High-Radix Implementation of IEEE Floating-Point Addition

Peter-Michael Seidel
Computer Science & Engineering Department
Southern Methodist University
Dallas, TX, 75275
seidel@engr.smu.edu

Abstract

We are proposing a micro-architecture for high-performance IEEE floating-point addition that is based on a (non-redundant) high-radix representation of the floating-point operands. The main improvement of the proposed IEEE FP addition implementation is achieved by avoiding the computation of full alignment and normalization shifts which impose major delays in conventional implementations of IEEE FP addition. This reduction is achieved at the cost of wider operand interfaces and an increased complexity for IEEE compliant rounding. We present a detailed discussion of an IEEE FP adder implementation using the proposed high-radix format and explain the specific benefits and challenges of the design.

1 Introduction

Floating-point addition and subtraction are the most frequent floating-point operations. Both operations use a floating-point (FP) adder. Therefore, latency and throughput of FP adders are important for high-performance FP support and lot of effort has been spent on improving the performance of FP adders (see [6] and the references that appear there). Design optimizations have been mostly based on parallelizing, reordering and simplifying computation steps from previous implementations (e.g. the dual path-optimization from [2]). In [6] a broad overview of various implementations and their optimizations is given. For improved throughput, also the use of alternative (redundant) representations for operands and results has been proposed. This includes the forwarding of partial and/or redundant representations between dependent operations for improved throughput (e.g. [1, 4]). The main intention of the use of redundant representations in these implementations seems to be the reduction or hiding of most of the latency of the actual compression of the significand sum by redundant additions and the related computations for IEEE rounding. But a delay analysis of the architectures reveals that the compression of the significand sum and IEEE rounding does not necessarily impose the major delay in IEEE FP addition implementations. When also counting all partial compressions that are necessary during FP addition on redundant operand representations in addition to the actual significand addition, the overall significand compression delay is not even significantly reduced in these designs.

Our strategy to reduce the latency of IEEE FP addition is different. We identify that in conventional implementations a major fraction of the delay is needed for the various shifts (alignment shift and normalization shift) and their preparation. It is our goal to significantly reduce the delay needed for these shifts in the implementation. Our approach to reduce these shift delays is based on the introduction of a wider, but non-redundant representation of IEEE floating-point numbers using a very high radix. The proposed high-radix FP representations allow to represent exactly the same values as binary IEEE FP representations and allow to reduce the amount of shifts that are necessary to obtain a 'normalized' representation. This is achieved by introducing two penalties: the operand representation becomes wider and the implementation of IEEE rounding becomes more complex.

We are demonstrating the possibilities for optimization of floating-point addition regarding the new format. We present a detailed discussion of an IEEE FP adder implementation using the proposed high-radix format and explain the specific benefits and challenges of the design.

In Section 2, notation is presented and normalized high-radix FP representations are introduced. In Section 3, the steps of conventional IEEE FP addition based on normalized binary operands are reviewed. In Section 4, the main strategies of the proposed FP adder implementation based on normalized high-radix FP representations are explained. We finally conclude in Section 5.

2 Notation

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).

Normalized Binary IEEE FP numbers [3]. As operands we consider the values of 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 $\mathrm{F}[0:53] \in \{0,1\}^{53}$. The values of exponent and significand are defined by:

$$e = \sum_{i=0}^{10} \mathrm{E}[i] \cdot 2^i - 1023, \qquad f = \sum_{i=0}^{52} \mathrm{F}[i] \cdot 2^{-i}.$$

Since we only consider normalized IEEE FP numbers, we have $\mathrm{F}[0] = 1$ and $f \in [1,2)$. A FP number $(\mathrm{S}, \mathrm{E}[10:0], \mathrm{F}[0:52])$ represents the value:

$$val(\mathrm{S}, \mathrm{E}, \mathrm{F}) \;=\; (-1)^{\mathrm{S}} \cdot 2^e \cdot f.$$

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. The advantage of using factorings is the ability to ignore representation details and focus on values.

The inputs of binary 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. an operation $\mathrm{SOP} \in \{0,1\}$ where $\mathrm{SOP} = 0$ denotes an addition and $\mathrm{SOP} = 1$ denotes a subtraction;
3. IEEE rounding mode.

