Pipelined Packet-Forwarding Floating Point: II. An Adder *

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

David W. Matula and C. N. Lyu
Dept. of Computer Science and Engineering
Southern Methodist University
Dallas, Texas

Guy Even‡
Univ. des Saarlandes
66123 Saarbrücken, Germany

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 algorithm employs 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 principal focus of this paper. The last two cycles perform the rounding and was covered in the first paper of this series [5]. The addition algorithm accepts one operand in a standard binary floating point format at the start of cycle one. Packets comprising the other operand in our packet forwarding floating point format are input at the start of cycles one and two. Output of the result occurs in the packet format after cycles two and three with the format representing a floating point value equal to the standard IEEE 754 rounded result. The same result in a standard binary floating point format is available after 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.

*This work was supported in part by a grant from Cyrix Corporation and by the Texas Advanced Technology Program Grant 003613013.

† Supported by grant no. 5.21.08.02 from the Danish Research Council.

‡ Supported by the North Atlantic Treaty Organization under a grant awarded in 1996.

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. [4, 6, 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 prealignment 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 prealignment shift is necessary. It follows in this case that the post normalization step is trivial, due to absence of significant cancellation.

Related Work. This paper relies on results presented in the following two papers [2, 5]. 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

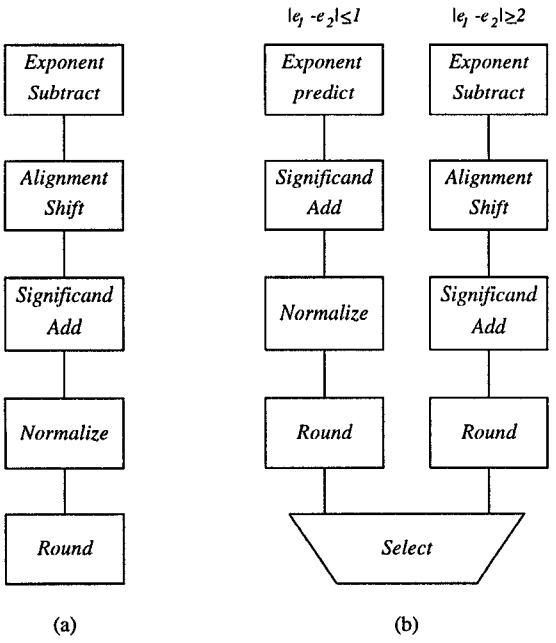

Figure 1. Traditional floating point addition algorithms

to avoid compressing the sum into a non-redundant number, and allow for obtaining a short latency. The first paper of this series [5] presents a packet forwarding floating point format, sets the pipelining scheme 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 presented in [5].

Contribution. 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. To meet this goal, we rearrange and avoid non essential 2-1 compression steps of traditional floating point addition implementations. This is achieved by using the packet forwarding floating point format [5]. A preliminary exploration of this format was described in [6]. Since one of the inputs of the adder can be represented in redundant format, there is no need to perform a 2-1 compression when the result is forwarded. Hence, the only 2-1 compression 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. Only 4-2 and 3-2 redundant additions are employed before the rounding step (see Fig. 2). While the value of the forwarded redundant result equals the value required by the IEEE Standard, the output of the rounding unit complies with the IEEE Standard also with regard to the format, and therefore, can be retired to a register.

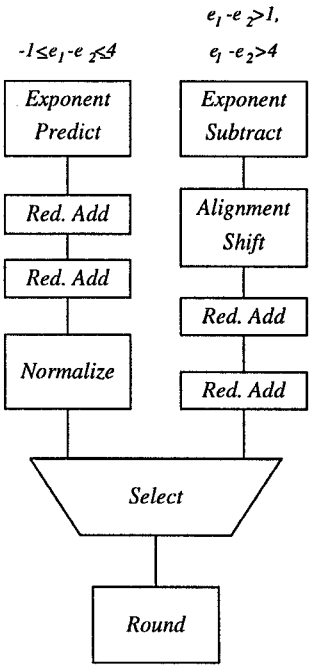

Figure 2. Proposed floating point addition algorithm

Overview. In Sec. 2 we briefly review the results of Daumas and Matula [2] on recoding and partial compression that we rely on. We define the packet forwarding format and refer the reader to paper [5] for further information on the properties of the representation as well as the rounder design.

The adder design for small exponent differences is discussed in Sec. 3. The execution is logically separated into two cases depending on whether the input carry-round packet makes a significant contribution to the post allignment normalization shift length. 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. When the carry-round packet is significant to the shift length, our algorithm yields the correct output comprising at most seven digits properly aligned by the alternative logic case.

The adder datapath for large exponent differences is described in Sec. 4. We show that only the principal part (or standard format input) needs to be aligned by a vari-

able length right shift. The late arriving carry-round packet can be directed to one of just two locations to complete the addition. Summary conclusions are given in Section 5.

