Delay-Optimized Implementation of IEEE Floating-Point Addition

Article *in* IEEE Transactions on Computers · March 2004

DOI: 10.1109/TC.2004.1261822 · Source: IEEE Xplore

2 authors:

Peter-Michael Seidel
University of Hawaiʻi at Mānoa
50 PUBLICATIONS **686** CITATIONS
SEE PROFILE

Guy Even
Tel Aviv University
135 PUBLICATIONS **3,988** CITATIONS
SEE PROFILE

Delay-Optimized Implementation of IEEE Floating-Point Addition

Peter-Michael Seidel and Guy Even

Abstract—We present an IEEE floating-point adder (FP-adder) design. The adder accepts normalized numbers, supports all four IEEE rounding modes, and outputs the correctly normalized rounded sum/difference in the format required by the IEEE Standard. The FP-adder design achieves a low latency by combining various optimization techniques such as: A nonstandard separation into two paths, a simple rounding algorithm, unification of rounding cases for addition and subtraction, sign-magnitude computation of a difference based on one's complement subtraction, compound adders, and fast circuits for approximate counting of leading zeros from borrow-save representation. We present technology-independent analysis and optimization of our implementation based on the Logical Effort hardware model and we determine optimal gate sizes and optimal buffer insertion. We estimate the delay of our optimized design at 30.6 FO4 delays for double precision operands (15.3 FO4 delays per stage between latches). We overview other IEEE FP addition algorithms from the literature and compare these algorithms with our algorithm. We conclude that our algorithm has shorter latency (-13 percent) and cycle time (-22 percent) compared to the next fastest algorithm.

Index Terms—Floating-point addition, IEEE rounding, delay optimization, dual path algorithm, logical effort, optimized gate sizing, buffer insertion.

1 Introduction and Summary

Floating-point addition and subtraction are the most frequent floating-point operations. Both operations use a floating-point adder (FP-adder). Therefore, a lot of effort has been spent on reducing the latency of FP-adders (see [2], [9], [20], [22], [23], [25], [27], [28], and the references that appear there). Many patents deal with FP-adder design (see [6], [10], [11], [14], [15], [19], [21], [31], [32], [35]).

We present an FP-adder design that accepts normalized double precision significands, supports all IEEE rounding modes, and outputs the normalized sum/difference that is rounded according to the IEEE FP standard 754 [13]. The latency of our design is analyzed using the Logical Effort Model [33]. This model allows for technology-independent delay analysis of CMOS circuits. The model enables rigorous delay analysis that takes into account fanouts, drivers, and gate-sizing. Following Horowitz [12], we use the delay of an inverter, the fanout of which equals 4, as a technology-independent unit of delay. An inverter with fanout 4 is denoted by FO4. Our analysis using the Logical Effort Model shows that the delay of our FP-adder design is 30.6 FO4 delays. This design is partitioned into two pipeline stages, the delay of which is bounded by 15.3 FO4 delays. Extensions of the algorithm that deal with denormal inputs and outputs are discussed in [16], [27]. It is shown there that the delay overhead for supporting denormal numbers can be reduced to 1-2 logic levels (i.e., XOR delays).

We employ several optimization techniques in our algorithm. A detailed examination of these techniques enables us to demonstrate how these techniques can be combined to achieve an overall fast FP-adder design. In particular, effective reduction of latency by parallel paths requires balancing the delay of the paths. We achieve such a balance by a gate-level consideration of the design. The optimization techniques that we use include the following techniques:

1. A two path design with a nonstandard separation criterion. Instead of separation based on the magnitude of the exponent difference [10], we define a separation criterion that also considers whether the operation is effective subtraction and the value of the significand difference. This separation criterion maintains the advantages of the standard two-path designs, namely, alignment shift and normalization shift take place only in one of the paths and the full exponent difference is computed only in one path. In addition, this separation technique requires rounding to take place only in one path.
2. Reduction of IEEE rounding to three modes [25] and use of injection based rounding [8].
3. A simpler design is obtained by using unconditional preshifts for effective subtractions to reduce to 2 the number of binades that the significands' sum and difference may belong to.
4. The sign-magnitude representation of the difference of the exponents and the significands is derived from one's complement representation of the difference.
5. A parallel-prefix adder is used to compute the sum and the incremented sum of the significands [34].
6. Recodings are used to estimate the number of leading zeros in the nonredundant representation of a number represented as a borrow-save number [20].
7. Postnormalization is advanced and takes place before the rounding decision is ready.

- *P.-M. Seidel is with the Computer Science and Engineering Department, Southern Methodist University, Dallas, TX, 75275. E-mail: seidel@acm.org.*
- *G. Even is with the Electrical Engineering Department, Tel-Aviv University, Tel-Aviv 69978, Israel. E-mail: guy@eng.tau.ac.il.*

Manuscript received 1 May 2001; revised 24 May 2002; accepted 6 Mar. 2003. For information on obtaining reprints of this article, please send e-mail to:

An overview of FP-adder algorithms from technical papers and patents is given. We summarize the optimization techniques that are used in each of these designs. We analyze the algorithms from two particular implementations from literature in some more detail [11], [21]. To allow for a "fair" comparison, the functionality of these designs are adopted to match the functionality of our design. Our design uses simpler rounding circuitry and is more amenable to partitioning into two pipeline stages of equal latency. We also analyze the latency of [21] using the Logical Effort Model. Our analysis gives the following results: 1) the total delay is 35.2 FO4 delays (i.e., 13 percent slower than our design) and 2) the delay of the slowest pipeline stage is 19.7 FO4 delays (i.e., 22 percent slower). This paper focuses on double precision FP-adder implementations. Many FP-adders support multiple precisions (e.g., x86 architectures support single, double, and extended double precision). In [27], it is shown that by aligning the rounding position (i.e., 23 positions to the right of the binary point in single precision and 52 positions to the right of the binary point in double precision) of the significands before they are input to the design and postaligning the outcome of the FP-adder, it is possible to use the FP-adder presented in this paper for multiple precisions. Hence, the FP-addition algorithm presented in this paper can be used to support multiple precisions.

Testing of our FP-adder design was done by exhaustive testing of a reduced precision version of our design [1]. More details about testing of our algorithm are provided in Section 7.

2 Notation

Values and their representation. We denote binary strings in upper case letters (e.g., S, E, F). The value represented by a binary string is represented in italics (e.g., s, e, f).

IEEE FP-numbers. We consider normalized IEEE FP-numbers. In double precision, IEEE FP-numbers are represented by three fields $(\mathrm{S}, \mathrm{E}[10:0], \mathrm{F}[0:52])$ with *sign bit* $\mathrm{S} \in \{0,1\}$, exponent string $\mathrm{E}[10:0] \in \{0,1\}^{11}$ and significand string $\mathrm{F}[0:52] \in \{0,1\}^{53}$. The values of exponent and significand are defined by: $e = \sum_{i=0}^{10} \mathrm{E}[i] \cdot 2^i - 1023$, $f = \sum_{i=0}^{52} \mathrm{F}[i] \cdot 2^{-i}$. Since we only consider normalized FP-numbers, we have $f \in [1,2)$.

The value represented by a FP-number $(\mathrm{S}, \mathrm{E}[10:0], \mathrm{F}[0:52])$ is: $fp\_val(\mathrm{S}, \mathrm{E}, \mathrm{F}) = (-1)^{\mathrm{S}} \cdot 2^e \cdot f$.

Factorings. Given an IEEE FP-number $(\mathrm{S}, \mathrm{E}, \mathrm{F})$, we refer to the triple (s, e, f) as the *factoring* of the FP-number. Note that $s = \mathrm{S}$ since S is a single bit. The advantage of using factorings is the ability to ignore representation details and focus on values.

Inputs. The inputs of a FP-addition/subtraction are:

1. Operands denoted by $(\mathrm{SA}, \mathrm{EA}[10:0], \mathrm{FA}[0:52])$ and $(\mathrm{SB}, \mathrm{EB}[10:0], \mathrm{FB}[0:52])$.
2. Operation denoted by $\mathrm{SOP} \in \{0,1\}$ to indicate addition/subtraction.
3. IEEE rounding mode.

Output. The output is a FP-number $(\mathrm{S}, \mathrm{E}[10:0], \mathrm{F}[0:52])$. The value represented by the output equals the IEEE rounded value of

$$fpsum = fp\_val(\mathrm{SA}, \mathrm{EA}[10:0], \mathrm{FA}[0:52]) + (-1)^{\mathrm{SOP}} fp\_val(\mathrm{SB}, \mathrm{EB}[10:0], \mathrm{FB}[0:52]).$$

3 Naive FP-Adder Algorithm

In this section, we overview the "vanilla" FP-addition algorithm. To simplify notation, we ignore representation and deal only with the values of the inputs, outputs, and intermediate results. Throughout the paper, we use the notation defined for the naive algorithm.

Let (sa, ea, fa) and (sb, eb, fb) denote the factorings of the operands with a sign-bit, an exponent, and a significand, and let SOP indicate whether the operation is an addition or a subtraction. The requested computation is the IEEE FP representation of the rounded sum:

$$rnd(sum) = rnd((-1)^{sa} \cdot 2^{ea} \cdot fa + (-1)^{\mathrm{SOP}+sb} \cdot 2^{eb} \cdot fb).$$

Let $\mathrm{S.EFF} = sa \oplus sb \oplus \mathrm{SOP}$. The case that $\mathrm{S.EFF} = 0$ is called *effective addition* and the case that $\mathrm{S.EFF} = 1$ is called *effective subtraction*.

We define the exponent difference $\delta = ea - eb$. The "large" operand, (sl, el, fl), and the "small" operand, (ss, es, fs), are defined as follows:

$$(sl, el, fl) = \begin{cases} (sa, ea, fa) & \text{if } \delta \geq 0 \\ (\mathrm{SOP} \oplus sb, eb, fb) & \text{otherwise} \end{cases}$$

$$(ss, es, fs) = \begin{cases} (\mathrm{SOP} \oplus sb, eb, fb) & \text{if } \delta \geq 0 \\ (sa, ea, fa) & \text{otherwise.} \end{cases}$$

The sum can be written as:

$$sum = (-1)^{sl} \cdot 2^{el} \cdot (fl + (-1)^{\mathrm{S.EFF}} (fs \cdot 2^{-|\delta|})).$$

To simplify the description of the datapaths, we focus on the computation of the result's significand, which is assumed to be normalized (i.e., in the range $[1,2)$ after rounding). The significand sum is defined by

$$fsum = fl + (-1)^{\mathrm{S.EFF}} (fs \cdot 2^{-|\delta|}).$$

The significand sum is computed, normalized, and rounded as follows:

1. exponent subtraction: $\delta = ea - eb$,
2. operand swapping: compute sl, el, fl, and fs,
3. limitation of the alignment shift amount: $\delta\_lim = \min\{\alpha, abs(\delta)\}$, where α is a constant greater than or equal to 55,
4. alignment shift of fs: $fsa = fs \cdot 2^{-\delta\_lim}$,
5. significand negation: $fsan = (-1)^{\mathrm{S.EFF}} fsa$;
6. Significand addition: $fsum = fl + fsan$;
7. conversion: $abs\_fsum = abs(fsum)$ (the sign S of the result is determined by $\mathrm{S} = sl \oplus (fsum < 0)$),
8. normalization shift $n\_fsum = norm(abs\_fsum)$, and
9. rounding and postnormalization of n_fsum.