The output is a FP number $(\mathrm{S}, \mathrm{E}[10:0], \mathrm{F}[0:52])$ that represents the IEEE rounded value of

$$\begin{aligned} fpsum \;=\; & val(\mathrm{SA}, \mathrm{EA}[10:0], \mathrm{FA}[0:52]) \\ & +(-1)^{\mathrm{SOP}} val(\mathrm{SB}, \mathrm{EB}[10:0], \mathrm{FB}[0:52]). \end{aligned}$$

Normalized High-Radix FP representations. We introduce an alternative choice for the representation of IEEE values in input operands and results. We target our discussion towards an implementation for double precision FP operands in the following, but the description could be easily adopted to other precisions. We consider high radix (HR) FP number representations radix 2^{53} in which we require the exponent of the representation to be a multiple of 53 (the width of the binary significand representation). To allow for the representation of any IEEE value, the normalized significand radix 2^{53} has a value in the range $[1, 2^{53})$ and is represented with 106 bits (53 to the left, and 53 to the right of the binary point, only 53 out of which are actually used for a given value). Note, that normalized HR representations of IEEE FP values are non-redundant.

Definition 1 *A floating-point factoring (s, e, f) is said to be normalized radix 2^{53}, iff the exponent e is an integral multiple of 53, and the significand f is in the range $[1, 2^{53})$. Note, that a conventional normalized binary floating-point factoring is normalized radix 2.*

To represent exactly the IEEE FP values, the proposed normalized HR representation has additional requirements:

- Out of the 106-bit representation of the significant at most 53 are allowed to be non-zero,
- the most significant non-zero bit needs to be located to the left of the binary point;
- the least significant bit of the significand follows the most significant bit of the significand by 53 bits.

The embedding of the binary significand as 53 significant bits out of 106 bits of the HR significand is illustrated in figure 1.

Definition 2 *A HR floating-point representation is given by the tuple $(\mathrm{SHR}, \mathrm{EHR}[5:0], \mathrm{FHR}^h[52:0].\mathrm{FHR}^l[52:0])$ with sign bit $\mathrm{SHR} \in \{0,1\}$, exponent string $\mathrm{EHR}[5:0] \in \{0,1\}^6$ and significand string $\mathrm{FHR} = \mathrm{FHR}^h[52:0].\mathrm{FHR}^l[52:0] \in \{0,1\}^{106}$, where $\mathrm{FHR}^h[52:0]$ represents the integer part and $\mathrm{FHR}^l[52:0]$ represents the fractional part of the significand string. The values of exponent and significand are defined by:*

$$\begin{aligned} ehr \;&=\; -32 \cdot \mathrm{EHR}[5] + \sum_{i=0}^{4} \mathrm{EHR}[i] \cdot 2^i, \\ fhr \;&=\; \sum_{i=0}^{52} \mathrm{FHR}^h[i] \cdot 2^i + 2^{-53} \cdot \sum_{i=0}^{52} \mathrm{FHR}^l[i] \cdot 2^i. \end{aligned}$$

A HR FP number $(\mathrm{SHR}, \mathrm{EHR}[5{:}0], \mathrm{FHR}^h[52{:}0].\mathrm{FHR}^l[52{:}0])$ represents the value:

$$hrval(shr, ehr, fhr) \;=\; (-1)^{\mathrm{SHR}} \cdot 2^{53 \cdot ehr} \cdot fhr.$$

The most significant significand bit (MSSB) position is defined as the leading non-zero bit position in the significand of a HR floating-point representation. Such a position must exist and must be uniquely defined for the representation of non-zero values. For normalized HR significands the MSSB position has to be in the integer part of the significand $\mathrm{FHR}^h[52:0]$ and therefore it has a value in the range $[52:0]$ The least significant significand bit (LSSB) position is defined as the least significant bit position in the significand of a HR floating-point representation. Because this is the position where a possible rounding increment needs to be applied, this position is also referred to as the rounding position. For IEEE FP values the LSSB has to follow the MSSB by 53 bit positions. Therefore, if the MSSB position is 52, the LSSB is given by $\mathrm{FHR}^h[0]$. In all other cases the LSSB resides in the fractional part of the significand

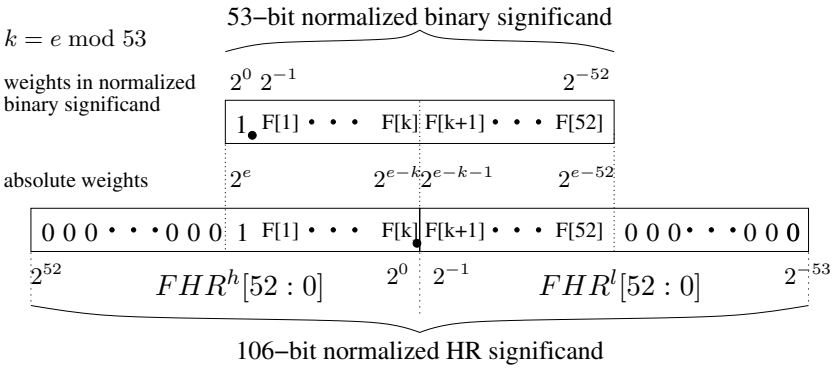

Figure 1. Embedding of normalized binary significands in normalized HR significands.

