positions is handled in [23] by extending the adder to the right and the other three are prevented via PN recoding logic.

The latency of the latter prediction process is at best logarithmic in the number of significand digits. However, given that storing of the rounding value does not lead to any exponent adjustment, the prediction process may be taken off the critical path by overlapping it with the exponent difference computation of the next floating-point addition. Having obtained the rounding position, we recognize two cases for storing the rounding value:

- *Good rounding positions*: When the computed rounding position R_p coincides with one of the binary positions of the result's LSD, the posibits to the right of that position affect the rounding value R_v to be added in position R_p.

- *Bad rounding positions*: In the two cases when R_p falls to the right of the LSD, the most-significant bit that contributes to the rounding value weighs $ulp/4$ or $ulp/8$. The resulting rounding value that is not an integral multiple of ulp, regardless of the sign, cannot be stored with the LSD. After conversion to nonredundant format, however, a normalization shift of 1-2 bits moves the rounding position to the rightmost position of the LSD.

ACKNOWLEDGMENTS

REFERENCES

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs,* second ed. Oxford Univ. Press, 2010.

[2] Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Std 754-1985, Aug. 1985.

[3] Institute of Electrical and Electronics Engineers, *IEEE Standard for Floating-Point Arithmetic,* IEEE Std 754-2008, Aug. 2008.

[4] T.-J. Lin, H.-Y. Lin, C.-M. Chao, and C.-W. Liu, "A Compact DSP Core with Static Floating-Point Arithmetic," *J. VLSI Signal Processing,* vol. 42, no. 2, pp. 127-138, Feb. 2006.

[5] R. Oslejsek and J. Sochor, "Generic Graphics Architecture," *Proc. Conf. Theory and Practice of Computer Graphics,* pp. 105-112, 2003.

[6] P.M. Farmwald, "On the Design of High Performance Digital Arithmetic Units," PhD thesis, Stanford Univ., Aug. 1981.

[7] A.M. Nielsen, D.W. Matula, C.N. Lyu, and G. Even, "An IEEE Compliant Floating-Point Adder that Conforms with the Pipelined Packet-Forwarding Paradigm," *IEEE Trans. Computers,* vol. 49, no. 1, pp. 33-47, Jan. 2000.

[8] S.F. Oberman and M.J. Flynn, "Reducing the Mean Latency of Floating-Point Addition," *Theoretical Computer Science,* vol. 196, pp. 201-214, 1998.

[9] P.-M. Seidel and G. Even, "On the Design of Fast IEEE Floating-Point Adders," *Proc. 15th IEEE Symp. Computer Arithmetic,* pp. 184-194, 2001.

[10] H.A.H. Fahmy and M.J. Flynn, "The Case for a Redundant Format in Floating-Point Arithmetic," *Proc. 16th IEEE Symp. Computer Arithmetic,* pp. 95-102, 2003.

[11] G. Jaberipur, B. Parhami, and M. Ghodsi, "Weighted Two-Valued Digit-Set Encodings: Unifying Efficient Hardware Representation Schemes for Redundant Number Systems," *IEEE Trans. Circuits and Systems I,* vol. 52, no. 7, pp. 1348-1357, July 2005.

[12] H.A.H. Fahmy, "A Redundant Digit Floating Point System," PhD thesis, Stanford Univ., June 2003.

[13] S.G. Campbell, "Floating-Point Operations," *Planning a Computer System,* W. Buchholz, ed., pp. 92-106, McGraw-Hill, 1962.

[14] S.F. Anderson, J.G. Earle, R.E. Goldschmidt, and D.M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Research and Development,* vol. 11, no. 1, pp. 34-53, 1967.

[15] P.-M. Seidel and G. Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition," *IEEE Trans. Computers,* vol. 53, no. 2, pp. 97-113, Feb. 2004.

