An IEEE compliant floating-point adder that conforms with the pipelined packet-forwarding paradigm *

Asger Munk Nielsen[†] David W. Matula[‡] C. N. Lyu[§] Guy Even[¶]

June 6, 1999

Abstract

This paper presents a floating point addition algorithm and adder pipeline design employing a packet forwarding pipeline paradigm. The packet forwarding format and the proposed algorithms constitute a new paradigm for handling data hazards in deeply pipelined floating point pipelines.

The addition and rounding algorithms employ a four stage execution phase pipeline with each stage suitable for implementation in a short clock period, assuming about fifteen logic levels per cycle. The first two cycles are related to addition proper and are the focus of this paper. The last two cycles perform the rounding and have been covered in a paper by Nielsen and Matula [8].

The addition algorithm accepts one operand in a standard binary floating point format at the start of cycle one. The second operand is represented in the packet forwarding floating point format, namely, it is divided into four parts: the sign bit, the exponent string, the principal part of the significand, and the carry-round packet. The first three parts of the second operand are input at the start of cycle one and the carry-round packet is input at the start of cycle two. The result is output in two formats that both represent the rounded result as required by the IEEE 754 standard. The result is output in the packet forwarding floating point format at the end of cycles two and three to allow forwarding with an effective latency of two cycles. The result is also output in standard IEEE 754 binary format at the end of cycle four for retirement to a register.

The packet forwarding result is thus available with an effective two cycle latency for forwarding to the start of the adder pipeline or to a cooperating multiplier pipeline accepting a packet forwarding operand. The effective latency of the proposed design is two cycles for successive dependent operations while preserving IEEE 754 binary floating point compatibility.

*A preliminary version of this work appeared as "Pipelined Packet-Forwarding Floating Point: II. An Adder" in the Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA, July 6 - July 9, 1997, pp. 148–155.

†Dept. of Mathematics and Computer Science Odense University, Denmark.

‡SEAS, Southern Methodist University Dallas, Texas. Supported in part by a grant from Cyrix Corporation and by the Texas Advanced Technology Program Grant 003613013.

§???

¶Dept. of Electrical Engineering Systems, Tel-Aviv University, Tel-Aviv 69978, Israel. E-mail:guy@eng.tau.ac.il. Supported in part by the North Atlantic Treaty Organization under a grant awarded in 1996. Supported in part by Intel Israel LTD. under a grant awarded in 1997.

1 Introduction

Background. Although simple by conception, floating point addition is a surprisingly complex arithmetic operation, since it involves several time consuming dependent operations. The conceptual steps of the algorithm are (see Fig. 1 (a)): Compute the exponent difference, align the significand with the smaller exponent by shifting it right by an amount equal to the exponent difference, add or subtract the significands, normalize the sum by shifting out leading zeros to the left, and round the final result. As described, the algorithm consists of two potentially full precision shifts and three additions (exponent subtract, significand addition, and rounding). Two of the three addition operations are slow in the sense that they require a delay that scales logarithmically with the precision of the number.

In order to minimize latency, the addition unit can be divided into two separate datapaths operating in parallel (see Fig. 1 (b)), as investigated in numerous contributions to the literature e.g. [5, 12, 13, 7]. The left path operates under the assumption that the exponents of the two operands are different by no more than a unit, i.e. $|e_1 - e_2| \leq 1$. In this case only a short pre-alignment shift of at most one position is needed. Since the operands are close in range, the difference (or sum if the operands have opposite signs) of the aligned significands may be close to zero, and consequently a potentially long normalization shift is required. If, on the other hand, the exponent difference is "large", i.e. $|e_1 - e_2| \geq 2$, a potentially long pre-alignment shift is necessary. It follows in this case that the post normalization step is trivial, due to absence of significant cancellation.

Further minimization of latency was suggested by combining the rounding increment with the addition of the significands [13]. This minimization requires that the addition of the significands compute the *sum*, $sum + 1$, and $sum + 2$ in parallel by using a compound adder. Quach and Flynn [13] report a latency of less than 20ns for a double precision floating-point adder that was laid out in a 1μ technology. Modern microprocessors require $2 - 4$ clock cycles for floating point addition; the faster the clock is, the more clock cycles are required[1]. Different implementations of floating point adders have different precisions and some use multiply-and-add units, hence, comparing the latencies of floating-point adders is not trivial and technology dependent. However, we estimate that each pipeline stage in our algorithm requires 15 logic levels, and that the pipeline can be implemented with a clock rate of 450MHz (even with static CMOS). This means that the pipelined packet-forwarding paradigm can lead to designs with shorter latencies.

Oberman *et al.* [11] proposed a variable latency floating point addition algorithm that is based on a three stage pipeline. Their algorithm is designed to output the sum after one, two, or three cycles, depending on the values of the operands. They propose to reduce the average latency using their algorithm, but still have a worst case latency of three clock cycles.

Related Work. This paper relies on two previous papers [2, 8]. In the paper of Daumas and Matula [2] recodings of redundant numbers are defined and investigated. These definitions and properties play a crucial role in our addition algorithm since they enable us to avoid compressing

[1] Some examples are [9]: (a) DEC-Alpha 21164 with a clock rate of 500MHz requires 4 clock cycles; (b) MIPS R10000 with a clock rate of 200MHz requires 2 clock cycles; and (c) HP PA-8000 with a clock rate of 250MHz requires 4 clock cycles (and uses a fused multiply-add unit)

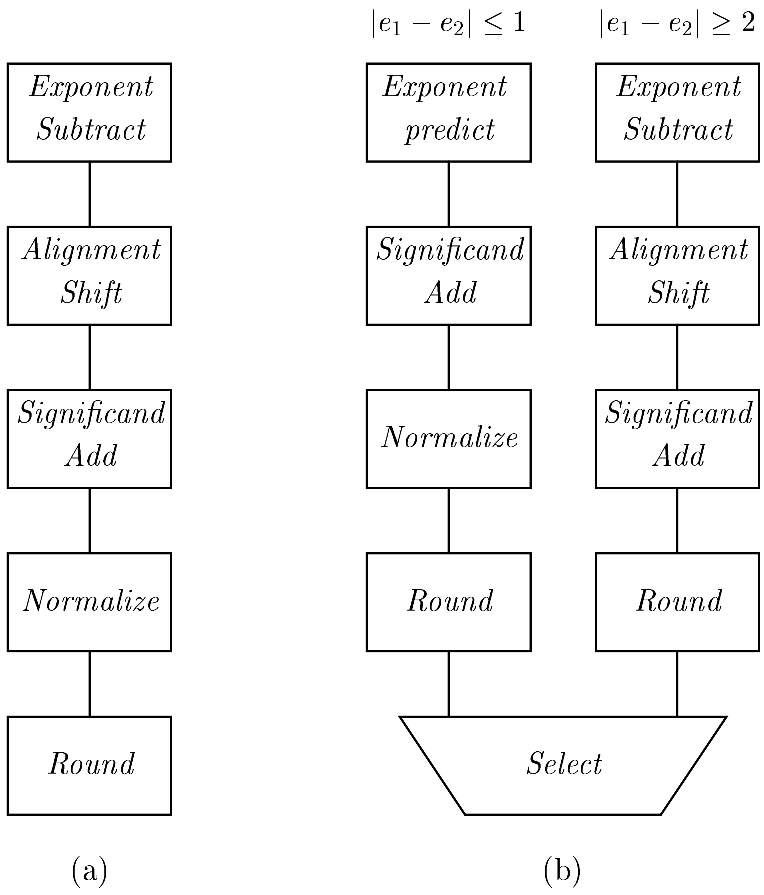

Figure 1: Traditional floating point addition algorithms

the sum into a non-redundant number, and allow for obtaining a short latency. In addition, Daumas and Matula show how to recode a borrow save encoded digit string to obtain a radix 4 Booth recoding. This recoding can be employed to design a floating-point multiplier that complies with the pipelined packet forwarding paradigm.

The paper of Nielsen and Matula [8] presents a *packet forwarding floating point format*, sets the pipelining paradigm with packet forwarding, and presents a design of a rounding unit for such a micro-architecture. In our paper we present only the first two pipeline stages of the adder. The last two stages of the pipeline are the rounding unit that is presented in [8]. The same rounding unit can be used in a multiplier that conforms with the pipelined packet forwarding format.

The rounding unit presented in [8] is fed a significand in the range $(1, 4)$ that is encoded by a borrow save digit string of 132 digits. During the third stage (which is the first rounding stage), the rounding unit computes the rounding decision and outputs the 2-digit carry-round packet. During the last stage, the rounding unit adds the 64 most-significant digits and the carry-round packet to obtain the rounded standard (non-redundant) 64-bit significand.

Contribution. The main contribution of this paper is in demonstrating the feasibility of a floating point addition algorithm that complies with the pipelined packet-forwarding paradigm.

We distinguish between (a) redundant adders, that output a redundant representation of the sum, such as $3:2$- and $4:2$-adders; and (b) non-redundant adders that output a non-redundant binary representation of the sum (e.g. 2:1-adders). The delay of redundant adders is constant and does not depend on the length of the addends, whereas the delay of a non-redundant adder scales logarithmically with the length of the addends.

We show that successive dependent floating point additions in the pipelined packet-forwarding paradigm can be performed with only one non-redundant addition at the end of the whole computation. The elimination of intermediate non-redundant additions plays a key role in reducing the latency of our floating point addition algorithm.

Overview. Our goal is to design a floating-point adder suitable for implementation with a short clock period with an effective latency of two clock cycles for successive dependent additions. In terms of logic levels, the effective latency of the proposed algorithm is roughly 30 logic levels since we estimate the latency of each stage to be roughly 15 logic levels.

To meet this goal, we rearrange and avoid non essential 2:1-addition steps of traditional floating point addition implementations. This is achieved by using the packet forwarding floating point format [8]. A preliminary exploration of this format was described in [7]. Since one of the inputs of the adder can be represented in redundant format, there is no need to perform a 2:1-addition when the result is forwarded. Hence, the only 2:1-addition step in the addition algorithm is deferred to the latter portion of the rounding phase where it contributes no delay to the data forwarding process. The only additions that are employed before the rounding step are redundant additions .