The naive FP-adder implements the nine steps from above sequentially, where the delay of Steps 4, 6, 7, 8, 9 is logarithmic in the significand's length. Therefore, this is a slow FP-adder implementation.

4 Optimization Techniques

In this section, we outline optimization techniques that were employed in our FP-adder design.

4.1 Separation of FP-Adder into Two Parallel Paths

The FP-adder pipeline is separated into two parallel paths that work under different assumptions. The partitioning into two parallel paths enables one to optimize each path separately by simplifying and skipping some steps of the naive addition algorithm (Section 3). Such a dual path approach for FP-addition was first described by Farmwald [9]. Since Farmwald's dual path FP-addition algorithm, the common criterion for partitioning the computation into two paths has been the exponent difference. The exponent difference criterion is defined as follows: The *near path* is defined for small exponent differences (i.e., $-1, 0, +1$), and the *far path* is defined for the remaining cases.

We use a different partitioning criterion for partitioning the algorithm into two paths: we define the *N-path* for the computation of all effective subtractions with small significand sums $fsum \in (-1, 1)$ and small exponent differences $|\delta| \leq 1$, and we define the *R-path* for all the remaining cases. We define the path selection signal IS_R as follows:

$$\text{IS\_R} \iff \overline{\text{S.EFF}} \text{ OR } |\delta| \geq 2 \text{ OR } fsum \in [1, 2). \qquad (1)$$

The outcome of the R-path is selected for the final result if IS_R $=1$, otherwise the outcome of the N-path is selected. This partitioning has the following advantages:

1. In the R-path, the normalization shift is limited to a shift by one position (in Section 4.2, we show how the normalization shift may be restricted to one direction). Moreover, the addition or subtraction of the significands in the R-path always results with a positive significand and, therefore, the conversion step can be skipped.
2. In the N-path, the alignment shift is limited to a shift by one position to the right. Under the assumptions of the N-path, the exponent difference is in the range $\{-1, 0, 1\}$. Therefore, a 2-bit subtraction suffices for extracting the exponent difference. Moreover, in the N-path, the significand difference can be exactly represented with 53 bits, hence, no rounding is required.

Note that the N-path applies only to effective subtractions in which the significand difference $fsum$ is less than 1. Thus, in the N-path it is assumed that $fsum \in (-1, 1)$.

The advantages of our partitioning criterion compared to the exponent difference criterion stem from the following two observations: 1) A conventional implementation of a far path can be used to implement also the R-path and 2) The N-path is simpler than the near path since no rounding is required and the N-path applies only to effective subtractions. Hence, the N-path is simpler and faster than the near path presented in [28].

4.2 Unification of Significand Result Ranges

In the R-path, the range of the resulting significand is different in effective addition and effective subtraction. Using the notation of Section 3, in effective addition, $fl \in [1, 2)$ and $fsan \in [0, 2)$. Therefore, $fsum \in [1, 4)$. It follows from the definition of the path selection condition that in effective subtractions $fsum \in (\frac{1}{2}, 2)$ in the R-path. We unify the ranges of $fsum$ in these two cases to $[1, 4)$ by multiplying the significands by 2 in the case of effective subtraction (i.e., preshifting by one position to the left). The unification of the range of the significand sum in effective subtraction and effective addition simplifies the rounding circuitry. To simplify the notation and the implementation of the path selection condition, we also preshift the operands for effective subtractions in the N-path. Note that, in this way, the preshift is computed in the N-path unconditionally, because in the N-path all operations are effective subtractions. In the following, we give a few examples of values that include the conditional preshift (note that an additional "p" is included in the names of the preshifted versions):

$$flp = \begin{cases} 2 \cdot fl & \text{if S.EFF} \\ fl & \text{otherwise,} \end{cases}$$
$$fspan = \begin{cases} 2 \cdot fsan & \text{if S.EFF} \\ fsan & \text{otherwise,} \end{cases}$$
$$fpsum = \begin{cases} 2 \cdot fsum & \text{if S.EFF} \\ fsum & \text{otherwise.} \end{cases}$$

Note that, based on the significand sum *fpsum*, which includes the conditional preshift, the path selection condition (1) can be rewritten as

$$\text{IS\_R} \iff \overline{\text{S.EFF}} \text{ OR } |\delta| \geq 2 \text{ OR } fpsum \in [2, 4). \qquad (2)$$

4.3 Reduction of IEEE Rounding Modes

The IEEE-754-1985 Standard defines four rounding modes: Round toward 0, round toward $+\infty$, round toward $-\infty$, and round to nearest (even) [13]. Following Quach et al. [25], we reduce the four IEEE rounding modes to three rounding modes: Round-to-zero RZ, round-to-infinity RI, and round-to-nearest-up RNU. The discrepancy between round-to-nearest-even and RNU is fixed by pulling down the LSB of the fraction (see [8] for more details).

In the rounding implementation in the R-path, the three rounding modes RZ, RNU, and RI are further reduced to truncation using injection based rounding [8]. The reduction is based on adding an injection that depends only on the rounding mode. Let $\mathrm{X} = \mathrm{X}_0.\mathrm{X}_1\mathrm{X}_2\ldots\mathrm{X}_k$ denote the binary representation of a significand with the value $x = |\mathrm{X}| \in [1, 2)$ for which $k \geq 53$ (double precision rounding is trivial for $k < 53$), then the injection is defined by:

$$inj = \begin{cases} 0 & \text{if RZ} \\ 2^{-53} & \text{if RNU} \\ 2^{-52} - 2^{-k} & \text{if RI.} \end{cases}$$

For double precision and $mode \in \{RZ, RNU, RI\}$, the effect of adding *inj* is summarized by:

$$|\mathrm{X}| \in [1, 2) \quad \Rightarrow \quad rnd_{mode}(|\mathrm{X}|) = rnd_{RZ}(|\mathrm{X}| + inj). \qquad (3)$$

4.4 Sign-Magnitude Computation of a Difference

In this technique, the sign-magnitude computation of a difference is computed using one's complement representation [22]. This technique is applied in two situations:

1. Exponent difference. The sign-magnitude representation of the exponent difference is used for two purposes:
   a. the sign determines which operand is selected as the "large" operand; and
   b. the magnitude determines the amount of the alignment shift.
2. Significand difference. In case the exponent difference is zero and an effective subtraction takes place, the significand difference might be negative. The sign of the significand difference is used to update the sign of the result and the magnitude is normalized to become the result's significand.

Let A and B denote binary strings and let $|\mathrm{A}|$ denote the value represented by A (i.e., $|\mathrm{A}| = \sum_i \mathrm{A}[i] \cdot 2^i$). The technique is based on the following observation:

$$abs(|\mathrm{A}| - |\mathrm{B}|) = \begin{cases} |\mathrm{A}| + |\overline{\mathrm{B}}| + 1 & \text{if } |\mathrm{A}| - |\mathrm{B}| > 0 \\ |\mathrm{A}| + |\overline{\mathrm{B}}| & \text{if } |\mathrm{A}| - |\mathrm{B}| \leq 0. \end{cases}$$

The actual computation proceeds as follows: The binary string D is computed such that $|\mathrm{D}| = |\mathrm{A}| + |\overline{\mathrm{B}}|$. We refer to D as the *one's complement lazy* difference of A and B. We consider two cases:

1. If the difference is positive, then $|\mathrm{D}|$ is off by an ULP and we need to increment $|\mathrm{D}|$. However, to save delay, we avoid the increment as follows:
   a. In the case of the exponent difference that determines the alignment shift amount, the significands are preshifted by one position to compensate for the error.
   b. In the case of the significand difference, the missing ULP is provided by computing the incremented sum of $|\mathrm{A}|$ and $|\overline{\mathrm{B}}|$ using a compound adder.
2. If the exponent difference is negative, then the bits of D are negated to obtain an exact representation of the magnitude of the difference.

4.5 Compound Addition

The technique of computing in parallel the sum of the significands as well as the incremented sum is well known. The rounding decision controls which of the sums is selected for the final result, thus enabling the computation of the sum and the rounding decision in parallel.

Technique. We follow the technique suggested by Tyagi [34] for implementing a compound adder. This technique is based on a parallel prefix adder in which the carry-generate and carry-propagate strings, denoted by Gen_C and $Prop\_C$, are computed [3]. Let $Gen\_C[i]$ equal the carry bit that is fed to position i. The bits of the sum S of the addends A and B are obtained as usual by:

$$S[i] = \mathrm{XOR}(A[i], B[i], Gen\_C[i]).$$

The bits of the incremented sum SI are obtained by:

$$SI[i] = \mathrm{XOR}(A[i], B[i], \mathrm{OR}(Gen\_C[i], Prop\_C[i])).$$

Application. There are two instances of a compound adder in our FP-addition algorithm. One instance appears in the second pipeline stage of the R-path where our delay analysis relies on the assumption that the MSB of the sum is valid one logic level prior to the slowest sum bit.

The second instance of a compound adder appears in the N-path. In this case, we also address the problem that the compound adder does not "fit" in the first pipeline stage according to our delay analysis. We break this critical path by partitioning the compound adder between the first and second pipeline stages as follows: A parallel prefix adder placed in the first pipeline stage computes the carry-generate and carry-propagate signals as well as the bitwise XOR of the addends. From these three binary strings, the sum and incremented sum are computed within two logic levels as described above. However, these two logic levels must belong to different pipeline stages. We, therefore, compute first, the three binary strings $S[i]$, $P[i] = A[i]\ \mathrm{XOR}\ B[i]$ and $GP\_C[i] = \mathrm{OR}\ (Gen\_C[i], Prop\_C[i])$, which are passed to the second pipeline stage. In this way, the computation of the sum is already completed in the first pipeline stage and only an XOR-line is required in the second pipeline stage to compute also the incremented sum.

4.6 Approximate Counting of Leading Zeros

In the N-path, a resulting significand in the range $(-1, 1)$ must be normalized. The amount of the normalization shift is determined by approximating the number of leading zeros. Following Nielsen et al. [20], we approximate the number of leading zeros so that a normalization shift by this amount yields a significand in the range $[1, 4)$. The final normalization is then performed by post-normalization. The input used for counting leading zeros in our design is a borrow-save representation of the difference. This design is amenable to partitioning into pipeline stages, and admits an elegant correctness proof that avoids a tedious case analysis.

Nielsen et. al. [20] presented the following technique for approximately counting the number of leading zeros. The method is based on a constant delay reduction of a borrow-save encoded string to a binary vector. The number of leading zeros in the binary vector almost equals the number of leading zeros in the binary representation of the absolute value of the number represented by the borrow-save encoded string. We describe this reduction below.

The input consists of a borrow-save encoded digit string $\mathrm{F}[-1\!:\!52] \in \{-1, 0, 1\}^{54}$. We compute the borrow-save encoded string $\mathrm{F}'[-2\!:\!52] = P(N(\mathrm{F}[-1\!:\!52]))$, where $P()$ and $N()$ denote P-recoding and N-recoding [5], [20]. (P-recoding is like a "signed half-adder" in which the carry output has a positive sign, N-recoding is similar but has an output carry with a negative sign). The correctness of the technique is based on the following claim.