string. If the MSSB position is given by $Mpos$, then the LSSB position can be computed by $((Mpos + 1) \bmod 53)$. Note, that for IEEE FP values the last fractional bit $\text{FHR}^l[0]$ is never used in a normalized HR representation. This is why a LSSB position of 0 uniquely refers to bit $\text{FHR}^h[0]$.

Conversion. Each binary normalized factoring (sb, eb, fb) of a IEEE FP value has a unique HR normalized factoring $(shr, ehr, fhr) = (sb, \lfloor eb/53 \rfloor, f \cdot 2^{(e \bmod 53)})$ with $val(sb, eb, fb) = hrval(shr, ehr, fhr)$.
The factoring (shr, ehr, fhr) is normalized radix 2^{53}, because the exponent is considered as a factor $2^{53 \cdot ehr}$ in hrval (including factor 53) and $fhr = fb \cdot 2^{(eb \bmod 53)} \in [1, 2^{53})$.

Each normalized HR factoring (shr, ehr, fhr) of a IEEE FP value can be easily converted to the corresponding binary normalized factoring $(sb, eb, fb) = (shr, 53 \cdot ehr + \lfloor log_2(fhr) \rfloor, fhr \cdot 2^{(-\lfloor log_2(fhr) \rfloor)})$, because

$$hrval(shr, ehr, fhr) = val(shr, 53 \cdot ehr + \lfloor log_2(fhr) \rfloor, fhr \cdot 2^{(-\lfloor log_2(fhr) \rfloor)})$$

and because $fb = fhr \cdot 2^{(-\lfloor log_2(fhr) \rfloor)}$ is a normalized binary significand in the range $[1, 2)$.

Exponent Range, Overflow and Underflow Detection The full range of values that is representable with normalized HR FP representations is larger than the range of IEEE FP values when using the full exponent range available. To limit the HR representation range to that of IEEE FP values we consider the normalized HR representation of the largest and smallest IEEE FP values. The maximum and minimum normalized IEEE FP values (magnitudes) are:

$$\begin{aligned} x_{max} &= (2 - 2^{-52}) \cdot 2^{1023} \\ x_{min} &= 2^{-1022}. \end{aligned}$$

The corresponding normalized HR factorings are:

$$\begin{aligned} x_{max} &= hrval(0, 19, 2^{17} - 2^{-36}) \\ x_{min} &= hrval(0, -20, 2^{38}). \end{aligned}$$

The normalized HR representation of x_{max} has a MSSB position of 16 and a LSSB position of 17. The normalized HR representation of x_{min} has a MSSB position of 38 and a LSSB position of 39. Therefore, an overflow in a normalized HR representation (shr, ehr, fhr) can be detected by the condition that either $ehr > 19$ or $ehr = 19$ and the MSSB position is strictly larger than 16. An underflow can be detected by the condition that either $ehr < -20$ or $ehr = -20$ and the MSSB position is strictly smaller than 38.

Rounding Position For IEEE rounding the value of the result needs to be chosen between the two closest representable IEEE values (called rounding candidates $rc1$ and $rc2$ here) that sandwich the exact result as $rc1 \leq x < rc2$. The difference between these two values is by a unit in the last place. For normalized binary FP representations this unit in the last place of the significands has a fixed weight of 2^{-52}. In normalized HR representations the absolute weight of this difference is identical to the normalized binary case, but the position of the corresponding bit within the significand varies. In normalized HR representations this difference is a unit in the LSSB position of the significand. We have discussed the ranges and dependencies between the MSSB and the LSSB position above with the definition of HR number representations.

We can express the position and weight of the rounding increment here based on a different measure that can be easily read off from a normalized HR significand: the number of leading zeros in the integer part of the normalized HR significand representation. Let $Mpos$ represent the position of the MSSB, and $runit$ the unit of the rounding increment (unit of a bit in the LSSB position) of a normalized HR representation $(\text{SHR}, \text{EHR}[5:0], \text{FHR}^h[52:0], \text{FHR}^l[52:0])$. Then,

$$\begin{aligned} Mpos &= 52 - lzero(fhr^h[52{:}0]) \\ runit &= 2^{-lzero(fhr^h[52:0])}. \end{aligned}$$

This equation will be useful to determine the rounding candidates.