Figure 2 depicts a block diagram of the proposed addition algorithm. The adder accepts one operand in standard format, denoted by (s_1, e_1, f_1), where s_1 denotes the sign bit, e_1 denotes the exponent string, and f_1 denotes the significand string. The adder accepts the other operand in the pipelined packet-forwarding format (reviewed in Sec. 2.1) denoted by $(s_2, e_2, f_2^+, f_2^-, c_2)$, where s_2 denotes the sign bit, e_2 denotes the exponent string, f_2^+ and f_2^- denote the borrow-save encoded principal part, and c_2 denotes the carry-round packet. Note, that the carry-round packet is only fed at the beginning of the second clock cycle. The rounded sum is output in two formats that represent equal values. At the end of the fourth cycle, the result is output according to the standard IEEE 754 binary format and may be stored in a floating-point register. The result is also output in the pipelined packet forwarding floating point format at the end of cycles two and three to allow forwarding with an effective latency of two cycles. The sign bit, exponent string, and borrow-save encoded principal part of the sum are output at the end of the second cycle, and the carry-round packet of the result is output at the end of the third cycle.

The proposed addition algorithm is also divided into two separate datapaths operating in parallel depending on the exponent difference. Note, that the range used for the small exponent difference is slightly extended to simplify the processing in the large exponent difference path (see Sec. 4). The first two cycles of the pipeline deal with addition proper, and are the focus of this paper. The input to the third pipeline stage consists of a sign bit, an exponent string, and a significand in the range $(1, 4)$ represented by borrow-save encoded 132-digit string. The input to the third pipeline stage is not necessarily the precise sum, but represents a value that is rounded to the same value that the exact sum is rounded to.

The last two stages of the pipeline are presented in [8] and consist of computing the signs (i.e. negative, zero, or positive) of the "lower half" and the "upper part" of the borrow-save encoded digit string. These signs together with the least significant, guard, and round digits determine the carry-round packet which is output at the end of cycle three. In cycle four, the standard result is obtained by a 4:2-redundant addition of the principal part packet and the carry-round packet that results with a borrow-save encoded digit string. This borrow-save encoded digit string is then compressed into binary encoded bit string and normalized to the range $[1, 2)$ to produce a the standard result at the end of cycle four. Thus, only one non-redundant addition is used, and this addition takes place in the fourth pipeline stage. More details on the computation of the carry round packet appear in Appendix A.

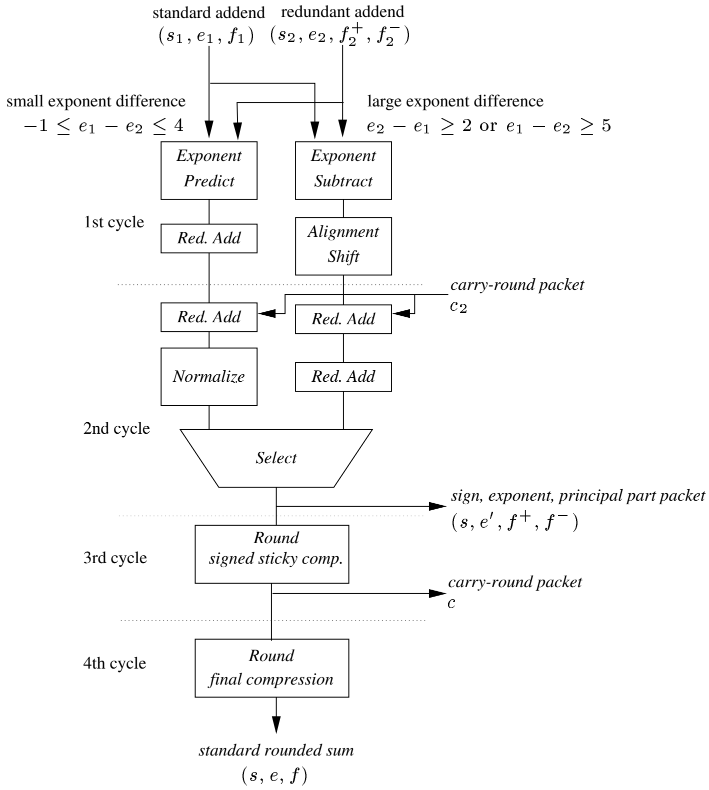

Figure 2: Block diagram of the proposed floating point addition algorithm

Organization. In Sec. 2 we review the general setting and the main techniques of our algorithm. This includes a review of the pipelined packet forwarding format, the pipelined packet forwarding, recoding, and partial compression. In Sec. 3, the adder design for small exponent differences is discussed. In Sec. 4, the adder datapath for large exponent differences is described. Summary conclusions are given in Section 5. Appendix A outlines how the carry-round packet is computed. Appendix B contains proofs of the compression properties of 3:2- and 4:2-redundant adders.

2 Preliminaries

In this section we provide a brief description of the packet-forwarding format, recoding of borrow-save encoded digit strings, and partial compression results. The reader is referred to the paper of Daumas and Matula [2] and to Appendix B for more details and proofs.

2.1 pipelined packet-forwarding format

In this section we briefly review the pipelined packet-forwarding floating-point format presented by Nielsen and Matula [8].

A *standard* double extended IEEE 754 floating-point operand with unique factorization

$$(-1)^s \cdot 2^e \cdot f$$

as specified in [1], consists of (see Fig. 3 (a)):

1. a sign bit denoted by s;
2. an encoded exponent field of 15 bits denoted by e; and
3. a normalized binary significand having 63 bits to the right of the radix point denoted by $f = 1.a_1a_2 \cdots a_{63}$.

The *packet-forwarding floating-point format* [8] is depicted in Fig. 3 (b). A floating-point operand is represented by four parts: a sign bit, an exponent string, the principal part of the significand, and the carry-round packet. The packet-forwarding format is a redundant representation of floating-point numbers, and thus, factorization is not unique. The factorization of floating-point numbers according to the packet-forwarding format is

$$(-1)^s \cdot 2^e \cdot (f + c \cdot 2^{-63}) \tag{1}$$

with the same sign bit and exponent format as in the standard double extended precision format. The significand factor, $(f + c \cdot 2^{-63})$, in Eq. 1 is in the pre-normalized range $1 \leq f + c \cdot 2^{-63} \leq 4$ and is partitioned into two parts:

1. $\mathbf{f} = 1b_0.b_1b_2 \cdots b_{62}$ with $b_i \in \{-1, 0, 1\}$ is a 64 digit borrow-save encoded digit string termed the *principal part packet.* Using the notation in Fig. 3 (b), each digit $b_i \in \{-1, 0, 1\}$ of the principal part is represented by a positive bit b_i^+ and a negative bit b_i^-, where $b_i = b_i^+ - b_i^-$.
2. $\mathbf{c} = c_{62}c_{63}$, termed the *carry-round packet*, is a two digit borrow-save encoded digit string with $\|c_{62}c_{63}\| \in \{-2, -1, 0, 1, 2\}$. Note that a two digit borrow-save encoded digit string can have the values 3 or (-3), but the carry-round packet is restricted to the set $\{-2, -1, 0, 1, 2\}$.

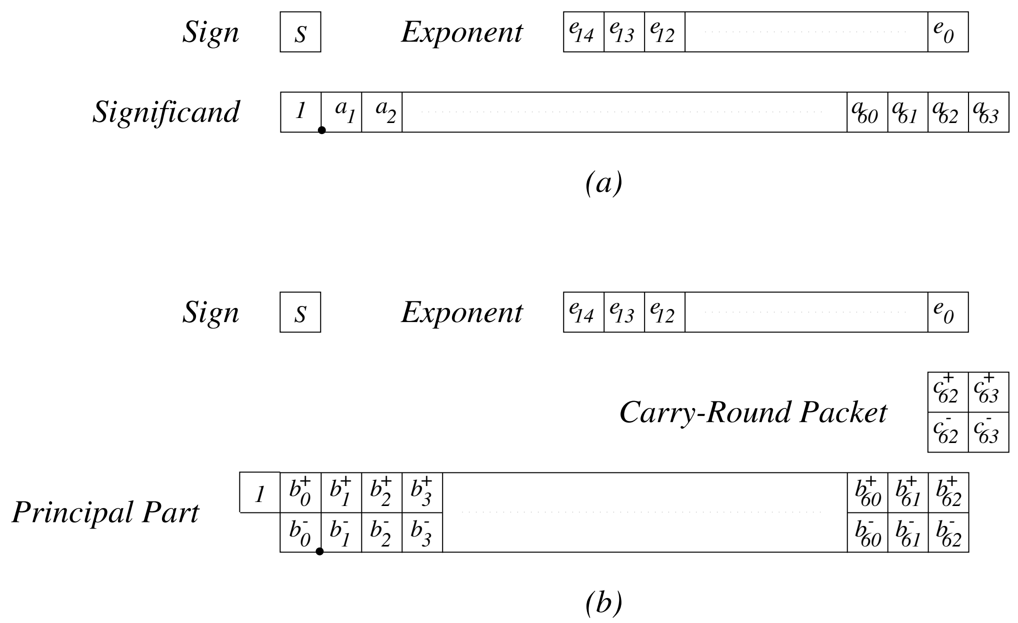

Figure 3: Floating-point operand formats: (a) Standard IEEE 754 operand format; (b) Packet-forwarding operand format.

One can easily verify that powers of two (e.g. $x = 2^k$, where k is an integer) have up to three possible factorizations, namely, the significand factor can equal either $1, 2$ or 4, provided that the exponent lies within the range of representable exponent values. Values that are not powers of two have two factorizations, namely, the significand factor can belong either to the interval $(1, 2)$ or to the interval $(2, 4)$, provided that the exponent lies within the range of representable exponent values. This slackness plays a key role in making it possible to avoid full compression of the results and reduction of the latency of forwarded results.

2.2 the pipelined packet-forwarding paradigm

In this section we briefly overview the pipelining paradigm suggested by Matula and Nielsen [8].

A floating point adder or floating point multiplier according to the pipelined packet-forwarding paradigm is implemented by a pipeline of four stages as depicted in Fig 4. The first two stages perform the operation (i.e. addition or multiplication), and the last two stages deal with rounding. The pipeline accepts two operands: one in standard format and the other in the pipelined packet-forwarding format. The operand in standard format is fed to the first stage, and the operand in the pipelined packet-forwarding format is fed to the first two stages as follows. The sign-bit, the exponent, and the principal part are fed to the first stage. The carry-round packet is fed to the second stage. The result is output in two formats that represent the same value. The standard rounded result is output by the fourth stage. The result is also output according to the pipelined packet-forwarding format as follows: the sign, exponent, and principal part of the result are output by the second stage, and the carry-round packet is output by the third stage.