Claim 1. [20] Suppose the borrow-save encoded string $\mathrm{F}'[-2\!:\!52]$ is of the form $\mathrm{F}'[-2\!:\!52] = 0^k \cdot \sigma \cdot t[1\!:\!54-k]$, where $\cdot$ denotes concatenation of strings, 0^k denotes a block of k zeros, $\sigma \in \{-1, 1\}$, and $t \in \{-1, 0, 1\}^{54-k}$.

Then, the following holds:

1. If $\sigma = 1$, then the value represented by the borrow-save encoded string $\sigma.t$ satisfies:

$$\sigma + \sum_{i=1}^{54-k} t[i] \cdot 2^{-i} \in \left(\frac{1}{4}, 1\right).$$

2. If $\sigma = –1$, then the value represented by the borrow-save encoded string $\sigma.t$ satisfies:

$$\sigma + \sum_{i=1}^{54-k} t[i] \cdot 2^{-i} \quad \in \quad \left(-\frac{3}{2}, -\frac{1}{2}\right).$$

The implication of Claim 1 is that after PN-recoding, the number of leading zeros in the borrow-save encoded string $\mathrm{F}'[-2:53]$ (denoted by k in the claim) can be used as the normalization shift amount to bring the normalized result into one of two binades (i.e., in the positive case either $(\frac{1}{4}, \frac{1}{2})$ or $[\frac{1}{2}, 1)$, and in the negative case after negation either $(\frac{1}{2}, 1)$ or $[1, \frac{3}{2})$). Since we are only interested in the number of leading zeros of $\mathrm{F}'[-2:53]$, we may reduce this borrow-save encoded string to a binary string by applying bitwise-XOR. We implemented this technique so that the normalized significand is in the range $[1, 4)$ as follows:

1. In the positive case, the shift amount is $lz2 = k = lzero(\mathrm{F}'[–2:52])$. (See signal $LZP2[5:0]$ in Fig. 3).
2. In the negative case, the shift amount is $lz1 = k–1 = lzero(\mathrm{F}'[–1:52])$. (See signal $LZP1[5:0]$ in Fig. 3).

The reduction of the borrow-save encoded string to the binary vector involves a P-recoding, an N-recoding, and bitwise XOR. Hence, the cost per bit is two half-adders and a XOR-gate (i.e., 3 XOR-gates and two AND-gates). The delay of this reduction is 3 XOR-gates.

In a recent survey of leading zeroes anticipation and detection by Schmookler and Nowka [26], a method that is faster by roughly one logic level is presented. The steps of this method are as follows:

1. Transform the negative vector of the borrow-save encoded string into a positive vector by using 2's complement negation (the increment can be avoided by padding a carry-save 2 digit from the right). Hence, the problem of computing the normalization shift amount is reduced to counting the number of leading zeros (ones) of the binary representation of the sum in case the sum is positive (negative).
2. The position of the leftmost 1 in the binary representation of the sum (when this sum is positive) is computed as follows: Reduce the carry-save encoded string into a binary string as follows: $\alpha_i = \mathrm{XOR}(P_i, \overline{K_{i+1}})$, where P_i is the ith "propagate"-bit (namely, P_i is the XOR of the bits in position i of the carry-save encoded string) and K_i is the ith "kill"-bit (namely, the NOR of the above bits). The position of the leftmost one in the vector α_i is either the position of the leading digit or one position to the right of the leading digit.
3. In case the sum is negative, the position of the leftmost zero is computed by the following reduction: $\beta_i = \mathrm{XOR}(P_i, \overline{G_{i+1}})$, where G_i is the ith "generate"-bit (namely, the AND of the bits in position i of the carry-save encoded string).
4. Each of the binary vectors α and β are fed to a priority encoder.

The cost of this method is five gates per bit (i.e., 3 XOR-gates, one NOR-gate, and one AND-gate). The delay is two logic levels (two XOR-gates).

A further improvement described in [26] is to the following reductions: $\alpha'_i = \mathrm{AND}(\overline{P_i}, \overline{K_{i+1}})$ and $\beta'_i = \mathrm{AND}(\overline{P_i}, \overline{G_{i+1}})$. This improvement reduces the cost per bit to three AND-gates, one XOR-gate, and one NOR-gate. The delay is reduced to one XOR-gate plus an AND-gate.

4.7 Precomputation of Postnormalization Shift

In the R-path, two choices for the rounded significand sum are computed by the compound adder (see Section 4.5). Either the "sum" or the "incremented sum" output of the compound adder is chosen for the rounded result. Because the significand after the rounding selection is in the range $[1, 4)$ (due to the preshifts from Section 4.2 only these two binades have to be considered for rounding and for the postnormalization shift), postnormalization requires at most a right-shift by one bit position. Because the outputs of the compound adder have to wait for the computation of the rounding decision (selection based on the range of the sum output), we precompute the postnormalization shift on both outputs of the compound adder before the rounding selection, so that the rounding selection already outputs the normalized significand result of the R-path.

5 Our FP-Adder Algorithm

5.1 Overview

In this section, we give an overview of our FP-adder algorithm. We describe the partitioning of the design implementing and integrating the optimization techniques from the previous section. The algorithm is a two-staged pipeline partitioned into two parallel paths called the R-path and the N-path. The final result is selected between the outcomes of the two paths based on the signal IS_R (see (2)). Several block diagrams of our algorithm are depicted in Fig. 1. We give an overview of the two paths in the following.

Overview R-Path. The R-path works under the assumption that 1) an effective addition takes place or 2) an effective subtraction with a significand difference (after preshifting) greater than or equal to 2 takes place or 3) the absolute value of the exponent difference $|\delta|$ is larger than or equal to 2. Note that these assumptions imply that the sign-bit of the sum equals SL.

The algorithm performs various operations on the significands before adding them. These operations include: swapping, negation, preshifting, and alignment. For the sake of brevity, we refer in the outline to a bit string that originates from the significands fs or fl as the "small operand" or "large operand", respectively.

The R-path is divided into two pipeline stages. In the first pipeline stage, the exponent difference is computed and the significands are swapped. In effective subtraction, both significands are preshifted and only the small operand is negated. The small operand is then aligned according to the exponent difference.

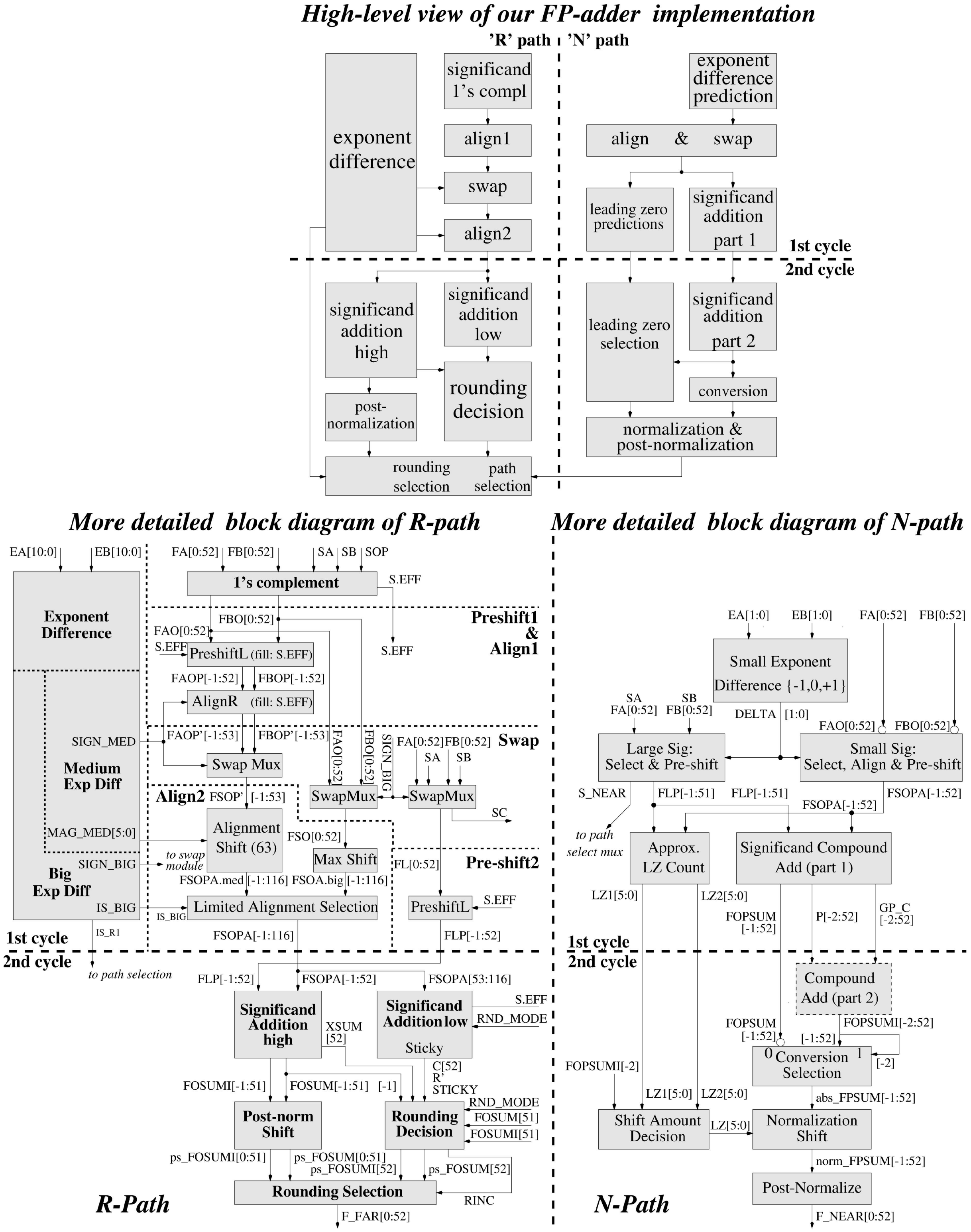


Fig. 1. Structure of our FP addition algorithm in different levels. Vertical dashed line separates two pipelines: R-path and N-path. Horizontal dashed line separates two pipeline stages.

In the *Significand One's Complement* box, the small operand is negated (recall that one's complement representation is used). In the *Align 1* box, the small operand is 1) preshifted to the right if an effective subtraction takes place and 2) aligned to the left by one position if the exponent difference is positive. The alignment to the left by one position compensates for the error in the computation of the exponent difference when the difference is positive due to the one's complement representation (see Section 4.4). In the *Swap* box, the large and small operands are selected according to the sign of the exponent difference. Note that since swapping takes place only after complementation and

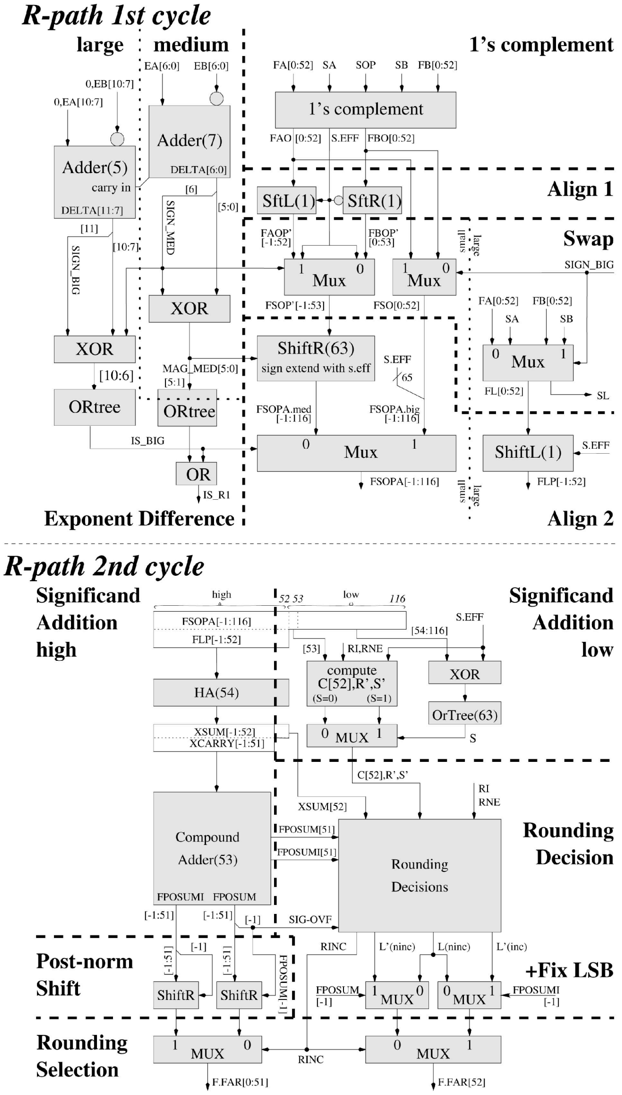


Fig. 2. Detailed block diagram of the first clock cycle and the second clock cycle of the R-path.