2 Floating Point Formats, Recodings and Partial Compression

In this section we provide a brief description of recoding and partial compression results based on specific floating point formats that will be used herein. The reader is referred to the paper of Daumas and Matula [2] for more details and proofs.

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

$$a = (-1)^s 2^e f \tag{1}$$

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

- s is a sign bit,
- e is encoded in an exponent field of 15 bits,
- $f = 1.a_1a_2\cdots a_{63}$ with $a_i \in \{0,1\}$ is a normalized binary significand with $p = 64$ bit precision.

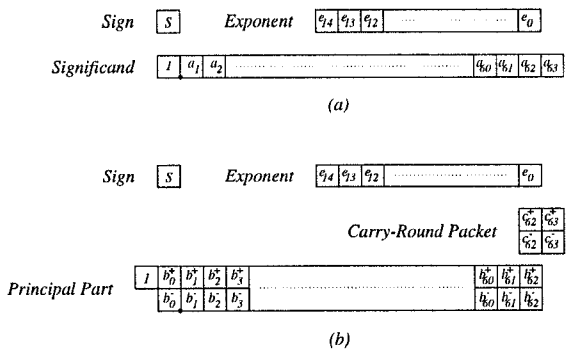

Figure 3. Floating point operand formats. (a) Standard IEEE 754 operand format. (b) Packet forwarding operand format.

Our *packet forwarding floating point operands* (see Fig. 3 (b)) have a non unique factorization

$$b = (-1)^s 2^e (f + c2^{-63}) \tag{2}$$

with the same sign and exponent format as a standard double extended precision operand. The significand factor $(f + c2^{-63})$ in (2) has the prenormalized range $1 \leq f + c2^{-63} \leq 4$ and is partitioned into two packets [5]:

- $\boldsymbol{f} = 1b_0.b_1b_2\cdots b_{62}$ with $b_i \in \{-1,0,1\}$, is a 64 digit borrow-save encoded digit string termed the *principal part packet.*
- $\boldsymbol{c} = c_{62}c_{63}$, termed the *carry-round packet*, is given by a two digit borrow-save digit string with $c = ||c_{62}c_{63}|| \in \{-2,-1,0,1,2\}$.

Herein, a k-digit borrow-save encoded digit string $\boldsymbol{a} = a_{k-1}a_{k-2}\cdots a_0$ denotes a $2 \times k$ bit array where the digit a_i is encoded by a positive bit $a_i^+ \in \{0,1\}$ and a negative bit $a_i^- \in \{0,1\}$ with digit value $a_i = a_i^+ - a_i^- \in \{-1,0,1\}$.

Definition 1 *The P-recoding of the k-bit borrow-save encoded digit string $\boldsymbol{a} = a_{k-1}a_{k-2}\cdots a_0$ denoted by $\boldsymbol{b} = P(\boldsymbol{a})$ is the $k+1$ digit borrow-save encoded digit string $\boldsymbol{b} = b_kb_{k-1}\cdots b_0$ defined by*

$$b_{i+1}^+ = a_i^+ \cdot not(a_i^-) \; for \; i = 1,\ldots,k \quad and \quad b_0^+ = 0$$
$$b_i^- = a_i^+ \oplus a_i^- \; for \; i = 0,\ldots,k-1 \quad and \quad b_k^- = 0$$

The N-recoding of $\boldsymbol{a}$, denoted by $\boldsymbol{c} = N(\boldsymbol{a})$, is the k-bit borrow-save encoded digit string $\boldsymbol{c} = c_kc_{k-1}\ldots c_0$, defined by

$$c_i^+ = a_i^+ \oplus a_i^- \; for \; i = 0,\ldots,k-1 \quad and \quad c_k^+ = 0$$
$$c_{i+1}^- = not(a_i^+) \cdot a_i^- \; for \; i = 1,\ldots,k \quad and \quad c_0^- = 0$$

The effect of a P-recoding is to selectively pass a positive carry out of each digit position and absorb it in the position one place higher. Similarly N-recodings selectively pass a negative carry (borrow) out of each position to be immediately absorbed. The reader can easily verify that the values denoted by the digit strings $\boldsymbol{a}$, $\boldsymbol{b} = P(\boldsymbol{a})$ and $\boldsymbol{c} = N(\boldsymbol{a})$ are all equal, i.e. that $\sum_0^{k-1} a_i 2^i = \sum_0^k b_i 2^i = \sum_0^k c_i 2^i$.