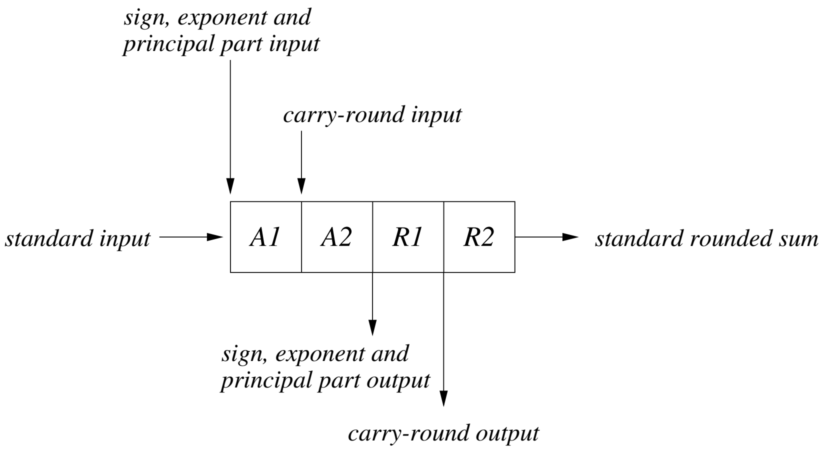

Figure 4: Input-Output schedule of pipeline

Figure 5 depicts an execution of successive dependent floating-point operations utilizing the proposed pipelining paradigm. The effective latency is two clock cycles and there is only one stall cycle between successive dependent operations.

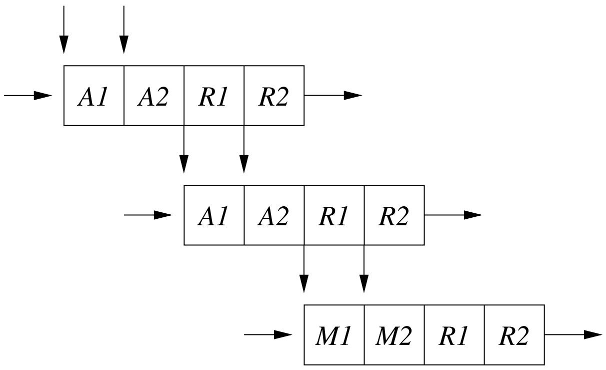

Figure 5: Execution of successive dependent operations

2.3 recoding

In this section we briefly review P- and N- recoding presented by Daumas and Matula [2] and extend them to 3:2-recodings. We review four definitions of recodings (P, N, mmp, and ppm) and their implementations [2, 4, 6, 10].

We represent a k-digit borrow-save encoded digit string $\mathbf{a} = a_{k-1}a_{k-2}\cdots a_0$ by a $2 \times k$ bit array. The digit $a_i \in \{-1, 0, 1\}$ is encoded by a "positive" bit $a_i^+ \in \{0, 1\}$ and a "negative" bit $a_i^- \in \{0, 1\}$. The value of the digit a_i equals $a_i = a_i^+ - a_i^- \in \{-1, 0, 1\}$.

Recodings of borrow-save encoded digit strings were defined by Daumas and Matula [2]. First, we define recodings of single borrow-save digits depicted in Fig. 6.

Definition 1 *Let (a^+, a^-) denote a borrow-save digit. The P-recoding of (a^+, a^-) is a 2-digit borrow save encoded string, denoted by $(x^+, 0), (0, x^-)$, where*

$$\begin{aligned} x^+ &= a^+ \text{ AND } not(a^-) \\ x^- &= a^+ \text{ XOR } a^-. \end{aligned}$$

The N-recoding of $(a^+, a^-)$is a 2-digit borrow save encoded string, denoted by $(y^+, 0), (0, y^-)$, where

$$\begin{aligned} y^+ &= a^+ \text{ XOR } a^- \\ y^- &= a^- \text{ AND } not(a^+). \end{aligned}$$

Note that P and N-recodings do not change the value. Namely, $a^+ - a^- = 2x^+ - x^-$ and $a^+ - a^- = -2y^- + y^+$.

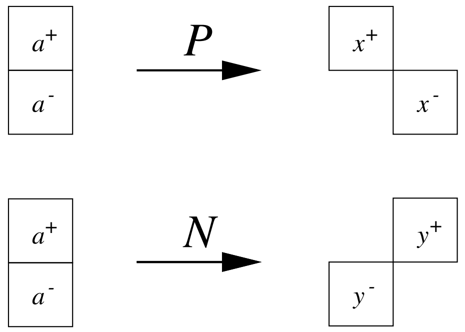

Figure 6: P and N recodings of a single borrow save digit.

The following definition extends the definitions of P- and N-recodings to borrow-save encoded digit strings, as depicted in Fig. 7.

Definition 2 *Let $\mathbf{a} = a_{k-1}a_{k-2}\cdots a_0$ denote a k-digit borrow-save encoded digit string. The P-recoding of $\mathbf{a}$, denoted by $\mathbf{b} = P(\mathbf{a})$, is a $k+1$-digit borrow-save encoded digit string defined by:*

$$\begin{aligned} b_i^+ &= \begin{cases} a_{i-1}^+ \text{ AND } not(a_{i-1}^-) & \textit{if } i \in \{1, \ldots, k\} \\ 0 & \textit{if } i = 0 \end{cases} \\ b_i^- &= \begin{cases} a_i^+ \text{ XOR } a_i^- & \textit{if } i \in \{0, \ldots, k-1\} \\ 0 & \textit{if } i = k \end{cases} \end{aligned}$$

The N-recoding of a, denoted by $\mathbf{c} = N(\mathbf{a})$, is a $k+1$-digit borrow-save encoded digit string defined by:

$$c_i^+ = \begin{cases} a_i^+ \text{ XOR } a_i^- & \textit{if } i \in \{0, \ldots, k-1\} \\ 0 & \textit{if } i = k \end{cases}$$

$$c_i^- = \begin{cases} not(a_{i-1}^+) \text{ AND } a_{i-1}^- & \textit{if } i \in \{1, \ldots, k\} \\ 0 & \textit{if } i = 0 \end{cases}$$

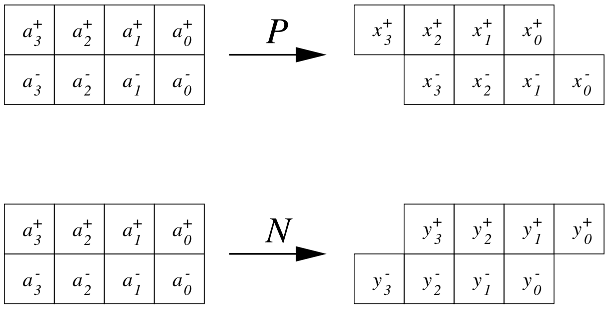

Figure 7: P and N-recoding of borrow save encoded digit strings

Note that P- and N-recodings can be implemented by a circuit that resembles a Half-Adder line: only one of the inputs of an AND-gate needs to be inverted. This implies that delay associated with computing P- and N-recodings equals the maximum of the delay of a XOR-gate and the sum of the delays of an inverter and an AND-gate.

P- and N-recodings recode a borrow-save encoded digit strings and are analogous to Half-Adder lines. Our addition algorithm is based also on 3:2-recodings that are analogous to Full-Adder lines, namely, 3:2-counters [4, 6]. These recodings are fed digit strings in which each digit compromises three bits: two bits have positive weight and one bit has negative weight (or vice-versa), and output borrow-save encoded digit strings.

We generalize the definitions of P- and N-recodings to 3:2-recodings as depicted in Fig. 8. Consider three bits a^+, a^-, b^+ that represent the value $a^+ - a^- + b^+$. A *ppm*-recoding adds these three bits into two bits x^+, x^- that represent the value $2x^+ - x^-$ as follows:

$$x^+ = \begin{cases} 1 & \text{if } a^+ - a^- + b^+ \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$x^- = a^+ \oplus a^- \oplus b^+$$

The reason we consider *ppm*-recoding to be a generalization of P-recoding is that

$$ppm(a^+, a^-, 0) = P(a^+, a^-)$$

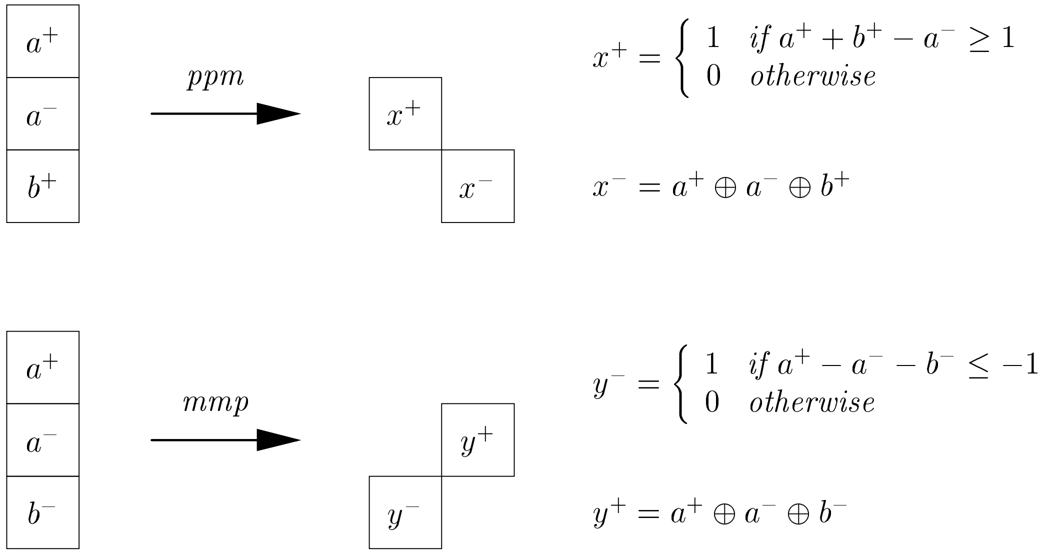

Figure 8: *ppm*- and *mmp*-recodings

Similarly, given three bits a^+, a^-, b^- that represent the value $a^+ - a^- + b^-$, *mmp*-recoding adds these three bits into two bits y^+, y^- that represent the value $2y^+ - y^-$ as follows:

$$\begin{aligned} y^- &= \begin{cases} 1 & \text{if } a^+ - a^- + b^- \leq -1 \\ 0 & \text{otherwise} \end{cases} \\ y^+ &= a^+ \oplus a^- \oplus b^- \end{aligned}$$

The reason we consider *mmp*-recoding to be a generalization of N-recoding is that

$$mmp(a^+, a^-, 0) = N(a^+, a^-)$$

Note that *ppm*- and *mmp*-recodings can be extended to recodings of $3 \times k$ bit arrays. In fact, *ppm*- and *mmp*-recodings are redundant additions in which a $3 \times k$ bit array is transformed into a $2 \times (k+1)$ bit array that represents the same value. One can implement *mmp*- and *ppm*-recodings by a Full-Adder line in which one of the inputs of the carry-bit circuit is inverted (for example, in *ppm*-recoding the b^- input is inverted). This implies that delay associated with computing *ppm* and *mmp*-recodings is not greater than the delay of a Full-Adder plus the delay of an inverter.