We consider the implementation of IEEE FP addition/subtraction based on using normalized HR FP representations for input operands and results. The inputs of HR FP addition/subtraction are:

1. operands $(\text{SA}, \text{EA}[5:0], \text{FA}^h[52:0].\text{FA}^l[52:0])$ and $(\text{SB}, \text{EB}[5:0], \text{FB}^h[52:0].\text{FB}^l[52:0])$;
2. an operation $\text{SOP} \in \{0,1\}$ where $\text{SOP} = 0$ denotes an addition and $\text{SOP} = 1$ denotes a subtraction;
3. IEEE rounding mode.

The output is $(\text{S}, \text{E}[5:0], \text{F}^h[52:0].\text{F}^l[52:0])$, a normalized HR FP representation that represents the IEEE rounded value of

$$fpsum = hrval(\text{SA}, \text{EA}[5:0], \text{FA}^h[52:0].\text{FA}^l[52:0]) + (-1)^{\text{SOP}} hrval(\text{SB}, \text{EB}[5:0], \text{FB}^h[52:0].\text{FB}^l[52:0]).$$

3 Conventional IEEE FP Addition

In this section we overview the steps of a naive FP addition algorithm on normalized binary IEEE FP operands. To simplify notation and to make the transition to the proposed high-radix FP representations easier, we ignore representation and deal only with the values of the inputs, outputs, and intermediate results. In the discussion of the proposed implementation we will refer to the notation defined for the naive algorithm.

Let (sa, ea, fa) and (sb, eb, fb) denote the factorings of the operands into 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:

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

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

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

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

The sum can be written as

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

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

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

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. limit alignment shift: $\delta\_lim = \min\{\alpha, abs(\delta)\}$, 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 $fsan = (-1)^{\text{S.EFF}} fsa$,
6. significand addition $fsum = fl + fsan$,
7. conversion $abs\_fsum = abs(fsum)$,
8. normalization $n\_fsum = norm(abs\_fsum)$,
9. rounding and post-normalization of n_fsum.

Although the sequential consideration of the above steps is a very slow operation it is a good starting point for the description of an implementation optimized for operands in the proposed normalized HR FP representation.

4 High-Radix FP Addition

The main optimization for the implementation of HR floating-point addition is based on a radical reduction of the efforts necessary for alignment and normalization shifts. This reduction is possible based on the limited possible relative alignments of the significands for significand addition. For the discussion we use the notation from the previous section on normalized binary FP addition. Here, we only apply them to the input operands of (sa, ea, fa) and (sb, eb, fb) given here as normalized HR factorings. For example the exponent difference is defined as $\delta = ea - eb$ where ea and eb denote the exponents of the HR representations (i.e. an increment of an exponent by one corresponds to a 53-bit shift of the corresponding significand.)

4.1 Structure of the Implementation

The aligned significands of the two operands only overlap for the three cases of $\delta \in \{-1, 0, 1\}$ as shown in Figure 2. In the two remaining cases of $\delta > 1$ and $\delta < -1$ the significands do not overlap and the significand of the smaller operand has at most an influence on whether a rounding increment is performed on the larger operand. The property that only exponent differences of $\delta \in \{-1, 0, 1\}$ are of interest reminds of the conditions in the NEAR path of the dual path FP adder implementation [2]. In this implementation it was suggested that the exponent difference is predicted based on two least significant bit positions of the exponents. We will consider only one bit here to distinguish between the cases $\delta = 0$ and $|\delta| = 1$. Based on this we select operand assignments in one level of muxes that realize the minimal alignment that needs to be considered here. The normalization shift is similarly simplified. The significand sum is mostly determined by the significand of

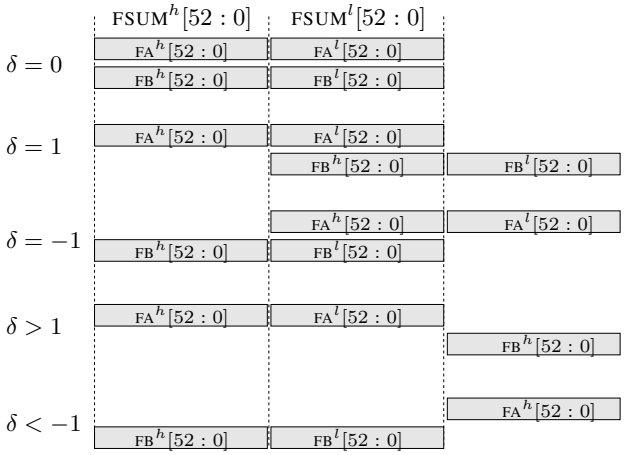

Figure 2. Cases of alignment for normalized HR significands.

the larger operand. Only if in the integer part of the significand sum the result exceeds the value of 2^{53} or becomes 0, the significand sum is not already normalized. In the rare case that normalization is necessary, it only can be by one radix-2^{53} digit to the right (effective additions) or to the left (effective subtractions).