Align 1, one must consider complementing and aligning both significands; only one of these significands is used as the small operand. In the *Align2* box, the small operand is aligned according to the computed exponent difference. The *exponent difference* box computes the swap decision and the alignment shift amount. This box is further partitioned into two paths for medium and large exponent differences. A detailed block diagram for the implementation of the first cycle of the R-path is depicted in Fig. 2.

The input to the second pipeline stage consists of the large and small operands. The goal is to compute their sum and round it while taking into account the error due to the one's complement representation for effective subtractions. The second pipeline stage is very similar to the rounding algorithm presented in [8]. The large and small operands are divided into a low part and a high part that are processed in parallel. The LSB of the final result is computed based on the low part and the range of the sum. The rest of the final result is computed based on the high part (i.e., either the sum or the incremented sum of the high part). The outputs of the compound adder are postnormalized before the rounding selection is performed. A detailed block diagram for the implementation of the second cycle of the R-path is depicted in Fig. 2.

Overview N-Path. The N-path works under the assumption that an effective subtraction takes place, the significand difference (after the swapping of the addends and preshifting) is less than 2, and the absolute value of the exponent difference $|\delta|$ is less than 2. The N-path has the following properties:

1. The exponent difference must be in the set $\{-1, 0, 1\}$. Hence, the exponent difference can be computed by subtracting the two LSBs of the exponent strings. The alignment shift is by at most one position. This is implemented in the *exponent difference prediction* box.
2. An effective subtraction takes place, hence, the small operand is always negated. We use one's complement representation for the negated subtrahend.
3. The significand difference (after swapping and preshifting) is in the range $(-2, 2)$ and can be exactly represented using 52 bits to the right of the binary point. Hence, no rounding is required

Based on the exponent difference prediction, the significands are swapped and aligned by at most one bit position in the *align and swap* box. The leading zero approximation and the significand difference are then computed in parallel. The result of the leading zero approximation is selected based on the sign of the significand difference according to Section 4.6 in the *leading zero selection* box. The *conversion* box computes the absolute value of the difference (Section 4.4) and the *normalization and postnormalization* box normalizes the absolute significand difference as a result of the N-path. Fig. 3 depicts a detailed block diagram of the N-path.

5.2 Specification

In this section, we specify the computations in the two computation paths. We specify the N-path and the two stages of the R-path.

5.2.1 Specification: R-Path First Cycle

The computation performed by the first pipeline stage in the R-path outputs the significands flp and $fsopa$, represented by $\mathrm{FLP}[-1:52]$ and $\mathrm{FSOPA}[-1:116]$. The significands flp and $fsopa$ are defined by

$$(flp, fsopa) = \begin{cases} (fl, fsan) & \text{if S.EFF} = 0 \\ (2 \cdot fl, 2 \cdot fsna - 2^{-116}) & \text{otherwise.} \end{cases}$$

The detailed view in Fig. 1 depicts how the computation of $\mathrm{FLP}[-1:52]$ and $\mathrm{FSOPA}[-1:116]$ is performed. For each box in the high level view, a region surrounded by dashed lines is depicted to assist the reader in matching the regions with blocks.

1. The exponent difference is computed for two ranges: The *medium exponent difference interval* consist of $[-63, 64]$, and the *big exponent difference intervals* consist of $(-\infty, -64]$ and $[65, \infty]$. The outputs of the *exponent difference* box are specified as follows: Loosely speaking, the SIGN_MED and MAG_MED are the sign-magnitude representation of δ, if δ is in the medium exponent difference interval. Formally,

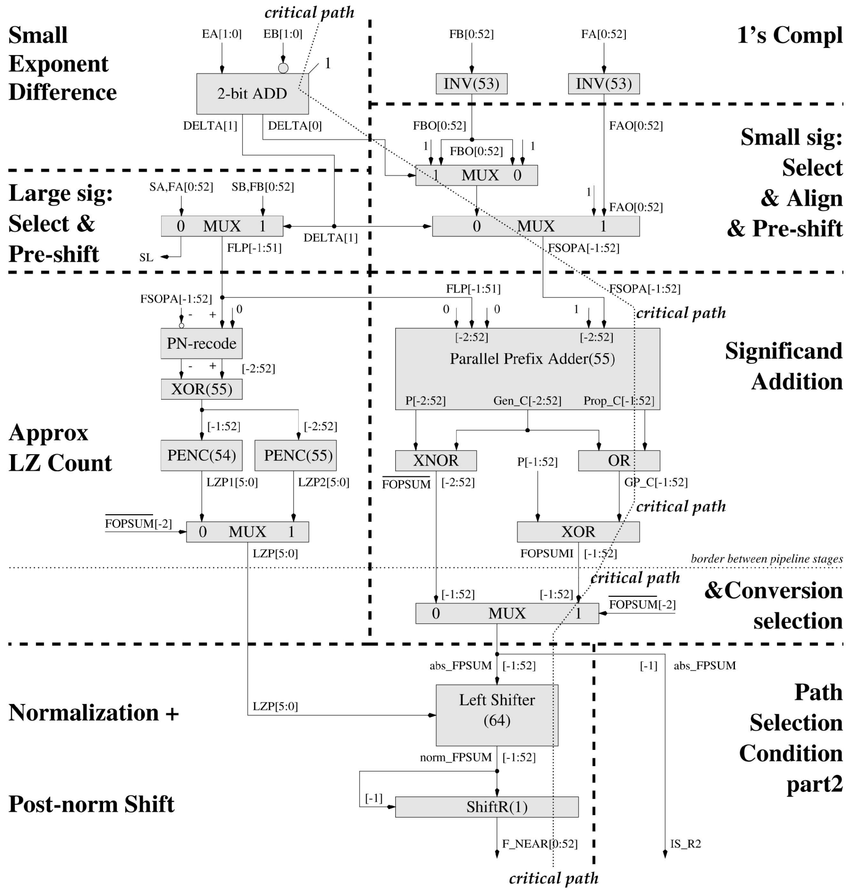


Fig. 3. Detailed block diagram of the N-path.

$$(-1)^{\mathrm{SIGN\_MED}} \cdot \langle \mathrm{MAG\_MED} \rangle = \begin{cases} \delta - 1 & \text{if } 64 \geq \delta \geq 1 \\ \delta & \text{if } 0 \geq \delta \geq -63 \\ \text{"don't-care"} & \text{otherwise.} \end{cases}$$

The reason for missing δ by 1 in the positive case is due to the one's complement subtraction of the exponents. This error term is compensated for in the *Align1* box.

2. SIGN_BIG is the sign bit of exponent difference δ. IS_BIG is a flag defined by:

$$\mathrm{IS\_BIG} = \begin{cases} 1 & \text{if } \delta \geq 65 \text{ or } \delta \leq -64 \\ 0 & \text{otherwise.} \end{cases}$$

3. In the big exponent difference intervals, the "required" alignment shift is at least 64 positions. Since all alignment shifts of 55 positions or more are equivalent (i.e., beyond the sticky-bit position), we may limit the shift amount in this case. In the *Align2* region, one of the following alignment shift occurs:
   a. a fixed alignment shift by 64 positions in case the exponent difference belongs to the big exponent difference intervals (this alignment ignores the preshifting altogether) or
   b. an alignment shift by *mag_med* positions in case the exponent difference belongs to the medium exponent difference interval.
4. In the *One's Complement* box, the signals FAO, FBO, and S.EFF are computed. The FAO and FBO signals are defined by

$$\mathrm{FAO}[0:52], \mathrm{FBO}[0:52] = \begin{cases} \mathrm{FA}[0:52], \mathrm{FB}[0:52] & \text{if S.EFF} = 0 \\ not(\mathrm{FA}[0:52]), not(\mathrm{FB}[0:52]) & \text{otherwise.} \end{cases}$$

5. The computations performed in the *Preshift and Align 1* region are relevant only if the exponent difference is in the medium exponent difference interval. The significands are pre-shifted if an effective subtraction takes place. After the pre-shifting, an alignment shift by one position takes place if $sign\_med = 1$. Table 1 summarizes the specification of FSOP'$[-1:53]$.
6. In the *Swap* region, the large operand is selected based on $sign\_big$. The small operand is selected for the medium exponent difference (based on $sign\_med$) interval and for the large exponent difference interval (based on $sign\_big$).
7. The *Preshift 2* region deals with preshifting the minuend in case an effective subtraction takes place.

5.2.2 Specification: R-path Second Cycle

The input to the second cycle consists of: The sign bit SL, a representation of the exponent el, the significand strings FLP$[-1:52]$ and FSOPA$[-1:116]$, and the rounding mode.

TABLE 1
Value of FSOP' $[-1:53]$ According to Fig. 1

SIGN_MED	S.EFF	pre-shift (left)	align-shift (right)	accumulated right shift	FSOP'$[-1:53]$
0	0	0	1	1	$(00, \text{FBO}[0:52])$
	1	1	1	0	$(1, \text{FBO}[0:52], 1)$
1	0	0	0	0	$(0, \text{FAO}[0:52], 0)$
	1	1	0	-1	$(\text{FAO}[0:52], 11)$

Together with the sign bit SL, the rounding mode is reduced to one of the three rounding modes: RZ, RNE or RI (see Section 4.3). The output consists of the sign bit SL, the exponent string (the computation of which is not discussed here), and the normalized and rounded significand $f\_far \in [1,2)$ represented by $\text{F\_FAR}[0:52]$. If the significand sum (after pre-shifting) is greater than or equal to 1, then the output of the second cycle of the R-path satisfies:

$$\begin{aligned} rnd(fsum) &= rnd\Big((-1)^{sl}\cdot(flp+fsopa+\text{S.EFF}\cdot 2^{-116})\Big) \\ &= (-1)^{sl}\cdot f\_far. \end{aligned}$$

Note that, in effective subtraction, 2^{-116} is added to correct the sum of the one's complement representations to the sum of two's complement representations by the lazy increment from the first clock cycle. Fig. 1 depicts the partitioning of the computations in the 2nd cycle of the R-path into basic blocks and specifies the input- and output-signals of each of these basic blocks.