2.4 partial compression

In this section we briefly review the partial compression properties of P- and N- recodings proved by Daumas and Matula [2]. We extend these properties to 3:2-recodings and to 4:2-recodings.

Given a radix polynomial $\sum_{i=m}^{\ell} d_i \cdot 2^i$, the value obtained by "chopping" off leading digits and inserting a radix point immediately to the left of the "tail", is called the *fraction value at position* j. Formally, the fraction value at position j is denoted in terms of the digit string $d_m d_{m-1} \cdots d_\ell$ as follows:

$$
\begin{aligned}
f_j(d_m \cdots d_\ell) &= \|0.d_{j-1}d_{j-2}\cdots d_\ell\| \\
&= \sum_{i=\ell}^{j-1} d_i 2^{i-j}.
\end{aligned}
$$

The digit string $\mathbf{d} = d_m d_{m-1} \cdots d_\ell$ is said to have *fraction range values in the range* (c,d) if $f_j(d_m \cdots d_\ell) \in (c,d)$ for all $j \in \{\ell+1,\ldots,m+1\}$. The *width* of the fraction range (c,d) is $d-c$.

Daumas and Matula [2] proved that if a borrow-save encoded digit string B has fraction values in the range (c,d), then $P(B)$ has fraction values in the range $(\frac{c}{2}-\frac{1}{2},\frac{d}{2})$, and $N(B)$ has fraction values in the range $(\frac{c}{2},\frac{d}{2}+\frac{1}{2})$. The compound PN-recoding, $P(N(B))$, has fraction values in the range $(\frac{c}{4}-\frac{1}{2},\frac{d}{4}+\frac{1}{4}) \subseteq (-\frac{3}{4},\frac{1}{2})$. Successive recodings reduce the width of the fraction ranges; for example, if the initial width is 2, then successive recodings reduce the width to $1\frac{1}{2}, 1\frac{1}{4}, \ldots$. This reduction in the width of the fraction range is termed *partial compression*.

The partial compression obtained by P- and N-recoding can be extended to 3:2-recodings. We summarize the partial compression properties of 3:2-recodings and of 3:2- and 4:2-redundant adders in the next claim; the proof appears in Appendix B.

Claim 3 *Let $a^+, a^-, b{+}^+, b^-$ denote k-bit strings (i.e. $a^+ = a^+_{k-1}\cdots a^+_0$). Let α, β and γ denote radix polynomilas the coefficients of which are defined by: $\alpha_i = a^+_i - a^-_i + b^+_i$, $\beta_i = a^+_i - a^-_i - b^-_i$ and $\gamma_i = a^+_i - a^-_i + b^+_i + b^-_i$. Let $x = ppm(a^+, a^-, b^+)$ and let $y = mmp(a^+, a^-, b^-)$.*

1. *If the fraction range of α is in (c,d) (where $-1 \le c \le 0 \le d \le 2$), then the fraction range of x is in $(\frac{c}{2}-\frac{1}{2},\frac{d}{2})$.*
2. *If the fraction range of β is in (c,d) (where $-2 \le c \le 0 \le d \le 1$), then the fraction range of y is in $(\frac{c}{2},\frac{d}{2}+\frac{1}{2})$.*
3. *If the fraction range of γ is in (c,d) (where $-2 \le c \le 0 \le d \le 2$), then the fraction range of $z = mmp(x, b^-)$ is in $(\frac{c}{4}-\frac{1}{2},\frac{d}{4}+\frac{1}{2})$.*

The implication of Claim 3 is that a 3:2-redundant adder implemented by *ppm*-recoding has the same partial compression properties that P-recoding has. Similarly, a 3:2-redundant adder implemented by *mmp*-recoding has the same partial compression properties that N-recoding has. Moreover, a 4:2-redundant adder of two borrow-save encoded strings implemented by *ppm*-recoding followed by *mmp*-recoding reduces the fraction range from (c,d) to $(\frac{c}{4}-\frac{1}{2},\frac{d}{4}+\frac{1}{2})$.

3 Adder Datapath for Small Exponent Differences

In this section we describe the first two pipeline states of the addition algorithm for the case that the exponent difference is small. We have adjusted the exact range of exponent differences to be

$-1 \le e_1 - e_2 \le 4$. This adjustment simplifies significantly the algorithm for the case of a large exponent difference at a modest cost for the case of a small exponent difference.

The execution for the case that the exponent difference is small is logically separated into two cases depending on whether the input carry-round packet makes a significant contribution to the resulting sum: (a) When the carry-round packet is not significant to the shift length, our principal result is the following. Employing the standard operand significand and the principal part of the packet format operand we can determine the post alignment normalization shift to within one of two final positions in the first cycle. (b) When the carry-round packet is significant to the shift length, our algorithm yields the correct result compromising at most seven digits properly aligned by the alternative logic case.

Notation. Let (s_1, e_1, f_1) denote the sign-bit, exponent, and significand of the operand given in the IEEE Standard's format. Let $(s_2, e_2, f_2^+, f_2^-, c^+, c^-)$ denote the second operand given in the packet-forwarding format; that is, f_2^+, f_2^- denotes the positive and negative bit strings of the principal part packet $\mathbf{f}$ and c^+, c^- denotes the positive and negative bit strings of the carry-round packet $\mathbf{c}$.

Description. Figure 9 depicts the addition algorithm operating under the assumption that $-1 \le e_1 - e_2 \le 4$. Since the exponent difference is small, it suffices to consider only the 3 LSB's of the exponents in order to compute the difference $e_1 - e_2$. This difference determines the alignment shift of f_1. The extension of the small exponents difference range from $[-1, +1]$ to $[-1, 4]$ makes the alignment shift somewhat costlier. However, the algorithm for the large exponents difference is greatly simplified, as discussed in Sec. 4. Note that only f_1 is shifted, and that "traditional operand swapping" is not performed. The reason for this is that the significand of the second operand is represented as a borrow-save number and has twice as many bits as f_1, and hence, restricting the alignment shift only to f_1 saves hardware.

While the significand f_1 is being aligned, the redundant significand is negated, if necessary, and recoded. Note, that negating a borrow-save number amounts to swapping the positive and negative weight vectors. The recoded redundant significand and the aligned non-redundant significand are then added using a 3-2 adder. The redundant significand has bits in positions $[-1 : 62]$ and the non-redundant significand has bits in positions $[-4 : 64]$. (Note, that we index position to the left of the radix point with negative indices, and positions to the right of the radix point are indexed with positive indices). The bits in positions $[63 : 64]$ of the non-redundant significand are passed through without modification. The 3-2 adder adds the bits of the non-redundant significand in positions $[-4 : 62]$ with the redundant significand. The output of the 3-2 adder (which has bits in positions $[-5 : 62]$) is PN-recoded. The output of the recoding (which has bits in positions $[-6 : 62]$) is concatenated with the bits of the non-redundant significand bits in positions $[63 : 64]$, and the resulting sum is denoted by $\mathbf{g}$. The sum $\mathbf{g}$ is sent to to destinations: (a) the registers between the two pipeline stages; and (b) a leading-zero estimator.

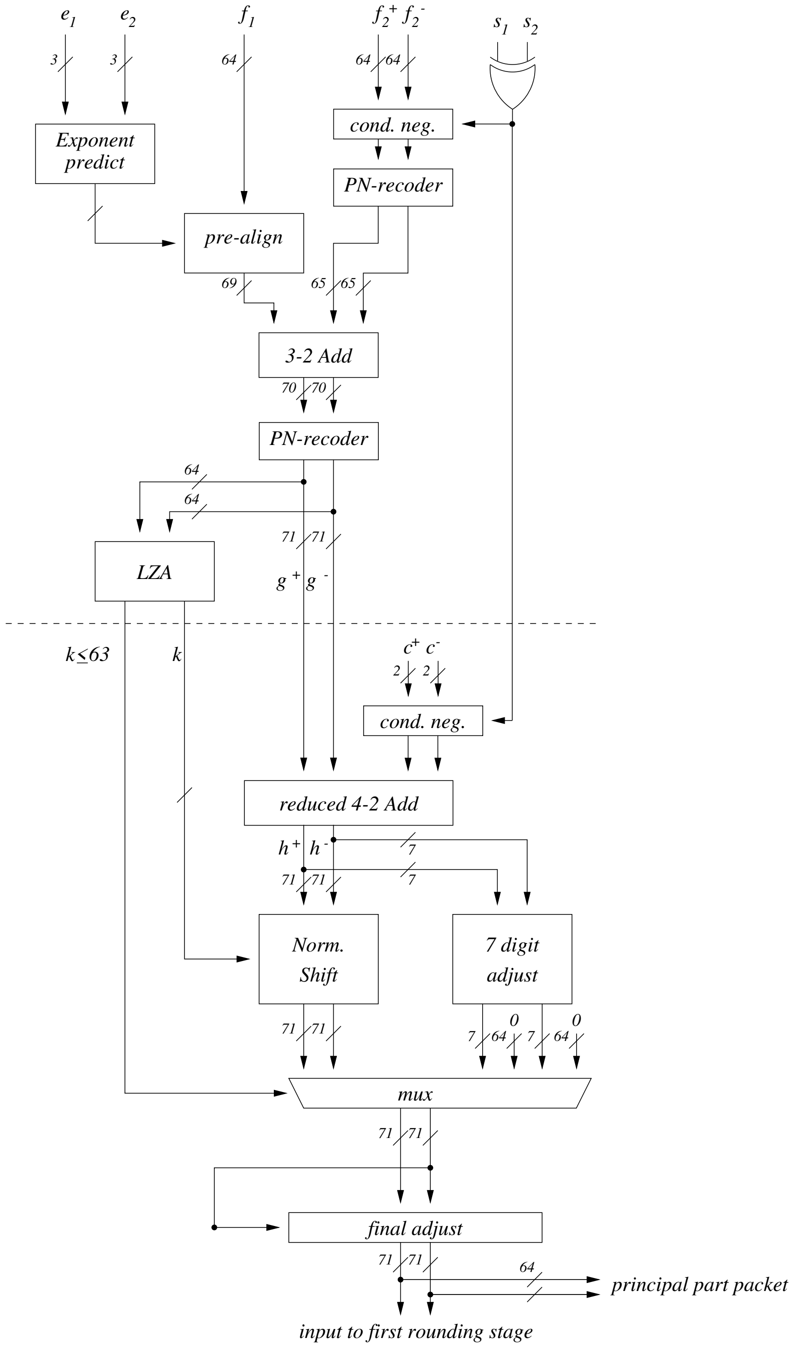

Figure 9: Adder datapath for $-1 \leq e_1 - e_2 \leq 4$.