[16] S. Vassiliadis, D.S. Lemon, and M. Putrino, "S/370 Sign-Magnitude Floating-Point Adder," *IEEE J. Solid-State Circuits,* vol. 24, no. 4, pp. 1062-1070, Aug. 1989.

[17] D.W. Matula and A.M. Nielsen, "Pipelined Packet-Forwarding Floating Point: I. Foundations and a Rounder," *Proc. 13th IEEE Symp. Computer Arithmetic,* pp. 140-147, 1997.

[18] G. Jaberipur, B. Parhami, and M. Ghodsi, "An Efficient Universal Addition Scheme for All Hybrid-Redundant Representations with Weighted Bit-Set Encoding," *J. VLSI Signal Processing,* vol. 42, no. 2, pp. 149-158, Feb. 2006.

[19] M.C. Mekhallalati and M.K. Ibrahim, "New High Radix Maximally Redundant Signed Digit Adder," *Proc. Int'l Symp. Circuits and Systems,* vol. 1, pp. 459-462, 1999.

[20] G. Jaberipur and S. Gorgin, "A Nonspeculative One-Step Maximally Redundant Signed Digit Adder," *Comm. Computer and Information Science,* vol. 6, pp. 235-242, 2008.

[21] G. Jaberipur and B. Parhami, "Constant-Time Addition with Hybrid-Redundant Numbers: Theory and Implementations," *Integration: The VLSI J.,* vol. 41, no. 1, pp. 49-64, Jan. 2008.

[22] M. Daumas and D.W. Matula, "Further Reducing the Redundancy of a Notation over a Minimally Redundant Digit Set," *J. VLSI Signal Processing,* vol. 33, pp. 7-18, 2003.

[23] H.A.H. Fahmy and M.J. Flynn, "Rounding in Redundant Digit Floating Point Systems," *Proc. SPIE Conf. Algorithms, Architectures, and Devices,* Aug. 2003.

Ghassem Jaberipur received the BS degree in electrical engineering and the PhD degree in computer engineering from Sharif University of Technology in 1974 and 2004, respectively, the MS degree in engineering from the UCLA in 1976, and the MS degree in computer science from the University of Wisconsin, Madison, in 1979. He is currently an associate professor of computer engineering in the Department of Electrical and Computer Engineering of Shahid Beheshti University, Tehran, Iran. He is also affiliated with the School of Computer Science, Institute for Research in Fundamental Sciences, Tehran, Iran. His main research interest is in computer arithmetic.

Behrooz Parhami received the PhD degree from the University of California, Los Angeles, in 1973. He is currently a professor of electrical and computer engineering and associate dean for academic affairs, College of Engineering, at the University of California, Santa Barbara. He has research interests in computer arithmetic, parallel processing, and dependable computing. In his previous position with Sharif University of Technology, Tehran, Iran, during 1974-1988, he was also involved in educational planning, curriculum development, standardization efforts, technology transfer, and various editorial responsibilities, including a five-year term as editor of *Computer Report*, a Persian language computing periodical. His technical publications include more than 260 papers in peer-reviewed journals and international conferences, a Persian-language textbook, and an English/Persian glossary of computing terms. Among his publications are three textbooks on parallel processing (Plenum, 1999), computer architecture (Oxford Univ. Press, 2005), and computer arithmetic (Oxford Univ. Press, second ed., 2010). He is currently serving on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, and *International Journal of Parallel, Emergent and Distributed Systems*. He is a fellow of the IEEE, the IET, and the British Computer Society, a member of the Association for Computing Machinery, and a distinguished member of the Informatics Society of Iran for which he served as a founding member and president during 1979-1984. He also served as the chairman of the IEEE Iran Section during 1977-1986 and received the IEEE Centennial Medal in 1984.

Saeid Gorgin received the BS and MS degrees in computer engineering from the South Branch, and the Science and Research Branch, of Azad University of Tehran in 2001 and 2004, respectively. Since 2005, he has been working toward the PhD degree in the Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran. His research interests include computer arithmetic and VLSI design.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.