5.2.3 Specification: N-Path.

A detailed block diagram of the N-path and the central signals are depicted in Fig. 1.

1. The *Small Exponent Difference* box outputs DELTA $[1:0]$ representing $ea - eb$.
2. The input to the *Small Significands: Select, Align, and Preshift* box consists of the inverted significand strings FAO and FBO. The selection means that if the exponent difference equals -1, then the subtrahend corresponds to FA, otherwise it corresponds to FB. The preshifting means that the significands are preshifted by one position to the left (i.e. multiplied by 2). The alignment means that if the absolute value of the exponent difference equals 1, then the subtrahend needs to be shifted to the right by one position (i.e., divided by 2). The output signal FSOPA is therefore specified by

$$\text{FSOPA}[-1:52] = \begin{cases} (0, \text{FA}[0:52]) & \text{if } ea - eb = -1 \\ (\text{FB}[0:52], 0) & \text{if } ea - eb = 0 \\ (0, \text{FB}[0:52]) & \text{if } ea - eb = 1. \end{cases}$$

   Note that $\text{FSOPA}[-1:52]$ is the one's complement representation of $-2 \cdot fs/2^{abs(ea-eb)}$.
3. The *Large Significands: Select and Preshift* box outputs the minuend $\text{FLP}[-1:51]$ and the sign-bit of the addend it corresponds to. The selection means that if the exponent difference equals -1, then the minuend corresponds to FB, otherwise it corresponds to FA. The preshifting means that the significands are preshifted by one position to the left (i.e., multiplied by 2). The output signal FSOPA is therefore specified by

$$\text{FLP}[-1:51], \text{SL} = \begin{cases} \text{FB}[0:52],\ \text{SB} & \text{if } ea - eb = -1 \\ \text{FA}[0:52],\ \text{SA} & \text{if } ea - eb \geq 0. \end{cases}$$

   Note that $\text{FLP}[-1:51]$ is the binary representation of $2 \cdot fl$. Therefore:

$$\begin{aligned} flp + fsopa &= 2(fl - fs/2^{abs(ea-eb)}) - 2^{-52} \\ &= fpsum - 2^{-52}. \end{aligned}$$

4. The *Approximate LZ count* box outputs two estimates, $\ell zp1, \ell zp2$ of the number of leading zeros in the binary representation of $abs(fpsum)$. The estimates $\ell zp1, \ell zp2$ satisfy the following property:

$$\begin{aligned} -fpsum \cdot 2^{\ell zp1} \in [1,4) \quad & \text{if } fpsum < 0 \\ fpsum \cdot 2^{\ell zp2} \in [1,4) \quad & \text{if } fpsum > 0. \end{aligned}$$

5. The *Shift Amount Decision* box selects the normalization shift amount between $\ell zp1$ and $\ell zp2$ depending on the sign of the significand difference as follows:

$$\ell zp = \begin{cases} \ell zp1 & \text{if } fpsum < 0 \\ \ell zp2 & \text{if } fpsum > 0. \end{cases}$$

6. The *Significand Compound Add* boxes parts 1 and 2 together with the *Conversion Selection* box compute the sign and magnitude of $fpsum = flp + fsopa + 2^{-52}$. The magnitude of $fpsum$ is represented by the binary string $abs\_\text{FPSUM}[-1:52]$ and the sign of the sum is represented by $\text{FSOPUMI}[-2]$. The method of how the sign and magnitude are computed is described in Section 4.4.
7. The *Normalization Shift* box shifts the binary string $abs\_\text{FPSUM}[-1:53]$ by ℓzp positions to the left, padding in zeros from the right. The normalization shift guarantees that *norm_fpsum* is in the range $[1,4)$.
8. The *Postnormalize* box which outputs:

$$f\_near = \begin{cases} norm\_fpsum & \text{if } norm\_fpsum \in [1,2) \\ norm\_fpsum/2 & \text{if } norm\_fpsum \in [2,4). \end{cases}$$

5.3 Detailed Description

In this section, we describe our FP-adder algorithm in detail. We separately discuss the implementation of the first stage and the second stage of the R-Path, the implementation of the N-path and the implementation of the path selection condition.

5.3.1 Details: R-Path First Cycle

Fig. 2 depicts a detailed block diagram of the first cycle of the R-path. The nonstraightforward regions are described below.

1. The *Exponent Difference* region is implemented by cascading a 7-bit adder with a 5-bit adder. The 7-bit adder computes the lazy one's complement exponent difference if the exponent difference is in the medium interval. This difference is converted to a sign and magnitude representation denoted by *sign_med* and *mag_med*. The cascading of the adders enables us to evaluate the exponent difference (for the medium interval) in parallel with determining whether the exponent difference is in the big range. The SIGN_BIG signal is simply the MSB of the lazy one's complement exponent difference. The IS_BIG signal is computed by OR-ing the bits in positions $[6:10]$ of the magnitude of the lazy one's complement exponent difference. This explains why the medium interval is not symmetric around zero.
2. The *Align 1* region depicted in Fig. 2 is an optimization of the *Preshift and Align1* region in Fig. 1. The reader can verify that the implementation of the *Align 1* region satisfies the specification of $\text{FSOP}'[-1:53]$ that is summarized in Table 1.
3. The following condition is computed during the computation of the exponent difference

$$\begin{aligned}\text{IS\_R1} &\iff (|ea-eb|\geq 2)\\ &\iff ORtree(\text{IS\_BIG}, \text{MAG\_MED}[5:0],\\ &\qquad \text{AND}(\text{MAG\_MED}[0], \text{NOT}(\text{SIGN\_BIG}))),\end{aligned}$$

which will be used later for the selection of the valid path. Note that, the exponent difference is computed using one's complement representation. This implies that the magnitude is off by one when the exponent difference is positive. In particular, the case of the exponent difference equal to 2 yields a magnitude of 1 and a sign bit of 0. This is why the expression $\text{AND}(\text{MAG\_MED}[0], \text{NOT}(\text{SIGN\_BIG}))$ appears in the OR-tree used to compute the IS_R signal.

5.3.2 Details: R-Path Second Cycle

Fig. 2 depicts a detailed block diagram of the second cycle of the R-path. The details of the implementation are described below.

Our implementation of the R-path in the second cycle consists of two parallel paths called the upper part and the lower part. The *upper part* deals with positions $[-1:52]$ of the significands and the *lower part* deals with positions $[53:116]$ of the significands. The processing of the lower part has to take into account two additional values: the rounding injection, which depends only on the reduced rounding mode, and the missing ULP (2^{-116}) in effective subtraction due to the one's complement representation.

The processing of $\text{FSOPA}[53:116]$, $\text{INJ}[53:116]$ and $\text{S.EFF}\cdot 2^{-116}$ is based on:

$$|\text{TAIL}[52:116]| = \begin{cases} |\text{FSOPA}[53:116]| + |\text{INJ}[53:116]| & \text{if S.EFF} = 0\\ |\text{FSOPA}[53:116]| + |\text{INJ}[53:116]| + 2^{-116} & \text{if S.EFF} = 1.\end{cases}$$

The bits $C[52], R', S'$ are defined by $C[52] = \text{TAIL}[52], R' = \text{TAIL}[53], S' = \text{OR}(\text{TAIL}[54:116])$. They are computed by using a 2-bit injection string that has a different value for effective addition and subtraction.

1. Effective addition. Let S_{add} denote the sticky bit that corresponds to $\text{FSOPA}[54:116]$, then

$$S_{add} = \text{OR}(\text{FSOPA}[54], \ldots, \text{FSOPA}[116]).$$

The injection can be restricted to two bits $\text{INJ}[53:54]$ and we simply perform a 2-bit addition to obtain the three bits $C[52], R', S'$:

$$|(C[52], R', S')| = |\text{INJ}[53:54]| + |(\text{FSOPA}[53], S_{add})|.$$

2. Effective subtraction. In this case, we still have to add to FSOPA the missing 2^{-116} that we neglected to add during the first cycle. Let S_{sub} denote the sticky bit that corresponds to bit positions $[54:116]$ in the binary representation of $|\text{FSOPA}[54:116]| + 2^{-116}$, then

$$S_{sub} = \text{OR}(\text{NOT}(\text{FSOPA}[54:116])).$$

The addition of 2^{-116} can create a carry to position $[53]$, which we denote by $C[53]$. The value of $C[53]$ is one iff $\text{FSOPA}[54:116]$ is all ones, in which case the addition of 2^{-116} creates a carry that ripples to position $[53]$. Therefore, $C[53] = \text{NOT}(S_{sub})$. Again, the injection can be restricted to two bits $\text{INJ}[53:54]$, and we compute $C[52], R', S'$ by

$$|C[52],\ R',\ S'| = |(\text{FSOPA}[53],\ S_{sub})| + |\text{INJ}[53:54]| + 2C[53].$$

Note that the result of this addition cannot be greater than $7\cdot 2^{-54}$, because $C[53] = \text{NOT}(S_{sub})$.

A fast implementation of the computation of $C[52], R', S'$ proceeds as follows. Let $S = S_{add}$ in effective addition, and $S = S_{sub}$ in effective subtraction. Based on $\text{S.EFF}, \text{FSOPA}[53]$ and $\text{INJ}[53:54]$, the signals $C[52], R', S'$ are computed in two paths: one assuming that $S = 1$ and the other assuming that $S = 0$.

Fig. 2 depicts a naive method of computing the sticky bit S to keep the presentation structured rather than obscure it with optimizations. A conditional inversion of the bits of $\text{FSOPA}[54:116]$ is performed by XOR-ing the bits with S.EFF. The possibly inverted bits are then input to an OR-tree. This suggestion is somewhat slow and costly. A better method would be to compute the OR and AND of (most of) the bits of $\text{FS}[54:116]$ during the alignment shift in the first cycle. The advantages of advancing (most of) the sticky bit computation to the first cycle is twofold: 1) There is ample time during the alignment shift whereas the sticky bit should be ready after at most 5 logic levels in the second cycle; and 2) This saves the need to latch all 63 bits (corresponding to $\mathrm{FS}[54:116]$) between the two pipeline stages.

The upper part computes the correctly rounded sum (including post-normalization) and uses for the computation the strings $\mathrm{FLP}[-1:52], \mathrm{FSOPA}[-1:52]$, and $(C[52], R', S')$. The rest of the algorithm is identical to the rounding presented, analyzed, and proven in [8] for FP multiplication.

5.3.3 Details: N-Path

Fig. 3 depicts a detailed block diagram of the N-path. The nonstraightforward boxes are described below.