Since we are dealing with the case of a small exponents difference, the sum (or difference) of the significands can be in the range $(-1, 1)$. The rounding unit assumes that a non-zero significand is in the range $[1, 4]$, and hence, a normalization shift may be required.[2] A naive approach is to compress the sum to a non-redundant representation, count the number of leading zeros, and perform a left shift. We avoid this time-consuming 2-1 addition as follows.

First, the 64 most-significant positive and negative bits of the recoded borrow-save encoded sum are pairwise XORed. The resulting 64-bit binary string is fed to a leading-zeros counter, LZA [3]. Hence, the number of leading zeros in the representation of the sum using borrow-save digits determines the shift amount. Note that the borrow-save encoding of the sum might have a large number of *leading insignificant one's*, i.e. a plus one followed by a string of minus ones or conversely a minus one followed by a string of plus ones. The existence of leading insignificant one's makes it hard to estimate the range of the sum. The recodings have a key role in restricting the adverse effect of leading insignificant one's so that the shifted sum belongs to one of two possible binades, as formalized in Lemma 4. This completes the description of the first clock cycle.

In the second clock cycle, we first add the carry-round packet, denoted by $\mathbf{c}$, to the sum, denoted by $\mathbf{g}$. This is done in two steps: First, the carry-round packet is negated, if necessary, just as the redundant significand was. Then, a redundant addition takes place. Since the redundant significant is not shifted, the position of the carry-round packet is not changed, and hence, the redundant addition simply computes: $h = \mathbf{g} + (-1)^{s_1 \oplus s_2} \cdot c \cdot 2^{-63}$. Claim 5 shows that this redundant addition can be performed by an redundant adder of 4 digits such that this addition does not generate a carry that changes the 64 most significant digits of $\mathbf{g}$.

After the carry-round packet has been added to the sum, the normalization shift takes place. The leading-zeros computation outputs two signals: "$k \leq 63$" - a flag that signals whether there is a non-zero digit among the 64 most-significant digits of $\mathbf{g}$; and "k" - the number of leading zeros in $\mathbf{g}$ in case there is a non-zero digit among the 64 most-significant digits of $\mathbf{g}$. The normalization is split into two paths, depending on the signal $k \leq 63$. If $k \leq 63$, then a normalization shift takes place, and the most-significant digit of shifted sum is guaranteed to be non-zero. If $k > 63$, then out of the 71 digits of the sum, the 64 most-significant digits are zeros. The remaining 7 digits are normalized and 64 zeros are padded to the right.

After the correct path has been selected according to the signal $k \leq 63$, a final adjustment takes place. This adjustment performs the following tasks: (a) detect the sign of the sum and then negate and shift the sum accordingly so that the principal part is within the range $(1, 4)$, as prescribed by Claim 6; and (b) adjust the two most significant digits of the sum so that the representation complies with the packet-forwarding format.

The proposed algorithm computes in effect

$$f_1 \cdot 2^{e_1 - e_2} + (-1)^{s_1 \oplus s_2} \cdot (f_2 + c \cdot 2^{-63})$$

Let s denote the sign of this sum. The algorithm extracts s from the most-significant negative bit that is fed to the final adjust box. The sign-bit of a non-zero final sum equals $s \oplus s_1$. When the final sum equals zero, the sign bit is determined by the rounding-mode if the operands are not both

[2] The case of a zero sum is discussed later. The case of a denormal sum is usually treated by using an extended exponent range so that even significands of denormal results need to be properly normalized.

zeros, or by the the signs of the operands (if the operands are both zeros). Therefore, an additional signal is generated during the second cycle to indicate whether the final sum is zero.

The exponent packet of a non-zero sum is defined by:

$$e = \begin{cases} e_2 - (k-6) - adjust & \text{if } k \leq 63 \\ e_2 - (64-6) - adjust' - adjust & \text{otherwise} \end{cases}$$

Where *adjust* denotes the shift amount that is done by the final adjust box. Namely, *adjust* is either 1 or 2 depending on the s. If $k \geq 64$, an additional left shift is carried out by the 7 digit adjust box. This shift amount is denoted by $adjust'$. In case the sum equals zero, the exponent needs to be set to $e_{\min}$.

This completes the description of the second clock cycle. The sign-bit, exponent, and principal part packet are ready to be forwarded as well as input to the rounding unit, as described in [8].

Correctness. The following lemma demonstrates the advantage of using carry recodings for partial compression. The lemma shows that the recodings nearly eliminate the range ambiguity caused by leading insignificant ones (and minus ones) in the borrow-save encoding of the sum. This lemma is instrumental in reducing the problem of computing the normalization shift amount to a problem of computing the number of leading zeros in a binary string.

Lemma 4 *Suppose the sum* $\mathbf{g}$ *is non-zero and let* $(0^k, \sigma, t)$ *denote the digit string representing the recoded sum, in which there are k leading zeros followed by a non-zero digit* $\sigma \in \{-1, 1\}$ *and a digit string* $t \in \{-1, 0, 1\}^{p-k-1}$. *If* $k \leq 63$, *then* $\sigma.t$ *(where the dot between* σ *and* t *is a radix point) is in the range* $(\frac{9}{32}, \frac{29}{32})$ *for* $\sigma = 1$ *and* $(-\frac{23}{16}, -\frac{35}{64})$ *for* $\sigma = -1$.

Proof: The borrow-save number $f_2 = f_2^+ - f_2^-$, with fraction range $(-1, 1)$, is PN-recoded. Therefore, the fraction range of the redundant significand that is input to the 3-2 adder is $(-\frac{3}{4}, \frac{1}{2})$. The fraction range of the non-redundant significand is $[0, 1)$. Therefore, the fraction range of the output of the 3-2 adder is $(-\frac{7}{8}, \frac{3}{4})$. The output of the 3-2 adder is PN-recoded, and hence, the output of the recoding, $\mathbf{g}[-6:62]$ has fraction range $(-\frac{23}{32}, \frac{7}{16})$.

The sum $\mathbf{g}$ also has two bits $g^+[63:64]$ that originate from the non-redundant significand. We assume that $k \leq 63$, therefore, the contribution of $g^+[63:64]$ to the fraction range in positions $[-6:56]$ is non-negative and less than 2^{-6}. Thus the fraction range of the sum $\mathbf{g}$ in positions $[-6:56]$ is $(-\frac{23}{32}, \frac{7}{16} + \frac{1}{64})$. Let $a = -\frac{23}{32}$ and $b = \frac{7}{16} + \frac{1}{64}$.

Consider the borrow save number $(0^k, \sigma, t)$. If $\sigma = 1$, then $0.1t \in (a, b)$, and thus, $1.t \in (1+a, 1+b) \cap (2a, 2b)$. This implies that $1.t \in (\frac{9}{32}, \frac{29}{32})$. Similarly we deduce $\bar{1}.t \in (-\frac{23}{16}, -\frac{35}{64})$. □

The sum $\mathbf{g}$ that is input to the second pipeline stage consists of 71 borrow-save digits, 7 of which are to the left of the radix point and 64 of which are to the right of the radix point. As mentioned above, the two least significant digits (i.e. $\mathbf{g}[63:64]$) originate uninterrupted from the aligned non-redundant significand, and therefore, consist of two (positive) bits.

The following claim shows that the (possibly negated) carry-round packet can be added with the 4 least significant digits of the sum $\mathbf{g}$ without generating a carry.

Claim 5 *Let* $\mathbf{g}[61:64]$ *denote the* 4 *least-significant borrow-save digits of the redundant sum that are input to the second pipeline stage, and let* $\mathbf{c}[63:62]$ *denote the* 2 *borrow-save digits of the carry round packet. Then,*

$$-12 \le \sum_{i=0}^{3} \mathbf{g}[64-i] \cdot 2^i + 4 \cdot \mathbf{c}[62] + 2 \cdot \mathbf{c}[63] \le 11$$

Proof: Define: $A = 2\mathbf{g}[63] + \mathbf{g}[64]$, $B = 2\mathbf{g}[61] + \mathbf{g}[62]$, and $C = 2\mathbf{c}[62] + \mathbf{c}[63]$. Since A originates from the non-redundant significand f_1, it follows that $0 \le A \le 3$. The PN-recoding performed just before the end of the first clock cycle, insures that $-2 \le B \le 1$. (Note that this recoding starts at position 62.) Finally, by definition, $-2 \le C \le 2$. Hence, $-12 \le A + 4B + 2C \le 11$ and the claim follows. □

As stated above, the sum $\mathbf{g}$ has 7 digits to the left of the radix point and 64 digits to the right of the radix point. Let $\mathbf{g}$ be of the form $(0^k, \sigma, t)$, as in Lemma 4. The normalization shift shifts $\mathbf{h}$ by k positions to the left and positions the radix point between σ and t. The combined effect of the shift and the repositioning of the radix point amounts to scaling the fraction by 2^{k-6}. Thus, the value output by the normalization shift is $\mathbf{h} = (\mathbf{g} + \mathbf{c} \cdot 2^{-63}) \cdot 2^{k-6}$. The next claim shows that, even after the addition of the carry round packet, the normalized sum belongs to a range of two binades.

Claim 6 *If* $k \le 63$*, then normalized sum* h *satisfies:*

1. *If* $\sigma = 1$*, then* $\mathbf{h} \in \left(\frac{1}{4}, 1\right)$.
2. *If* $\sigma = -1$*, then* $\mathbf{h} \in \left(-2, -\frac{1}{2}\right)$.