We give an short overview of how the steps of the basic implementation for normalized binary operands are realized in the following:

1. exponent subtraction becomes exponent difference prediction based on one bit. The full exponent difference still needs to be computed to allow for selecting the right path for the final result.

2. operand swapping: in all cases where operand swapping is of importance, both possibilities the difference and the negative of the difference are considered concurrently with later selection of the appropriate one implementing the conversion step.

3. significand alignment: only three cases of the exponent difference are of interest where $\delta \in \{-1, 0, 1\}$. A shift is needed by at most one radix-2^{53} digit. This is realized by one level of muxing.

4. significand negation is implemented based on computing conditional one's complementation for effective subtractions.

5. significand addition: this essential step of the FP addition is combined with the preparation of the rounding candidates. For normalized HR representations, the efficient computation of IEEE rounding is becoming more complicated, because the rounding position is not fixed within the significand, but can vary over the range of 53 different positions.

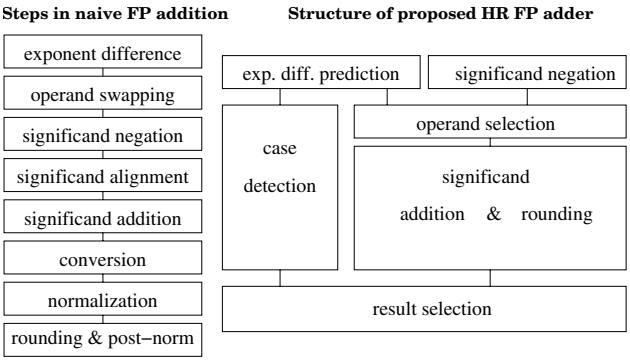

Figure 3. High level structure of the proposed high-radix FP adder implementation.

6. conversion is done by unconditionally computing the significand difference and its negative for effective subtractions for the case that $\delta = 0$. Selection of the positive of the two results then allows to determine the resulting significand in sign-magnitude representation.

7. normalization: there are only two (rare) cases when normalization is an issue. We call these two cases significand sum overflow and underflow indicating the cases that the integer part of the significand sum is either $\geq 2^{53}$ or $= 0$. These are the only two cases when the exponent of the result involves some computation and is not just a selection of the larger of the exponents of the input operands. These cases can be considered in parallel to the other computations.

8. rounding and post-normalization: the rounding computation consists of six parts: (i) the determination of the rounding position (in the form of an approximate representation of the rounding increment and truncation masks), (ii) an approximation of the rounding candidates based on the approximate rounding increments (this is computed integrated with the significand addition), (iii) the determination of the binade of the resulting significand sum and in particular whether it has changed over the larger input operand, (iv) the determination of the rounding decision, (v) the truncation and correction of the pre-computed rounding candidates, (vi) selection of the rounded significand result.

A simplified high-level view of the structure of the proposed HR FP adder implementation is shown in Figure 3.

In the following we explain parts of the implementation in more details:

4.2 Operand Selection

Let the rounding position in the fractional part of the significand sum be denoted by $Lpos$ and the value of the

rounding increment be denoted by $runit = 2^{Lpos-53}$ and let the truncation after bit position pos be denoted as $trunc_{pos}()$. Note that a truncation after bit position $Lpos$ lets the argument become a multiple of the rounding increment $runit$. We denote the rounding decision as $rinc$, where $rinc = 0$ means that the truncated significand sum needs to be considered, and $rinc = 1$ means that the incremented significand sum needs to be considered.