1. The region called "Path Selection Condition 2" computes the signal IS_R2, which signals whether the magnitude of the significand difference (after preshifting) is greater than or equal to 2. This is one of the clauses needed to determine if the outcome of the R-path should be selected for the final result.
2. The implementation of the *Approximate LZ Count* box deserves some explanations.

   a. The PN-recoding creates a new digit in position $[-2]$. This digit is caused by the negative and positive carries. Note that the P-recoding does not generate a new digit in position $[-3]$.

   b. The PENC boxes refer to priory encoders; they output a binary string that represents the number of leading zeros in the input string.

   c. How is $\mathrm{LZP2}[5:0]$ computed? Let k denote the number of leading zeros in the output of the 55-bitwise XOR. Claim 1 implies that if $flp + fsopa > 0$, then $(flp + fsopa) \cdot 2^k \in [1, 4)$. The reason for this is (using terminology of Claim 1) that the position of the digit σ equals $[k-2]$. We propose to bring σ to position $[-2]$ (recall that an additional multiplication by 4 is used to bring the positive result to the range $[1, 4)$). Hence, a shift by k positions is required and $\mathrm{LZP2}[5:0]$ is derived by computing k.

   d. How is $\mathrm{LZP1}[5:0]$ computed? If $flp + fsopa < 0$, then Claim 1 implies that $(flp + fsopa) \cdot 2^{k-1} \in [1, 4)$. The reason for this is that we propose to bring σ to position $[-1]$ (recall that, an additional multiplication by 2 is used to bring the negative result to the range $[1, 4)$). Hence, a shift by $k-1$ positions is required and $\mathrm{LZP1}[5:0]$ is computed by counting the number of leading zeros in positions $[-1:52]$ of the outcome of the 55-bitwise XOR.

5.3.4 Details: Path Selection

We select between the R-path and the N-path result depending on the signal IS_R. The implementation of this condition is based on the three signals IS_R1, IS_R2, and S.EFF, where $\mathrm{IS\_R1} \Longleftrightarrow (abs(delta) \geq 2)$ is the part of the path selection condition that is computed in the R-path and IS_R2 is the part of the path selection condition that is computed in the N-path. With the definition of IS_R1 we get according to (1):

$$\begin{aligned}\mathrm{IS\_R} &= \overline{\mathrm{S.EFF}} \text{ OR } \mathrm{IS\_R1} \text{ OR } (fpsum \in [2,4)) \\ &= \overline{\mathrm{S.EFF}} \text{ OR } \mathrm{IS\_R1} \text{ OR } \big((fpsum \in [2,4)) \\ &\quad \text{AND S.EFF AND } \overline{\mathrm{IS\_R1}}\big).\end{aligned}$$

We define $\mathrm{IS\_R2} = (fpsum \in [2,4)) \wedge \mathrm{S.EFF} \wedge \overline{\mathrm{IS\_R1}}$, so that $\mathrm{IS\_R} = \overline{\mathrm{S.EFF}}$ OR IS_R1 OR IS_R2. Because the assumptions $\mathrm{S.EFF} = 1$ and $\overline{\mathrm{IS\_R1}}$ are exactly the assumptions that we use during the computation of $fpsum$ in the N-path, the condition IS_R2 is easily implemented in the N-path by the bit at position $[-1]$ of the absolute significand difference. The condition IS_R1 and the signal S.EFF are computed in the R-path. After IS_R is computed from the three components according to the above equation, the valid result is selected either from the R-path or the N-path accordingly.

6 Delay Analysis

In this section, we analyze the delay of our FP-adder algorithm using the *Logical Effort Model* [33]. In the analysis of gate delays, this delay model allows a rigorous treatment of the contributions of fanouts and gate sizing independent of the used technology, process, voltage, or feature size. Based on this delay model, we propose the delay-optimal implementation of our FP addition algorithm by considering optimized gate sizing and driver insertion. For comparisons we scale the delay estimations from this model to units of FO4 inverter delays (inverter delay with a fanout of four). The use of this unit has been a popular choice in literature on circuit design whenever delays are to be determined in technology-independent terms. It has been demonstrated for several examples that delay estimations in FO4 delay units are rather insensitive to choices of technological parameters [12].

Previous delay analysis in logic levels (LLs). Our current delay analysis differs from the delay analysis that we have described in earlier versions of our work [28], [29]. Previously, we analyzed delays in terms of logic levels (XOR delays) considering a basic hardware model that has already been used in [7], [8]. Restrictions and delay contributions of fanouts could not be modeled precisely in this simpler analysis in which gate sizes and gate delays were considered to be fixed. In our current analysis, each gate is assigned an independent size and delay. This allows for precise modeling of fanout contributions and full-custom gate size optimizations for minimum delay.

We previously estimated the total delay of our algorithm to be 24 logic levels between latches, which could be evenly partitioned into two stages with 12 logic levels each. One logic level (XOR delay with a fanout of 1) has a delay of 1.6 FO4 inverter delay units in our current framework (this is because $d(XOR1) = 8$ logical effort delays and $d(FO4) = 5$ logical effort delays), so that the previous analysis estimated the delay of our algorithm to be 38.4 FO4 delays. This is approximately 19 FO4 delays for each of two pipeline stages between latches.

Delay Analysis and Optimization of our Algorithm considering the Logical Effort Model. The Logical Effort Model [33] allows for technology-independent delay analysis of CMOS circuits and enables rigorous delay analysis that takes into account fanouts, drivers, and gate-sizing. In this model each gate χ_i of the circuit implementation is characterized by 4 independent parameters: the logical effort g_i, the branching effort b_i, the electrical effort h_i and the parasitic delay p_i.

The delay of a gate is determined by: $d(\chi_i) = g_i \cdot h_i + p_i$, the delay of a path is the accumulated gate delay on the path and the delay of a circuit is the maximum delay of any path from an input to an output of this circuit.

The logical efforts and the parasitic delays entirely depend on the type and the fanin of the gates. The other two parameters of the electrical effort (representing driving strength and gate size) and the branching effort (representing relative input capacitances at branching points) can be chosen arbitrarily to optimize for the overall delay of the implementation. Although the branching efforts do not occur in the above delay for a single gate, they can influence the delay on the critical path, because they determine relations of electrical efforts in a chain of gates if branching takes place.

For a single path, the model of logical effort from [33] allows very elegantly to determine the optimized gate sizes and the optimized number of driver stages to be added. For complex circuit with irregular and asymmetric branching structures as our FP-adder algorithm, such gate size optimizations become very complex and exceed the scope of what is described in [33]. We have developed a procedure to determine the optimal gate sizes and drivers even for the complex setting of our algorithm. Due to space limitation of the current manuscript, we do not discuss the details of our optimization procedure. The steps are explained in the extended version [30]. The optimal parameter settings for the implementation are summarized in Table 2. The result of the delay analysis of our optimized implementation is that we have two balanced stages with delays of 15.3 F04 delays (R-path, first cycle), 15.3 F04 delays (R-path, second cycle), 15.3 F04 delays (N-path, first cycle), 14.8 F04 delays (N-path, second cycle). The latency of our optimized FP adder implementation is 30.6 F04 delays and the cycle time is 15.3 F04 delays. Note that, although the current delay estimations are more accurate due to the delay model, they are considerably faster than anticipated in our previous work in [28], [29] due to the optimization of gate sizes.

7 Testing of Our Algorithm

The FP addition and subtraction algorithm presented in this paper was tested by Levin [16, chapter 8]. He used the following testing methodology. Two parametric algorithms for FP-addition were designed, each with p bits for the significand string and n bits for the exponent string. One algorithm is the naive algorithm, and the other algorithm is the FP-addition algorithm presented in this paper. Small values of p and n enable exhaustive testing (i.e., input all $2 \cdot 2^{p+n+1}$ binary strings). This exhaustive set of inputs was simulated on both algorithms. Mismatches between the results indicated mistakes in the design. The mismatches were analyzed using assertions specified in the description of the algorithm, and the mistakes were located. Interestingly, most of the mistakes were due to omissions of fill bits in alignment shifts. The algorithm presented in this paper passed this testing procedure without any errors. Bar-Or et al. [1] used the following novel test vector generation methodology. One may distinguish between "control" signals and "datapath" signals in the FP-addition algorithm. For example, the signal selecting between the R-path and the N-path is a control signal, whereas the actual value of the significand is a datapath signal. The goal of Bar-Or et al. was to generate a set of test vectors that generated as many combinations of control vectors as possible. This was achieved by collecting test vectors for a reduced precision FP-adder and translating these test vectors to the double-precision FP-addition algorithm. The translation was successful if the translated input vector created the same combination of control signals. Bar-Or et al. report successful translation of 84 percent of the reduced precision test vector. The algorithm presented in this paper passed all 2,872 test vectors generated in this fashion.

8 Comparison with Other FP-Adder Algorithms

In this section, we overview several FP-adder algorithms described in the literature and compare them with our FP-adder algorithms. These algorithms were published as technical papers [2], [9], [20], [22], [23], [24], [25], [27], [28], or as patents [6], [10], [11], [14], [15], [18], [19], [21], [31], [32], [35]. We summarize the optimization techniques that are used in these algorithms. The entries in this table are ordered according to their year of publication. The last entry in this list corresponds to our proposed FP-adder implementation. This entry considers the optimized gate sizing and driver insertion and lists our delay estimations in FO4 inverter delays with a rigorous, but technology-independent consideration of fanout effects.

Although many of the FP-adders have two paths for the computations, this separation does not always utilize Farmwald's idea of each path being simpler and faster than a unified design [9]. In some cases, the two paths are just used for different rounding cases. In other cases, rounding is not dealt within the two paths at all, but computed in a separate rounding step that is combined for both paths after the sum is normalized. Such implementations can be recognized in Table 3 by the fact that they do not precompute the possible rounding results and only have to consider one binade for rounding. We identified three different path selection conditions among the FP-adder designs that can be considered to be "true" two-paths designs. The conditions are as follows:

- The first group uses the path selection condition from [9] which is only based on the absolute value of the exponent difference. A "far"-path is then selected for $|\delta| > 1$ and a "near"-path is selected for $|\delta| \leq 1$. This path selection condition is used by the implementations from [2], [18], [22], [25], [28]. All of them have to consider four different result binades for rounding. Many binades for rounding require more latency.

TABLE 2
Optimized Parameter Choices and Delays for Our FP Adder Implementation

R-path 1st cycle

path through exp diff

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	CPA	-	-	-	1	0	0
2	CPA	-	-	-	1	0	0
3	CPA	-	-	-	1	16.38	16.38
4	XOR	4	2	0.93	3.73	4	24.11
5	INV	1	1	3.73	3.73	1	28.84
6	INV	1	53	3.73	3.73	1	33.57
7	2:1 MUX	2	1	1.69	3.38	4	40.95
8	4:1 MUX	2	1	1.69	3.38	8	52.33
9	4:1 MUX	2	1	1.69	3.38	8	63.71
10	4:1 MUX	2	1	1.69	3.38	8	75.09

path through complementation

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	3-XOR	12	1	0.28	3.38	4	7.38
2	INV	1	1.11	3.32	3.32	1	11.7
3	INV	1	1	3.32	3.32	1	16.02
4	INV	1	1	3.32	3.32	1	20.33
5	XOR	4	106	0.83	3.32	4	27.65
6	2:MUX	2	1	1.69	3.38	4	35.03
7	2:1 MUX	2	1	1.69	3.38	4	42.41
8	4:1 MUX	2	1	1.69	3.38	8	53.79
9	4:1 MUX	2	1	1.69	3.38	8	65.17
10	4:1 MUX	2	1	1.69	3.38	8	76.55

path through mux selection

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	3-XOR	12	1	0.28	3.38	4	7.38
2	INV	1	10.04	4.07	4.07	1	12.45
3	INV	1	1	4.07	4.07	1	17.52
4	INV	1	1	4.07	4.07	1	22.58
5	INV	1	106	4.07	4.07	1	27.65
6	2:1 MUX	2	1	1.69	3.38	4	35.03
7	2:1 MUX	2	1	1.69	3.38	4	42.41
8	4:1 MUX	2	1	1.69	3.38	8	53.79
9	4:1 MUX	2	1	1.69	3.38	8	65.17
10	4:1 MUX	2	1	1.69	3.38	8	76.55