Proof: If $\sigma = 1$, then Lemma 4 implies that

$$\mathbf{h} \quad \in \quad (\frac{9}{32}, \frac{29}{32}) + \mathbf{c} \cdot 2^{k-69}$$

Similarly, If $\sigma = -1$, then

$$\mathbf{h} \quad \in \quad (-\frac{23}{16}, -\frac{35}{64}) + \mathbf{c} \cdot 2^{k-69}$$

Our assumption that $k \le 63$ implies that

$$\left| \mathbf{c} \cdot 2^{k-69} \right| \le \frac{1}{32}.$$

and the claim follows. □

Claim 6 implies that if the partially normalized fraction is positive, then by a shift of two positions to the left, the normalized sum is shifted to the range $(1, 4)$. If the sum is negative, then by a negation and a shift by one position to the left, the normalized sum is shifted to the range $(1, 4)$. Thus, in the final adjustment stage, depending on the sign, we either negate the bits of the redundant significand and do a hardwired left shift, or perform a hardwired left shift by two positions. Note that the sign of the sum can be taken directly from the negative MSB before the final adjust, and that the positive MSB can be ignored. The reason is that the leading-zeros computation and subsequent normalization shift guarantee that the most-significant digit is non-zero.

4 Adder Datapath for Large Exponent Differences

In this section we describe the first two pipeline stages of the addition algorithm for the case that the exponent difference is large. This case is characterized by a large alignment shift and a small normalization shift (at most one position). In our algorithm, the alignment shift takes place during the first cycle although the carry round packet is input only at the beginning of the second cycle. Only the principal part (or standard format input) needs to be aligned by a variable length right shift. The late arriving carry-round packet can be directed to one of just two locations to complete the addition . A variable shifting of the carry-round packet is avoided by the choice of the threshold that separates between the case of a small exponent difference and a large exponent difference. Thus, the sign of the exponent difference $e_1 - e_2$ determines the location of the late arriving carry-round packet.

Description. Figure 10 depicts the addition algorithm operating under the assumption that $e_1 - e_2 \geq 5$ or $e_2 - e_1 \geq 2$. The first cycle begins with a full subtraction of the exponents. The magnitude of the difference $e_1 - e_2$ is limited by 66, since alignment shifts of 66 positions or more yield the same rounded result. Meanwhile, the redundant significand is negated, if necessary, and PN-recoded. The sign of the exponent difference controls which significand is aligned. Note, that the output of the *Swap* box is a borrow-save number, and that encoding the non-redundant significand as a borrow-save number is done by putting zeros in the negative vector. The significand with the larger exponent is sent to the second cycle. The significand with the smaller exponent is sent to two boxes: (a) The *high order alignment* box is a shifter capable of shifting a 66-digit borrow-save number by 2 to 66 positions to the right. (b) The *low order generator* computes the bit string $0^{64-Expmag} \cdot 1^{Expmag}$ and performs a bitwise AND between this string and the 64 least-significant digits of the significand (note that the PN-recoding introduces an additional digit to the left of the radix point). This produces the bits of the significand that would be shifted beyond bit position 65 and participate only in the generation of the *sticky digit* in the rounder. Hence, justifying these bits to the right does not change the sticky-digit. The advantage of this approach is that the carry-round packet needs to be placed only in one of two positions, depending on the sign of $e_1 - e_2$, as depicted in Fig. 11. This completes the description of the first clock cycle.

At the beginning of the second cycle, the high-order part of the shifted operand and the non-shifted operand are added by a 4-2 adder which outputs the sum **g**. Meanwhile, the carry-round packet, which is input at the beginning of the second clock cycle, is negated if necessary, just as the redundant significand was. The sign of the exponent difference determines whether the carry-round packet is added to the high order part or to the low order part, as depicted in Fig. 11. If $e_1 > e_2$, then then the redundant fraction f_2 was shifted and the carry-round packet is added in fixed positions 128 and 129. If $e_2 > e_1$, then the non-redundant fraction f_1 was shifted and the carry-round packet is added in fixed positions 62 and 63. Since we have separated the computation into high and low order parts, care should be taken, so that the addition of the carry-round packet modifies only one part. In particular, when the carry-ripple adder is added to the low order part, a ripple effect is not allowed. Claims 7 and 8 show that the addition of the carry-round packet can be performed by constant width 4-2 adders.

After the carry-round packet is added either to the high order part or to the low order part, the following adjustment takes place. (A) The sign of the redundant sign is negative iff $s_1 \neq s_2$ and $e_2 \geq e_1$. The reason is that in this case f_2 is negated even though it dominates the sum. Thus, when the above condition holds, the redundant sum must be negated. (B) The large exponent difference implies that the magnitude of the redundant sum is in the range $(\frac{1}{2}, 4\frac{1}{2})$. Therefore, if an effective addition took place (e.g. $s_1 = s_2$), then a right shift by one position might be required to bring the redundant sum to the the range $[1, 4)$. Note, that it is not necessary to find the exact value of the redundant sum (which would be just as slow as compressing it), and that it suffices to find out whether the redundant sum greater than $2\frac{1}{2}$ with an error of $\frac{1}{2}$. This can be determined by the digits whose weight is at least $\frac{1}{2}$, and there are only a constant number of such bits. Similarly, if an effective subtraction took place (e.g. $s_1 \neq s_2$), then a left shift by one position might be required to bring the redundant sum to the range $[1, 4)$. (C) Finally, an adjustment of the digits to the left of the radix point guarantees that the representation complies with the packet-forwarding format.

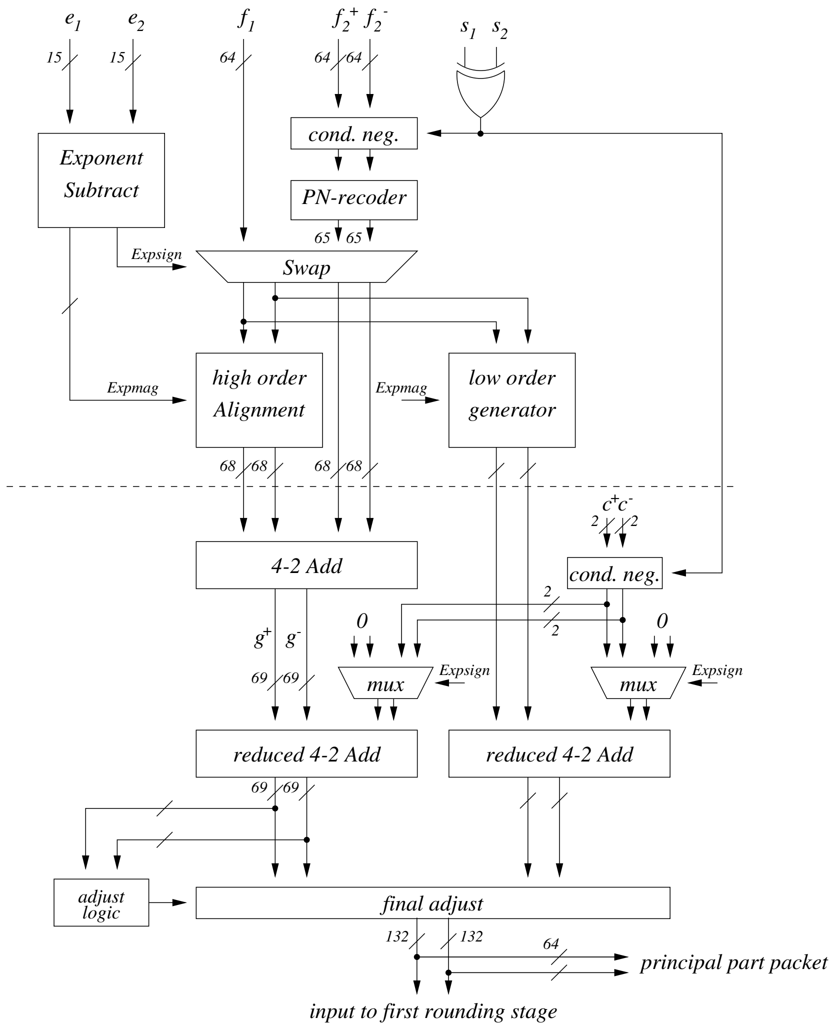

Figure 10: Adder datapath for $e_2 - e_1 \geq 2$ or $e_1 - e_2 \geq 5$.

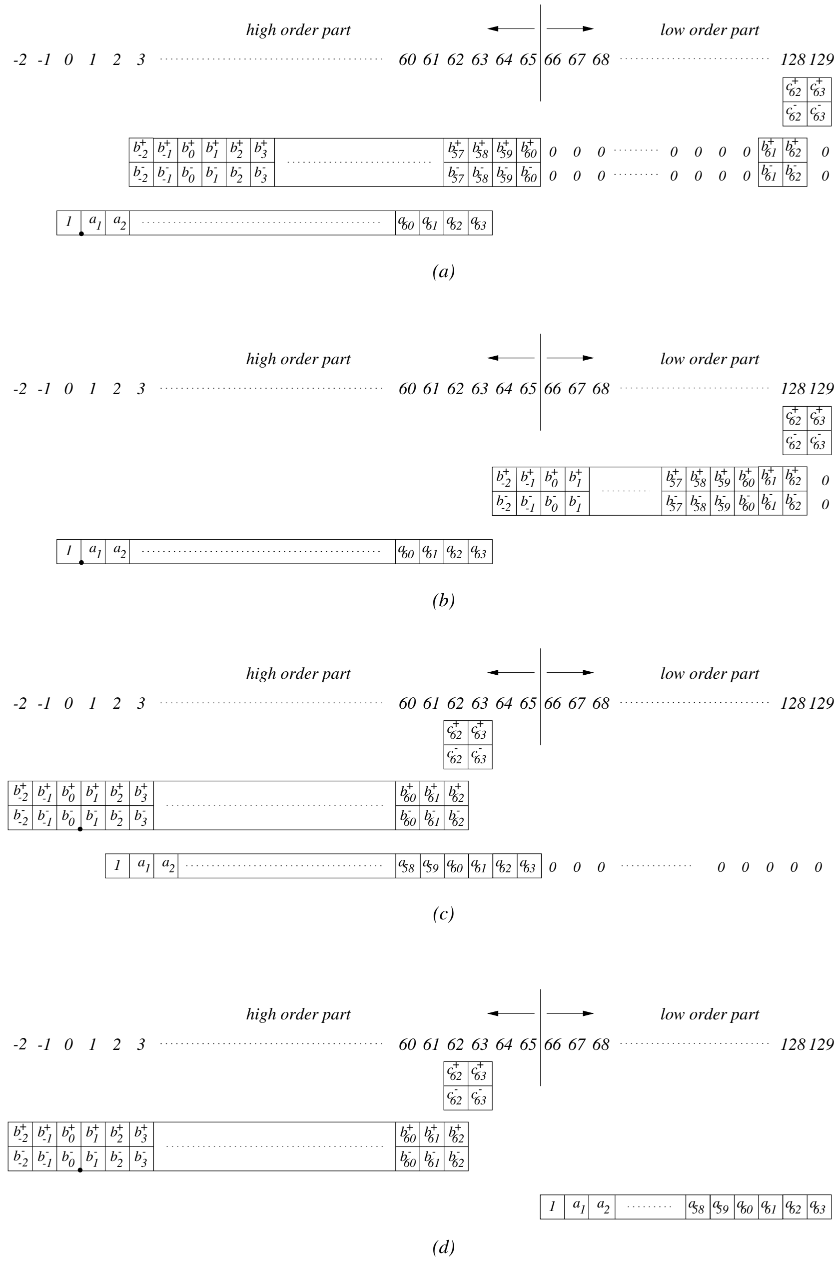

Figure 11: (a) alignment when $e_1 = e_2 + 5$; (b) alignment when $e_1 \geq e_2 + 66$; (c) alignment when $e_2 = e_1 + 2$; (d) alignment when $e_2 \geq e_1 + 66$;