Effective Significand Addition We distinguish between the 5 cases:

1. For $\delta = 0$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fa^l + fb^l) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fa^l + fb^l) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fa^h + fb^h + 1 & \text{if } fsum^l \geq 2^{53} \\ fa^h + fb^h & \text{otherwise} \end{cases}$$

2. For $\delta = 1$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fa^l + fb^h) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fa^l + fb^h) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fa^h + 1 & \text{if } fsum^l \geq 2^{53} \\ fa^h & \text{otherwise} \end{cases}$$

3. For $\delta = -1$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fa^h + fb^l) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fa^h + fb^l) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fb^h + 1 & \text{if } fsum^l \geq 2^{53} \\ fb^h & \text{otherwise} \end{cases}$$

4. For $\delta > 1$ we need to compute:
$$fsum^l = \begin{cases} fa^l & \text{if } rinc = 0 \\ fa^l + runit & \text{if } rinc = 1 \end{cases}$$
$$fsum^h = \begin{cases} fa^h + 1 & \text{if } fsum^l \geq 2^{53} \\ fa^h & \text{otherwise} \end{cases}$$

5. For $\delta < -1$ we need to compute:
$$fsum^l = \begin{cases} fb^l & \text{if } rinc = 0 \\ fb^l + runit & \text{if } rinc = 1 \end{cases}$$
$$fsum^h = \begin{cases} fb^h + 1 & \text{if } fsum^l \geq 2^{53} \\ fb^h & \text{otherwise} \end{cases}$$

Effective Significand Subtraction We distinguish between the 5 cases:

1. The case of $\delta = 0$ for effective subtractions is the only case that requires additional consideration of swapping, because the sign of the difference can not be immediately determined from the exponent difference, but it depends on the values of the operand's significands themselves. We need to distinguish between the cases where the first operand is the larger operand and where the second operand is the larger operand. We concurrently compute a separate addition for each of these two cases with later selection of the non-negative significand sum as the result.

   - To consider the case of $fa \geq fb$ we compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fa^l + \overline{fb^l}) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fa^l + \overline{fb^l}) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fa^h + \overline{fb^h} + 1 & \text{if } fsum^l \geq 2^{53} \\ fa^h + \overline{fb^h} & \text{otherwise} \end{cases}$$

   - To consider the case of $fa < fb$ we compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fb^l + \overline{fa^l}) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fb^l + \overline{fa^l}) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fb^h + \overline{fa^h} + 1 & \text{if } fsum^l \geq 2^{53} \\ fb^h + \overline{fa^h} & \text{otherwise} \end{cases}$$

2. For $\delta = 1$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fa^l + \overline{fb^h}) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fa^l + \overline{fb^h}) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fa^h & \text{if } fsum^l \geq 2^{53} \\ fa^h - 1 & \text{otherwise} \end{cases}$$

3. For $\delta = -1$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(\overline{fa^h} + fb^l) & \text{if } rinc{=}0 \\ trunc_{Lpos}(\overline{fa^h} + fb^l) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fb^h & \text{if } fsum^l \geq 2^{53} \\ fb^h - 1 & \text{otherwise} \end{cases}$$

4. For $\delta > 1$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fa^l + 2^{53}{-}1) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fa^l + 2^{53}{-}1) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fa^h & \text{if } fsum^l \geq 2^{53} \\ fa^h - 1 & \text{otherwise} \end{cases}$$

5. For $\delta < -1$ we need to compute:
$$fsum^l = \begin{cases} trunc_{Lpos}(fb^l + 2^{53}{-}1) & \text{if } rinc{=}0 \\ trunc_{Lpos}(fb^l + 2^{53}{-}1) + runit & \text{if } rinc{=}1 \end{cases}$$
$$fsum^h = \begin{cases} fb^h & \text{if } fsum^l \geq 2^{53} \\ fb^h - 1 & \text{otherwise} \end{cases}$$

In each of the cases above the computations can be implemented by two modified compound adders. Each computing two results: one of them for the integer part of the significand sum and the other one for the fractional part of the significand sum. In Table 1 we list the operand assignments for such two adders for the five cases (effective additions are listed with A& case number, effective subtractions

case	operands for $fsum^h$ ADD1^h	operands for $fsum^l$ ADD1^l	operands for $fsum^h$ ADD2^h	operands for $fsum^l$ ADD2^l
A1	fa^h,fb^h	fa^l,fb^l		
A2	fa^h, 0	fa^l,fb^h		
A3			fb^h,0	fa^h,fb^l
A4	fa^h,0	fa^l,0		
A5			fb^h,0	fb^l,0
S1(a,b)	fa^h, $\overline{fb^h}$	fa^l, $\overline{fb^l}$	fb^h, $\overline{fa^h}$	fb^l, $\overline{fa^l}$
S2	fa^h,-1	fa^l, $\overline{fb^h}$		
S3			fb^h,-1	fb^l, $\overline{fa^h}$
S4	fa^h,-1	fa^l,-1		
S5			fb^h,-1	fb^l,-1