R-path 2nd cycle

path through rnd decision

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	XOR	4	1	1	1	4	8
2	CPA	-	-	-	-	35	43
3	2:1 MUX	2	2	1.93	3.85	4	50.85
4	INV	1	1	3.85	3.85	1	55.7
5	INV	1	53	3.85	3.85	1	60.56
6	2:1 MUX	2	1	1.93	3.85	4	68.41
7	2:1 MUX	2	1	1.93	3.85	4	76.26

path with shifting of sum

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	XOR	4	1	1	1	4	8
2	CPA	-	-	-	-	35	43
3	INV	1	2	3.85	3.85	1	47.85
4	INV	1	1	3.85	3.85	1	52.7
5	2:1 MUX	2	53	1.93	3.85	4	60.56
6	2:1 MUX	2	1	1.93	3.85	4	68.41
7	2:1 MUX	2	1	1.93	3.85	4	76.26

path with shifting of incremented sum

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	XOR	4	1	1	4	4	8
2	CPA	-	-	-	-	35	43
3	NOR	1.67	1	2.17	3.62	2	48.62
4	INV	1	1	3.62	3.62	1	53.25
5	2:1 MUX	2	53	1.81	3.62	4	60.87
6	2:1 MUX	2	1	1.93	3.85	4	68.73
7	2:1 MUX	2	1	1.93	3.85	4	76.58

N-path 1st cycle

critical path through N-path 1st cycle

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	XOR	4	53	0.97	3.88	4	7.88
2	2:1 MUX	2	1	1.94	3.88	4	15.75
3	2:1 MUX	2	1	1.94	3.88	4	23.63
4	INV	1	1	3.88	3.88	1	28.50
5	INV	1	1	3.88	3.88	1	33.38
6	CPA		1	1		35	68.38
7	XOR	4	1	0.97	3.88	4	76.25

N-path 2nd cycle

critical path through N-path 2nd cycle

stage	type of gate	logical effort	branching effort	electrical effort	stage effort	parasitic delay	accumulated delay
1	INV	1	1	3.35	3.35	1	4.35
2	INV	1	1	3.35	3.35	1	8.70
3	INV	1	1	3.35	3.35	1	13.05
4	INV	1	53	3.35	3.35	1	17.40
5	2:1 MUX	2	1	1.68	3.35	4	24.75
6	4:1 MUX	2	1	1.68	3.35	8	36.10
7	4:1 MUX	2	1	1.68	3.35	8	47.45
8	4:1 MUX	2	53	1.68	3.35	8	58.80
9	2:1 MUX	2	1	1.68	3.35	4	66.15
10	2:1 MUX	2	1	1.93	3.85	4	74.00

- A second version of the path selection condition is used by [21]. In this case, the far path is additionally used for all effective additions. This allows to unconditionally negate the smaller operand in the "near"-path. Also this implementation has to consider four different result binades for rounding.
- In the implementations of [11], [17] a third version of the path selection condition is used. In this case, the "far"-path computes the result also in the cases where a normalization shift by at most one position is needed (this shift is a bidirectional shift). In this way, the "near"-path is simplified because no rounding takes place in the "near"-path. Still there are three different result binades to be considered for rounding and normalization in the "far"-path in the designs of [11], [17].

Our path selection condition is different from these three and was developed independently from the later two. Its advantages are described in detail in Section 4.1. Observe that, due to our path selection condition, the "near"-path in our FP-adder requires only subtraction and does not require rounding at all. In addition, we were also able to reduce to 2 the number of binades that have to be considered for rounding and normalization in the "far" path. As shown in Section 5, there is a very simple implementation for the path

TABLE 3
Overview of Optimization Techniques Used by Different FP-Adder Algorithms

Algorithm	two parallel computation paths	# CP adders for significands	only subtraction in one of the two paths	no rounding required in one path	pre-computation of rounding results	one's complement significand negation	parallel approx lead0/1 count	modified adder including round decision	injection-based rounding reduction	unification of rounding cases for add/sub	pre-computation of post-normalization	one's complement exponent difference	split of δ in upper and lower half	two alignment shifters for $\delta \leq 0$ & $\delta < 0$	# binades to consider for rounding	latency (LL) for double precision	latency (FO4 delays) for double precision
naive design(sec3)	-	1	-	-	-	-	-	-	-	-	-	-	-	-	1	>42	
Farmwald'87 [10]	-	1	-	-	-	-	-	-	-	-	-	-	-	-	1		
INTEL'91 [31]	-	2	-	-	X	X	X	-	-	-	-	-	-	-	3		
Toshiba'91 [14]	-	2	-	-	X	X	-	-	-	-	-	-	-	-	3		
Stanford Rep'91 [25]	X	1	-	-	-	-	-	-	-	-	-	-	-	-	3		
Weitek'92 [19]	X	2	-	-	-	X	-	-	-	-	-	-	-	-	1		
NEC'93 [18]	X	3	-	-	X	-	X	-	-	-	-	-	-	-	4		
Park etal'96 [23]	-	1	-	-	X	X	-	-	-	-	-	-	-	-	3		
Hitachi'97 [35]	-	1	-	-	-	X	X	-	-	-	-	-	-	-	1		
SNAP'97 [22]	X	2	-	-	X	X	X	-	-	-	-	-	-	-	4	>28	
Seidel/Even'98 [28]	X	2	-	-	X	X	X	-	X	X	X	X	X	-	3	24	
AMD'98 [32]	X	4	-	-	-	-	X	-	-	-	-	-	-	-	1		
IBM'98 [6]	X	2	-	-	-	X	-	-	-	-	-	-	-	-	1		
SUN'98 [11]	X	2	X	X	X	X	X	X	-	-	-	-	-	-	3	28	
NEC'99 [15]	-	1	-	-	-	-	X	-	-	-	-	-	-	-	1		
Adelaide'99 [2]	X	2	-	-	X	X	X	-	-	-	-	-	-	-	4	>28	
AMD'00 [21]	X	2	X	-	X	X	X	-	-	-	-	-	-	X	4	26	35.2
Our Algorithm $(with\ opt.\ gate\ sizing)$	X	2	X	X	X	X	X	-	X	X	X	-	-	X	2	30.6/1.6 = 19.1	30.6

selection condition in our design that only requires very few gates to be added in the R-path.

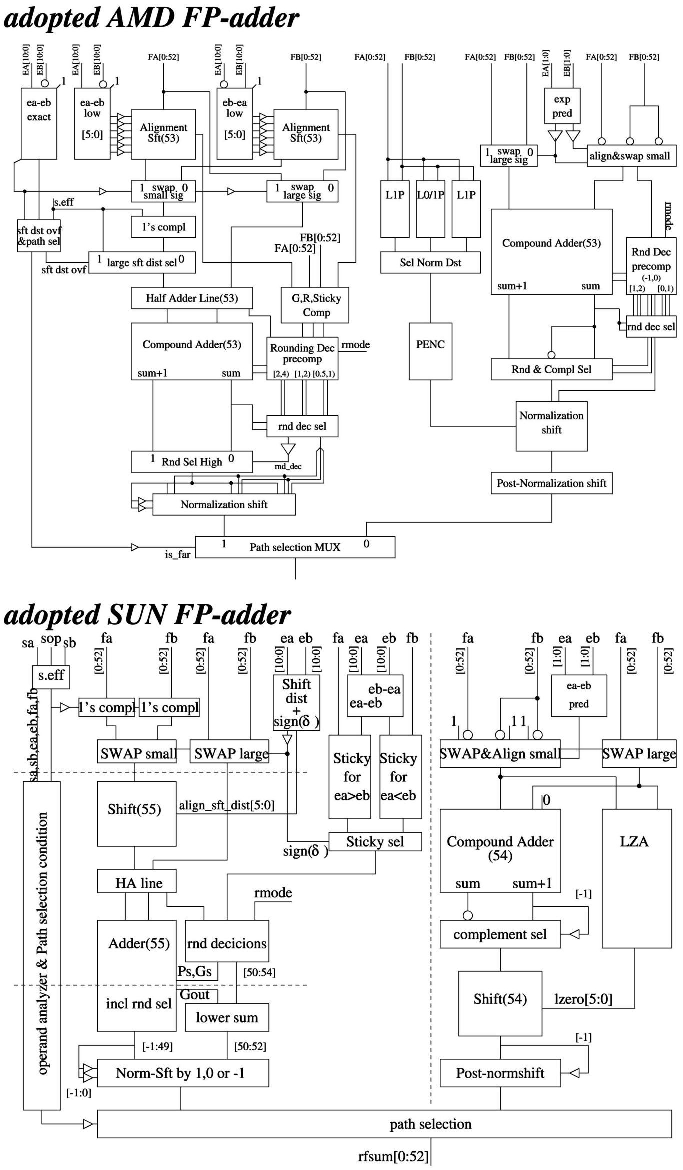
Other optimization techniques most commonly used in previous designs are: The use of one's complement negation for the significand, the parallel precomputation of all possible rounding results in an upper and a lower part, and the parallel approximate leading zero count for an early preparation of the normalization shift. We point out that leading zero approximation has attracted a lot of attention in the literature. Although slightly slower than the leading zero approximation in [26], the leading zero approximation used in our design is proven by a very elegant proof that is based on bounds of fraction ranges after recoding. A faster leading zeros approximation would not effect the latency of our design.

The following two FP-adder designs are described in detail: 1) An implementation based on the 2000 patent [21] from AMD and 2) An implementation based on the 1998 patent [11] from SUN. We chose to focus on these two FP-adders because the union of the optimization techniques used by these two implementation includes the main optimization techniques appearing in literature.

In the recent work from [4], an optimization technique has been described that deals with the combination of the sequence of a half-adder line with a carry-propagate adder and optimizes its delay by using the reverse-carry approach. Because the sequence of a half-adder and a carry propagate adder appears in almost any algorithm for FP addition, this optimization technique would be applicable to almost any FP adder algorithm and be orthogonal to most other optimization techniques. Because it deals with the internal structure of a compound adder implementation rather than the reorganization of available blocks in an FP addition algorithm, we do not discuss this technique in further detail. Several other optimization techniques appear in the literature. Techniques for reducing hardware cost by sharing hardware between the paths appear, for example, in [24], [31]. Another example is the pipelined-packet forwarding paradigm, designed to speed FP-multiplication, which was shown to be feasible also for FP-addition in [20]. We do not elaborate on these techniques since our goal was the design of fast IEEE FP-adders with nonredundant representation of the result.