The sign-bit of the sum is determined by the sign of the exponent difference, denoted by *Expsign*. If $e_1 > e_2$, then the sign bit equals s_1, and if $e_2 > e_1$ then the sign-bit equals s_2. The exponent of the sum equals $\max\{e_1, e_2\} + adjust$, where *adjust* denotes the shift that is performed by the final adjustment.

This completes the description of the second clock cycle. The sign-bit, exponent, and principal part packet are ready to be forwarded as well as input to the rounding unit, as described in [8].

Correctness. The following claim shows that a 3-digit adder suffices for adding the carry-round packet with the low-order part when $e_1 > e_2$.

Claim 7 *Let $c_{62}c_{63}$ denote the borrow-save digits of the carry-round packet. Consider the borrow-save number $f_2' = b'_{-2}b'_{-1}b'_0.b'_1b'_2\cdots b'_{62}$ obtained by possibly negating the redundant significand f_2 and performing a PN-recoding. Then, the sum of $b'_{61}b'_{62}$ and $c_{62}c_{63}$ can be represented by 3 digits, namely,*

$$-7 \leq 4 \cdot b'_{61} + 2 \cdot b'_{62} + 2 \cdot c_{62} + c_{63} \leq 7$$

Proof: From [2], it follows that $-2 \leq 2 \cdot b'_{61} + b'_{62} \leq 1$. The carry-round packet is in the range $[-2, 2]$, and the claim follows. □

The following claim shows that a 4-digit adder suffices for adding the carry-round packet with the high-order part when $e_2 > e_1$. The proof relies on the compression properties of the 4-2 adder that adds the significands.

Claim 8 *Let $c_{62}c_{63}$ denote the borrow-save digits of the carry-round packet. Consider the borrow-save number $\mathbf{g} = g_{-3}g_{-2}g_{-1}g_0.g_1g_2\cdots g_{65}$ computed by the 4-2 adder in Fig. 10. If the adder is implemented by cascading two 3-2 adders (i.e. ppm cells followed by mmp cells), then the sum of $g_{60}g_{61}g_{62}g_{63}$ and $c_{62}c_{63}$ can be represented by 4 digits, namely,*

$$-15 \leq 8 \cdot g_{60} + 4 \cdot g_{61} + 2 \cdot g_{62} + g_{63} + 2 \cdot c_{62} + c_{63} \leq 15$$

Proof: The fraction range of the adder's input is $[-\frac{3}{4}, 1\frac{1}{2}) = [0, 1) + (-\frac{3}{4}, \frac{1}{2})$, because the redundant significand is PN-recoded. Therefore, the fraction range of the output of the adder is $[-\frac{11}{16}, \frac{7}{8})$. The carry-round packet is in the range $[-2, 2]$, and hence, the fraction range of $0.00c_{62}c_{63}$ is $[-\frac{1}{8}, \frac{1}{8}]$. The sum of these fraction ranges is $[-\frac{13}{16}, 1)$, and the claim follows. □

5 Summary and Conclusion

An addition algorithm conforming with the packet-forwarding pipeline paradigm is presented. The adder accepts one operand in the standard format and the second operand in a *packet-forwarding floating point format* [8]. Note that a standard format operand can be trivially translated to conform with the packet-forwarding format, and therefore, the assumption on the second operand does not restrict inputting two standard format operands. The sum is output in the packet-forwarding floating point format starting at the end of the second cycle, as well as in the standard format at the end of the fourth cycle. This algorithm rearranges and avoids non essential 2:1-addition steps of the traditional floating point addition implementations. In fact, the processing of forwarded result that is output in the packet-forwarding format does not contain a 2:1-adder of the length of the significand; the only non-redundant addition is the exponent subtraction. This enables outputting of a nearly complete significand termed the *principal part packet* at the end of the second cycle. The rounding, which takes place during the third and fourth cycles [8], produces the *carry-round packet* at the end of the third cycle, and the standard format sum at the end of the fourth cycle. The value encoded by the packet-forwarding floating-point format output is equivalent to the standard IEEE 754 rounded output. By these means only one 2:1-addition is performed, and it is deferred to the latter portion of the rounding phase where it contributes no delay to the data forwarding process. All the additions in the first two cycles are 4:2- and 3:2-redundant additions.

Note that the proposed addition algorithm can fuse a sequence of dependent additions without intermediate 2:1-additions. Only the final sum is subjected to a 2:1-addition for retiring to a register. Nevertheless, every addition is subject to proper IEEE rounding, and therefore, the final sum agrees with the IEEE standard sum where every intermediate result should be properly rounded.

The algorithm employs P- and N-recodings for obtaining partial compression of the redundant numbers [2]. The cost of P- and N-recodings is roughly equivalent to that of passing every borrow-save digit through one half-adder. Most of the recodings do not lie on the critical paths. The partial compression obtained by recoding is used in several places in the algorithm, among them: avoiding full compression before counting leading zeros and simplifying the addition of the late arriving carry-round packet.

The algorithm uses a different threshold for distinguishing between the cases of a small and a large exponent difference. This modification introduces a small increase in the cost of the datapath for the small exponent difference, but enables us to avoid shifting the carry-round packet by a variable amount in the large exponent difference datapath. Thus, the late arriving carry-round packet in the large exponent difference datapath is directed without a variable shift to one of two locations depending on the sign of the exponent difference.

The proposed design demonstrates the feasibility of a four stage pipelined addition algorithm complying with the packet-forwarding paradigm in which the depth of each pipeline stage is roughly 15 logic levels. An optimization of the designed should even further reduce the depth of the pipeline stages.

Further work relating to complying with other aspects of the IEEE standard is still required. Our presentation does not address IEEE exception detection and handling. The forwarding of the results may need to be disabled when exceptions such as overflow and underflow occur. One way to deal with exceptions is to compute early signals warning that an exception might occur. Such a warning signal can be used to disable the forwarding and recourse to the standard result.

References

[1] "IEEE standard for Binary Floating-Point Arithmetic"; ANSI/IEEE Standard 754-1985, New York, The Inst. of Electrical and Electronics Engineers, Inc, Aug. 1985.

[2] Daumas, M., Matula, D. W.: "Recoders for Partial Compression and Rounding"; Technical Report RR97-01, Ecole Normale Superieure de Lyon, LIP, available at http://www.ens-

lyon.fr/LIP.

[3] Dadda, L., Piuri, V., Salice, F.: "Leading Zero Detectors"; In Proc. of 2nd Intern. Conf. on Massively Parallel Computing Systems. Ischia, Italy, May 6-9, 1996, 409–416.

[4] Jean Duprat, Yvan Herreros, and Jean-Michel Muller, "Some Results about On-Line Computation of Functions"; Proc. 9th IEEE Symposium on Computer Arithmetic, pp. 112-118, September 1989.

[5] Farmwald, M. P.: "On the Design of High-Performance Digital Arithmetic Units"; PhD thesis, Stanford University, Aug. 1981.

[6] Alain Guyot, Bertrand Hochet, and Jean-Michel Muller, "JANUS, an On-Line Multiplier/Divider for Manipulating Large Numbers"; Proc. 9th IEEE Symposium on Computer Arithmetic, pp. 106-111, September 1989.

[7] Lyu, C.-N.: "Micro-Architecture of a Pipelined Floating-Point Execution Unit"; Ph.D. Thesis, SMU, Dallas, Texas, Dec. 1995.

[8] Matula, D. W., Nielsen, A. M.: "Pipelined Packet-Forwarding Floating Point: I. Foundations and a Rounder"; *13th IEEE Symposium on Computer Arithmetic*, Asilomar, California, July 6-9, 1997, pp. 140–147.

[9] Microprocessor Report, various issues, 1994-1997.

[10] A. M. Nielsen, "Number Systems and Digital Serial Arithmetic", Ph.D. Thesis, Odense University, Denmark, Aug. 1997.

[11] S. F. Oberman, H. Al-Twaijry, and M. J. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units"; *13th IEEE Symposium on Computer Arithmetic*, Asilomar, California, July 6-9, 1997, pp. 156–165.

[12] Quach, N. T., Flynn, M. J.: "An improved Floating Point Addition Algorithm"; Technical Report CSL-TR-90-442, Stanford University, Jun. 1990. (available at http://umunhum.stanford.edu/main.html).

[13] Quach, N. T., Flynn, M. J.: "Design and Implementation of the SNAP Floating-Point Adder"; Technical Report CSL-TR-91-501, Stanford University, Dec. 1991. (available at http://umunhum.stanford.edu/main.html).

A The Computation of the Carry-Round Packet

In this appendix we outline how the carry-round packet is computed during the third pipeline stage. A more detailed and optimized description appears in [8].

The input to the third pipeline stage is (s, e', f^+, f^-), where the $f^+ = 1b_0^+.b_1^+b_2^+ \cdots b_{130}^+$ and $f^- = b_0^- b_1^- \cdots b_{130}^-$ are bit strings that represent the significand before rounding. It is guaranteed that $f^+ - f^- \in (1,4)$. The principal part, which is output at the end of the second pipeline stage, consists of these significand strings chopped after position 62. This means that the algorithm has committed to a value represented by the significand's digits in positions -1 to 62, and the role of the carry-round packet is to correct this value to take into account the rounding decision. The carry-round packet is limited to values in the set $\{-2^{-62}, -2^{-63}, 0, 2^{-63}, 2^{-62}\}$.