Table 1. Operand assignment of aligned significands.

are listed as S& case number). For case $S1$ of effective subtractions two different results are to be computed concurrently requiring two copies of the same structure (listed in the table as ADD1 and ADD2). The availability of these two copies allows also to join other cases of the computation. In particular the combination of cases 2 and 3 and of cases 4 and 5 makes the implementation more effective. The distinction between cases 1 and 2&3 can be easily decided based on XOR-ing the LSBs of the exponents: case 1 is then predicted by the condition $\text{EA}[0] \oplus \text{EB}[0] = 0$. In general, the implementation based on table 1 allows to vary how many copies of the general hardware structure are realized, so that various cases can be considered concurrently. A joint consideration of different cases by the same hardware saves implementation cost, but can add to the latency of the implementation in the delay needed to differatiate between the cases before the significand additions can be applied. If the distinction can be determined very efficiently, a joint implementation becomes useful. This is in particular the case for the distinction between effective addition and effective subtractions based on the bit S.EFF and the distinction between cases 1 and 2&3 as explained above. These are the cases that should be sharing the same hardware for their implementation. We are also sharing the implementation between the cases 4 and 5. The distinction is based on the sign of the 6-bit exponent difference. Because cases 4 and 5 are otherwise the simplest to compute, the delay overhead to determine the exponent difference sign balances with the faster computation of the significand sum, which only requires to consider increments in these cases.

4.3 Rounding Computation

The rounding computation is split into six parts: (i) the determination of the rounding position (in the form of an approximate representation of the rounding increment and truncation masks), (ii) an approximation of the rounding candidates based on the approximate rounding increments (this is computed integrated with the significand addition), (iii) the determination of the binade of the resulting significand sum and in particular whether it has changed over the larger input operand. (iv) the determination of the rounding decision, (v) the truncation and correction of the precomputed rounding candidates, (vi) selection of the rounded significand result.

The implementation is based on a similar organization as the implementation of variable position rounding for FP multiplication in [5], but the available operands are different and the masks need to be generated differently here.

Assuming that the compressed representation of the significand sum was available as a HR significand representation with an integer part $\text{FSUM}^h[52:0]$ and a fractional part $\text{FSUM}^l[52:0]$. These are illustrated in the top-most two lines of the figure 4. Note, that the MSSB position of $\text{FSUM}^h[52:0]$ needs to hold a logic 1 for normalized representations and that this position is just one position to the right of the LSSB position in the fractional part $\text{FSUM}^l[52{:}0]$ if the two are aligned.

We use a pattern close to $\text{FSUM}^h[52:0]$ to approximate the unit of the rounding increment. We approximate the rounding increment in two senses: (A) The upper part sum shifted by one bit to the left does not only have a one in the LSSB position, but might also have ones in less significant positions. To ensure that these less significant bits do not change the rounding result we need to apply two steps: (i) We need to ensure that there is no carry produced into the LSSB position during the computation of the sum in the fractional part. We implement this by using the inverted string $\overline{\text{FSUM}^h[52{:}0]}$ as a mask on both input operands in the lower half. This ensures that both operands have a 0 in their R-position (position to the right of the LSSB position) and, thus, any carry from less significant positions can not proceed over the R-position. (ii) the significand sum needs to be cleaned from the less significand ones in the representation of the approximate rounding increment. This is implemented by applying an exact truncation mask after the significand sum is generated. The generation of the exact truncation mask can be computed by applying $\text{FSUM}^h[52:0]$ to a Unary-PENC and inverting its outputs (see Figure 4). The generation of this pattern requires some delay, but it is available in time for masking the fractional part of the significand sum at the output of the significand adder. (B) Because $\text{FSUM}^h[52{:}0]$ is not available at the input, we use another approximation: Instead of $\text{FSUM}^h[52{:}0]$ we use the larger of the two significands in the higher part (This can be approximated by simply ORing the two operands). The MSSB determined in this way agrees with the final MSSB except for in the following three cases: (i) In effective addition there can be a carry shifting the leading one of the larger operand to the left by one bit; this also shifts the rounding position to the left by one bit position; (ii) in effective sub-