8.1 FP-Adder Implementation Corresponding to the AMD Patent [21]

The patent from [21] describes an implementation of an FP-adder for single precision operands that only considers the rounding mode round-to-nearest-up. To be able to compare this design with our implementation, we had to extend it to double precision and we had to add hardware for the implementation of the four IEEE rounding modes. The main changes that were required for the IEEE rounding implementation was the "large shift distance selection"-mux in the "far"-path to be able to deal also with exponent differences $|\delta| > 63$. Then, the half adder line in the far path before the compound adder had to be added to be able also to pre-compute all possible rounding results for rounding mode round-to-infinity. Moreover, some additional logic had to be used for a L-bit fix in the case of a tie in rounding


adopted AMD FP-adder

adopted SUN FP-adder

Fig. 4. Block diagrams of the AMD FP-adder implementation according to [21] adopted to accept double precision operands and to implement all four IEEE rounding modes and Block diagram of the SUN FP-adder implementation according to [11] adopted to work only on unpacked normalized double precision operands and to implement all four IEEE rounding modes.

mode round-to-nearest in order to implement the IEEE rounding mode RNE instead of RNU. Fig. 4 shows a block diagram of the adopted FP-adder implementation based on [21]. One main optimization technique in this design is the use of two parallel alignment shifters in the beginning of the "far"-path. This technique makes it possible to begin with the alignment shifts very early, so that the first part of the "far"-path is accelerated. This makes the split into two stages unbalanced for double precision implementation (as estimated with our previous delay analysis of the implementation without optimized gate sizes and also with our current delay analysis that considers an implementation with optimized gate sizes). Our previous analysis implies an effective delay of two clock periods each corresponding to fourteen to fifteen logic levels between latches although the first stage only requires twelve logic levels. To reduce the difference in the delay of the two stages of the "far"-path and reduce the cycle time for the adopted AMD implementation, the technique from [4] might be helpful, as it can reduce the delay of the half-adder/compound adder sequence in the second stage of the design.

Apart from our FP adder implementation, the adopted AMD algorithm is the fastest among all considered FP adder algorithms in the logic level analysis. To allow a better comparison, we also optimized its implementation by optimized gate sizing and driver insertion and determined its delay in terms of FO4 inverter delays with logical effort. Also in this analysis the "far"-path lies on the critical path. The first stage of the "far"-path of the optimized implementation has a delay of 15.5 FO4 inverter delays, which is slightly slower than our first stage. The second stage has been determined to have an optimized delay of 19.7 FO4 inverter delays. This shows that even after gate size optimization and optimized driver insertion the design is quite unbalanced between the stages. The delays from the two stages add up to a total latency of this algorithm of 35.2 FO4 inverter delays. In comparison to this our optimized implementation improves the latency by 13 percent. Because the cycle time is determined by the slowest stage, the optimized AMD implementation would require a cycle time of 19.7 FO4 inverter delays between pipeline stages. In comparison to this, our cycle time of 15.3 FO4 delays is an improvement of 22 percent.

8.2 FP-Adder Implementation Corresponding to the SUN Patent [11]

The patent from [11] describes an implementation of an FP-adder for double precision operands considering all four IEEE rounding modes. This implementation also considers the unpacking of the operands, denormalized numbers, special values and overflows. The implementation targets a partitioning into three pipeline stages. For the comparison with our implementation and the adopted AMD implementation, we reduce the functionality of this implementation also to consider just normalized double precision operands like in the other two implementations.

The FP-adder implementation corresponding to this SUN patent uses a special path selection condition that simplifies the "near"-path. The "near"-path deals only with effective subtractions and no rounding is required. Therefore, the implementation of the "near"-path in the SUN implementation and the N-path in our FP-adder are very similar. The main differences have to do with approximate leading zero counting and the possible ranges of the significand sum that have to be considered. We employ unconditional pre-shifts for the significands in the N-path that do not require any additional delay (they are mainly used to stick to the convention of the R-path).

The main contribution of the SUN patent in the "far"-path is to integrate the computation of the rounding decision and the rounding selection into a special carry propagate (CP) adder implementation. On the one hand, this simplifies the partitioning of this design into three pipeline stages as suggested in the patent, because this modified CP adder design can be easily cut in the "middle." The implementation of the path-selection condition seems to be more complicated than in other designs and is depicted in [11] by two large boxes used to analyze the operands in both paths. Fig. 4 depicts a block diagram of this adopted design. Because this implementation was estimated to be slower than the adopted AMD implementation in our previous analysis, we did not consider gate-size optimizations for this implementation.

We conclude that, in comparison to our gate-optimized FP-adder implementation, the FP-adder implementations corresponding to the AMD and SUN patents both seem to be considerably slower. Additionally, they have a more complicated IEEE rounding implementation and are not amenable to balanced partitioning into two pipeline stages.

Acknowledgments

An extended abstract of this work titled "On the design of fast IEEE Floating-Point Adders" appeared in [29], US patent pending.

References

[1] S. Bar-Or, G. Even, and Y. Levin, "Generation of Representative Input Vectors for Parametric Designs: from Low Precision to High Precision," *Integration, the VLSI J.*, vol. 36, issues 1-2, pp. 69-82, Sept. 2003.

[2] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim, "Reduced Latency IEEE Floating-Point Standard Adder Architectures," *Proc. 14th IEEE Symp. Computer Arithmetic,* pp. 35-43, 1999.

[3] R. Brent and H. Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Computers,* vol. 31, no. 3, pp. 260-264, Mar. 1982.

[4] J. Bruguera and T. Lang, "Using the Reverse-Carry Approach for Double-Datapath Floating-Point Addition," *Proc. 15th IEEE Int'l Symp. Computer Arithmetic (Arith15),* pp. 203-210, June 2001.

[5] M. Daumas and D. Matula, "Recoders for Partial Compression and Rounding," Technical Report RR97-01, Ecole Normale Superieure de Lyon, LIP 1996.

[6] L. Eisen, T. Elliott, R. Golla, and C. Olson, "Method and System for Performing a High Speed Floating Point Add Operation." IBM Corporation, US patent 5790445, 1998.

[7] G. Even, S. Müller, and P. Seidel, "Dual Precision IEEE Floating-Point Multiplier," *INTEGRATION The VLSI J.,* vol. 29, no. 2, pp. 167-180, Sept. 2000.

[8] G. Even and P.-M. Seidel, "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 638-650, July 2000.

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

[10] P. Farmwald, "Bifurcated Method and Apparatus for Floating-Point Addition with Decreased Latency Time," US patent 4639887, 1987.

[11] V. Gorshtein, A. Grushin, and S. Shevtsov, "Floating Point Addition Methods and Apparatus." Sun Microsystems, US patent 5808926, 1998.

[12] M. Horowitz, "VLSI Scaling for Architects," http://velox.stanford.edu/papers/VLSIScaling.pdf, 2000.

[13] "IEEE Standard for Binary Floating Point Arithmetic,"ANSI/IEEE754-1985, 1985.

[14] T. Ishikawa, "Method for Adding/Subtracting Floating-Point Representation Data and Apparatus for the Same," Toshiba, K.K., US patent 5063530, 1991.

[15] T. Kawaguchi, "Floating Point Addition and Subtraction Arithmetic Circuit Performing Preprocessing of Addition or Subtraction Operation Rapidly," NEC, US patent 5931896, 1999.

[16] Y. Levin, "Supporting Denormalized Numbers in an IEEE Compliant Floating-Point Adder Optimized for Speed," http://hyde.eng.tau.ac.il/Projects/FPADD/index.html, 2001.

[17] A. Naini, A. Dhablania, W. James, and D. Das Sarma, "1-GHz HAL SPARC64 Dual Floating Point Unit with RAS Features," *Proc. 15th IEEE Int'l Symp. Computer Arithmetic (Arith15),* pp. 173-183, June 2001.

[18] T. Nakayama, "Hardware Arrangement for Floating-Point Addition and Subtraction," NEC, US patent 5197023, 1993.

[19] K. Ng, "Floating-Point ALU with Parallel Paths," US patent 5136536, Weitek Corp., 1992.

[20] A. Nielsen, D. Matula, C.-N. Lyu, and G. Even, "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.

[21] S. Oberman, "Floating-Point Arithmetic Unit Including an Efficient Close Data Path," AMD, US patent 6094668, 2000.

[22] S. Oberman, H. Al-Twaijry, and M. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units," *Proc. 13th IEEE Symp. Computer Arithmetic,* pp. 156-165, 1997.

[23] W.-C. Park, T.-D. Han, S.-D. Kim, and S.-B. Yang, "Floating Point Adder/Subtractor Performing IEEE Rounding and Addition/Subtraction in Parallel," *IEICE Trans. Information and Systems,* vol. 4, pp. 297-305, 1996.

[24] N. Quach and M. Flynn, "Design and Implementation of the SNAP Floating-Point Adder," Technical Report CSL-TR-91-501, Stanford Univ., Dec. 1991.

[25] N. Quach, N. Takagi, and M. Flynn, "On fast IEEE Rounding," Technical Report CSL-TR-91-459, Stanford Univ., Jan. 1991.

[26] M. Schmookler and K. Nowka, "Leading Zero Anticipation and Detection—A Comparison of Methods, *Proc. 15th IEEE Symp. Computer Arithmetic,* pp. 7-12, 2001.

[27] P.-M. Seidel, "On The Design of IEEE Compliant Floating-Point Units and Their Quantitative Analysis," PhD thesis, Univ. of Saarland, Germany, Dec. 1999.

[28] P.-M. Seidel and G. Even, "How Many Logic Levels Does Floating-Point Addition Require?" *Proc. 1998 Int'l Conf. Computer Design (ICCD '98): VLSI, in Computers & Processors,* pp. 142-149, Oct. 1998.

[29] P.-M. Seidel and G. Even, "On the Design of Fast IEEE Floating-Point Adders," *Proc. 15th IEEE Int'l Symp. Computer Arithmetic (Arith15),* pp. 184-194, June 2001.

[30] P.-M. Seidel and G. Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition," http://www.engr.smu.edu/seidel/research/fpadd/, 2002.

[31] H. Sit, D. Galbi, and A. Chan, "Circuit for Adding/Subtracting Two Floating-Point Operands," Intel, US patent 5027308, 1991.

[32] D. Stiles, "Method and Apparatus for Performing Floating-Point Addition," AMD, US patent 5764556, 1998.

[33] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits.* Morgan Kaufmann 1999.

[34] A. Tyagi, "A Reduced-Area Scheme for Carry-Select Adders," *IEEE Trans. Computers,* vol. 42, no. 10, Oct. 1993.

[35] H. Yamada, F. Murabayashi, T. Yamauchi, T. Hotta, H. Sawamoto, T. Nishiyama, Y. Kiyoshige, and N. Ido, "Floating-Point Addition/Subtraction Processing Apparatus and Method Thereof," Hitachi, US patent 5684729, 1997.

Peter-Michael Seidel received the BSc degree in electrical engineering, and the BSc and MSc degrees in computer science from the University of Hagen and GMD-German Research Center for Information Technology in 1994 and 1996, respectively, the Dr.-Ing and Habilitation degrees in computer science from the University of the Saarland at Saarbruecken in 2000 and 2002. Since the Spring of 2000, he has been a tenure-track faculty member in the Computer Science and Engineering Department at Southern Methodist University, Dallas. Current areas of research include: formal verification and optimization for high-performance and low-power in computer arithmetic, computer architecture, and VLSI design.

Guy Even received the BSc degree in mathematics and computer science from the Hebrew University in Jerusalem in 1988, the MSc and DSc degrees in computer science from the Technion in Haifa in 1991 and 1994, respectively. During 1995-1997, he was a postdoctoral fellow for the chair of Professor Wolfgang Paul at the University of the Saarland at Saarbruecken. Since 1997, he has been a faculty member in the Electrical Engineering-Systems Department at Tel-Aviv University. Current areas of research include: computer arithmetic and the design of IEEE compliant floating-point units, approximation algorithms for NP complete problems related to VLSI design, and the design of systolic arrays.