The computation of the carry-round packet is based on computing the *signed-sticky* digit of a borrow-save encoded string, defined as follows.

$$signed\_stk(d_i d_{i+1} \cdots d_j) = \begin{cases} -1 & \text{if } \sum_{k=i}^{j} d_k \cdot 2^{-k} < 0 \\ 0 & \text{if } \sum_{k=i}^{j} d_k \cdot 2^{-k} = 0 \\ 1 & \text{if } \sum_{k=i}^{j} d_k \cdot 2^{-k} > 0 \end{cases}$$

Matula and Nielsen suggest a fast circuit for computing the signed-sticky digit [8].

Let $b_i = b_i^+ - b_i^-$. The range of $f^+ - f^-$ can be partitioned into $(1,2)$, $\{2\}$, and $(2,4)$ as follows.

$$\begin{aligned} signed\_stk(b_0 b_1 \cdots b_{130}) = -1 &\Rightarrow f^+ - f^- \in (1,2) \\ signed\_stk(b_0 b_1 \cdots b_{130}) = 0 &\Rightarrow f^+ - f^- = 2 \\ signed\_stk(b_0 b_1 \cdots b_{130}) = 1 &\Rightarrow f^+ - f^- \in (2,4) \end{aligned}$$

This partitioning into ranges determines which digits and signed-sticky digits are used for computing the carry-round packet. Suppose that the rounding mode and the sign bit s are used to determine one of three rounding modes: truncate, round up, or round to nearest even. We consider three cases:

1. If $f^+ - f^- \in (1,2)$, then the rounding decision is determined by b_{63}, b_{64}, and $signed\_stk(b_{65} \cdots b_{130})$. The rounding decision is to add a value $r \in \{-2^{-63}, 0, 2^{63}\}$ to $1b_0.b_1 \cdots b_{63}$. Since the principal part has been already output, the carry-round packet equals $b_{63} + r$.

2. If $f^+ - f^- = 2$, then the carry-round packet equals zero.

3. If $f^+ - f^- \in (2,4)$, then the rounding decision is determined by b_{62}, b_{63}, and $signed\_stk(b_{64} \cdots b_{130})$. The rounding decision is to add a value $r \in \{-2^{-62}, 0, 2^{62}\}$ to $1b_0.b_1 \cdots b_{62}$. Since the principal part has been already output, the carry-round packet equals r.

Matula and Nielsen suggest computing in parallel the signed-sticky digits $signed\_stk(b_0 \cdots b_{63})$ and $signed\_stk(b_{65} \cdots b_{130})$. These signed-sticky digits together with $b_{62}, b_{63}, b_{64}, s$ and the rounding-mode are used to compute carry-round packet [8].

B Partial Recoding - Proof of Claim 3

In this appendix we prove Claim 3. The proof technique is based on the proof of the lemma of Daumas and Matula [2] on the partial compression of P and N-recodings. The proof is presented in a schematic form to avoid the indices that are needed in a formal proof, but can be easily transformed into a formal proof.

Proof of part 1. We denote $x = ppm(a^+, a^-, b^+)$ by two bit strings x^+ and x^-. Our goal is to prove that the fraction range of (x^+, x^-) is in $(\frac{c}{2} - \frac{1}{2}, \frac{d}{2})$.

Figure 12 depicts a schematic proof of the partial compression of *ppm*-recoding. We consider a $2 \times \ell$ array consisting of $x_{\ell-1} \cdots x_0$ and put a radix point to the left of it. In the first line, we pluck the most-significant negative bit from the $2 \times \ell$ bit array that consists of x^+ and x^-. The contribution of this negative bit to the fraction range is either 0 or $-\frac{1}{2}$. We then apply a reverse *ppm*-recoding to the remaining $2 \times \ell$ bit array. Note that we have a leading zero between the radix point and the most significant digit of the $3 \times (\ell - 1)$ bit array, and thus the fraction range of the $3 \times (\ell - 1)$ bit array is $(\frac{c}{2}, \frac{d}{2})$, and the proof follows.

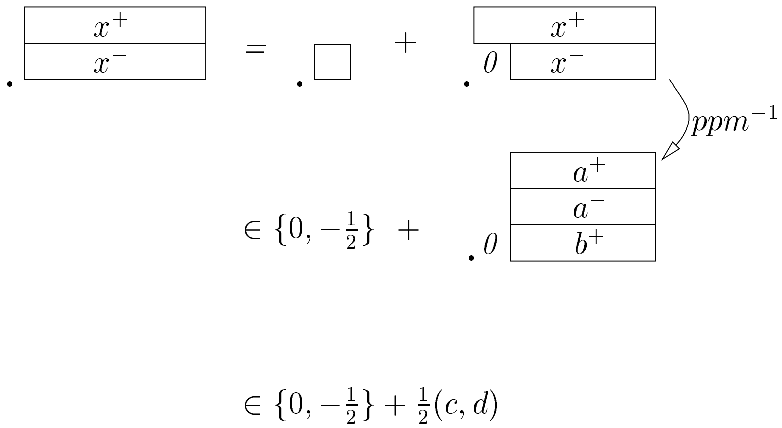

Figure 12: Schematic proof of the compression property of *ppm*-recodings

Proof of part 2. The proof of the partial compression property of *mmp*-recoding follows the same lines; only we pluck out the positive most significant bit and apply a reverse *mmp*-recoding.

Proof of part 3. We denote $z = mmp(x, b^-)$ by two bit strings z^+ and z^-. Our goal is to prove that the fraction range of (z^+, z^-) is in $(\frac{c}{4} - \frac{1}{2}, \frac{d}{4} + \frac{1}{2})$.

Figure 13 depicts a schematic proof of part 3. We consider a $2 \times \ell$ array consisting of $z_{\ell-1} \cdots z_0$ and put a radix point to the left of it. In the first line, we pluck the most-significant positive bit from the $2 \times \ell$ bit array that consists of z^+ and z^-. The contribution of this positive bit to the fraction range is either 0 or $\frac{1}{2}$. We then apply a reverse *mmp*-recoding to the remaining $2 \times \ell$ bit array. The reverse *mmp*-recoding yields a $3 \times \ell - 1$ bit array that consists of $x = ppm(a^+, a^-, b^+)$ and b^-. In the third line, we pluck the most-significant negative bits from the $3 \times \ell - 1$ bit array. The contribution of these two negative bits to the fraction range is either $0, -\frac{1}{4}$ or $-\frac{1}{2}$. In the fourth line, we apply a reverse *ppm*-recoding to $ppm(a^+, a^-, b^+)$, which yields the three bit strings: a^+, a^- and b^+. Note that we have two leading zeros between the radix point and the most significant digit of the $4 \times (\ell - 2)$ bit array, and thus the fraction range of the $4 \times (\ell - 2)$ bit array is $(\frac{c}{4}, \frac{d}{4})$, and the proof follows.

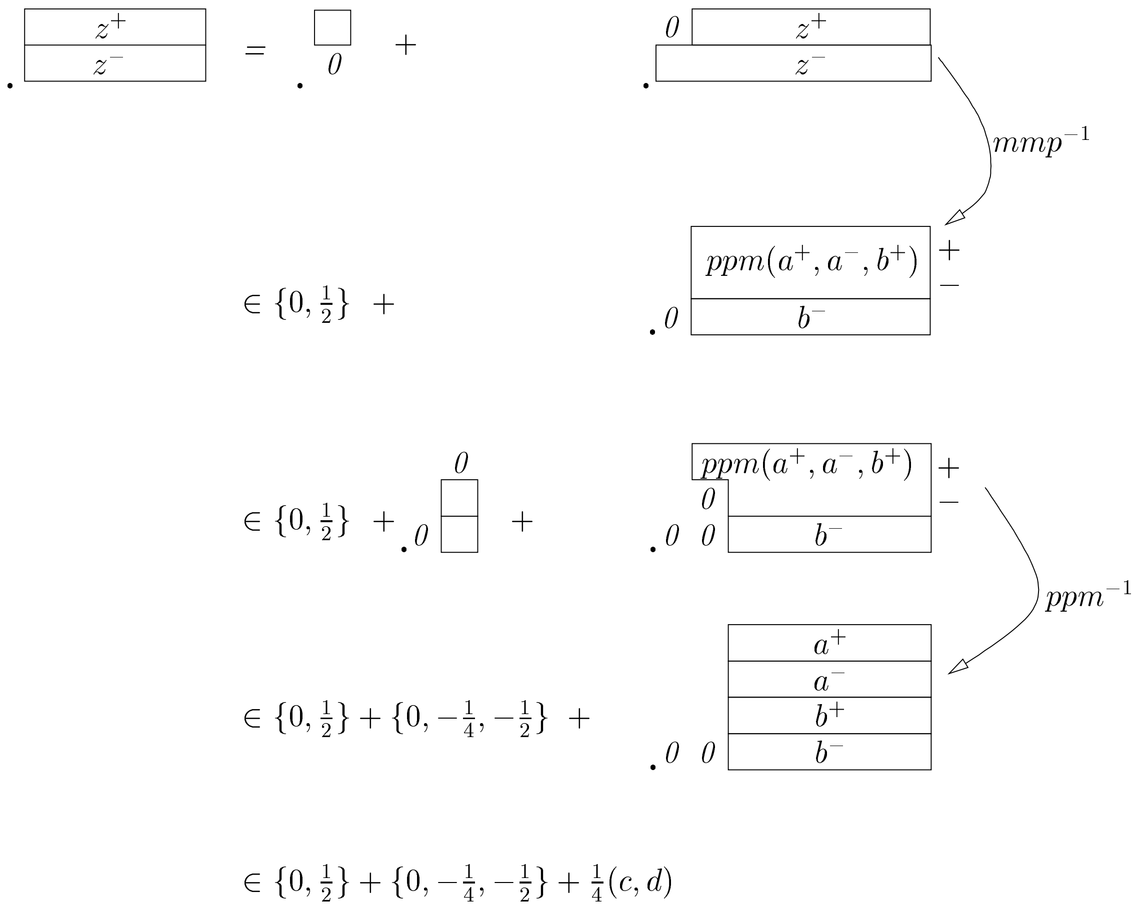

Figure 13: Schematic proof of the compression property of 4-2 addition