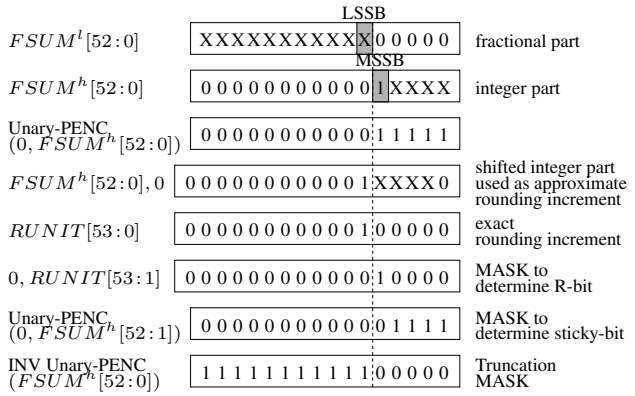

Figure 4. Patterns used for the implementation of IEEE rounding combined into the computation of significand addition.

tractions the leading one could be shifted to the right: This can be either by one or more positions. If it is by one position (case (a)) also the rounding position is just shifted by one position. If it is by more than one position (case (b)), this means that numerous significand bits are canceling out. This is only possible if initially the leading one of the two significands had been differing by at most one bit position. In this case the significand difference can be represented exactly with 53 bits and rounding is not needed.

The cases where the prediction of the LSSB position needs to be corrected by one bit position can be taken care of in a similar way as in the implementation from [5].

The actual rounding decision needs to be based on the rounding mode, the R-bit and a sticky bit as described in [6]. For these computations the R-bit can be determined by using the representation of $runit$ as a mask shifted right by one position. This mask does not need to be applied to the compressed sum, it can be applied to the smaller of the two operands (the larger operand has only zeros following the LSSB position). The sticky-bit can be determined very similarly: Instead of the shifted $runit$ representation, the shifted inverted exact truncation mask is used for the masking in this case (see figure 4).

The availability of the R-bit and the sticky-bit then allow the determination of the rounding decision which can then in turn determine which output of the significand adders needs to be chosen.

4.4 Significand Overflow and Underflow

Significand overflow and underflow are defined by the condition that the upper part of the significand sum computation does not include the leading one of the significand result. It can be expressed by the condition that $fsum^h \notin [1, 2^{53})$. Significand overflow occurs in the case that $fsum^h \geq 2^{53}$ and significand overflow occurs in the case that $fsum^h = 0$. Note that significand overflow can only occur for effective additions and significand underflow can only occur for effective subtractions. These are the only two cases where the exponent of the result is not equal to the exponent of the larger of the two operands el. For significand overflow we have $e.res = el + 1$, for significand underflow we have $e.res = el - 1$. We can easily precompute these 6-bit exponents and make a selection based on the detection of significand overflow or underflow.

5 Conclusions

We are proposing the use of a (non-redundant) high-radix representation for the floating-point operands and results of a FP adder. The use of this format allows the optimization of the FP adder design in that alignment and normalization shifts are significantly simplified. The implementation for these shifts can be split into five different cases, three of which can be easily joined with only minor delay overhead, thereby also reducing delays for the exponent difference on the critical path. The discussion of the implementation identifies as the main difficulty the fast implementation of IEEE rounding based on the HR representations, where the rounding position can vary over 53 different places. An implementation is outlined where the implementation of IEEE rounding only adds a minor overhead of a few logic levels for adding the rounding increment approximation and for masking the result to the delay of a conventional compound adder. Together with the delay savings that can be achieved for alignment shifts, normalization shifts and exponent difference computations the proposed organization promises a large potential for fast IEEE FP adder implementations. A detailed evaluation of practical implementations following our guidelines was beyond the scope of this manuscript and needs to be considered in future work.

References

[1] H. Fahmy and M. Flynn. The case for a redundant format in floating point arithmetic. In *Proc. 16th IEEE Int. Symp. on Computer Arithmetic (Arith16)*, pages 95–103, 2003.

[2] P. Farmwald. *On the design of high performance digital arithmetic units*. PhD thesis, Stanford Univ., Aug. 1981.

[3] IEEE standard for binary floating point arithmetic. ANSI / IEEE std. 754-1985, New York, 1985.

[4] A. Nielsen, D. Matula, C. Lyu, and G. Even. An IEEE compliant floating-point adder that conforms with the pipelined packet-forwarding paradigm. *IEEE Transactions on Computers*, 49(1):33–47, 2000.

[5] P.-M. Seidel. How to halve the latency of IEEE compliant floating-point multiplication. In *Proceedings of the 24th EUROMICRO Conference*, pages 329–332, 1998.

[6] P.-M. Seidel and G. Even. Delay-optimized implementation of IEEE floating-point addition. *IEEE Transactions on Computers*, 53(2):97–114, 2004.