Given a radix polynomial $\sum_{i=\ell}^m d_i[2]^i$, the value obtained by "chopping" off leading digits and inserting a radix point immediately to the left of the "tail", is the *fraction value at position j*. It is denoted in terms of the digit string $d_md_{m-1}\cdots d_\ell$ 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 borrow-save encoded digit string $d_md_{m-1}\cdots d_\ell$ is said to have *fraction values in the range* (c,d) for $-1 \leq c \leq 0 \leq d \leq 1$ whenever $c \leq f_j(d_md_{m-1}\cdots d_\ell) \leq d$ for all $l+1 \leq j \leq m+1$. The *width* of (c,d) is $d-c$. Daumas and Matula [2] prove that if a borrow-save encoded digit string has fraction values in the range (c,d), then the P-carry recoding has fraction values in the range $(\frac{c}{2} - \frac{1}{2}, \frac{d}{2})$, and the N-carry recoding has fraction values in the range $(\frac{c}{2}, \frac{d}{2} + \frac{1}{2})$. The compound PN-recoding $P(N(\boldsymbol{d}))$ then

has fraction values in the range $(\frac{c}{4} - \frac{1}{2}, \frac{d}{4} + \frac{1}{4}) \subset (-\frac{3}{4}, \frac{1}{2})$. Successive recodings can reduce the fraction range of an internal borrow-save encoded digit string from a width of two to $1\frac{1}{2}, 1\frac{1}{4}, \ldots$. This reduction is termed partial compression.

The partial compression obtained by P and N-recoding can be extended to redundant adders as follows.

1. Consider a 3-2 adder as having for input a $3 \times k$ bit array where each of the k digits are encoded by 3 bits, two positive bits and one negative bit. Let the fraction range be (c, d) with $-1 \leq c < 0 \leq d \leq 2$. Then the output fraction range is $(\frac{c}{2} - \frac{1}{2}, \frac{d}{2})$.

2. Consider a 4-2 adder as having for inputs a $4 \times k$ bit array where each of the k digits is encoded by 4 bits, two of positive and two of negative weight. If the fraction range of the input digits is (c, d), with $-2 \leq c \leq 0 \leq d \leq 2$, then the output fraction range is $(\frac{c}{4} - \frac{1}{2}, \frac{d}{4} + \frac{1}{2})$.

3 Adder Datapath: Small Exponent Difference

In this section we describe the addition algorithm for the case that the exponent difference is small. As mentioned in the introduction , this case includes the case that extensive cancellation takes place. We have adjusted the exact range of differences to be $-1 \leq e_1 - e_2 \leq 4$. This adjustment turns out to simplify significantly the algorithm for the case of a large exponent difference at a modest increased cost for the case of a small exponent difference.

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, with f_2^+, f_2^-, c^+, c^- identifying the corresponding positive and negative bit strings of the borrow-save encoded digit strings f and c.

Description. Figure 4 depicts the addition algorithm operating under the assumption that $-1 \leq e_1 - e_2 \leq 4$. 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 exponent difference range from $[-1, +1]$ to $[-1, 4]$ makes the alignment shift somewhat costlier. However, the algorithm for the large exponent 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. Restricting the alignment shift to f_1 saves hardware.

While the significand f_1 is being aligned, the redundant significand is negated, if necessary, and recoded. Note,

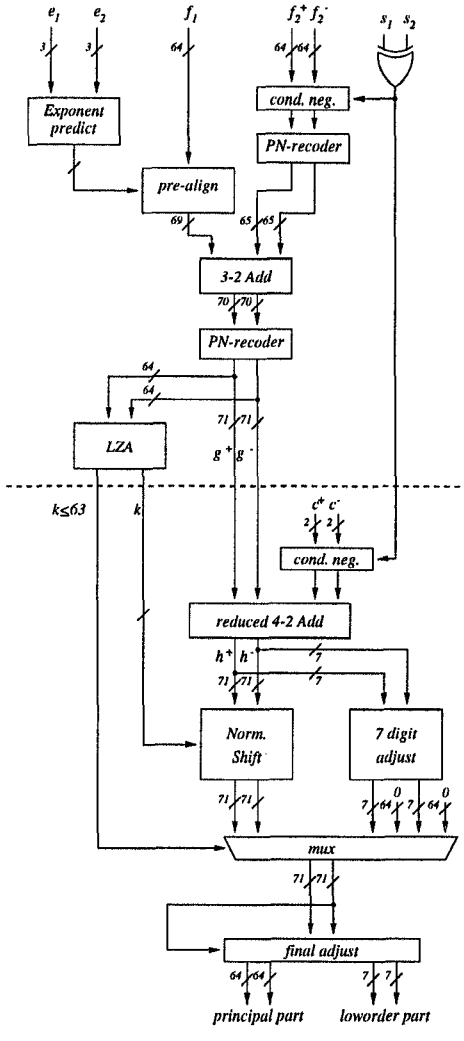

Figure 4. Adder datapath for $-1 \leq e_1 - e_2 \leq 4$.

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 sum, represented as a borrow-save number, is recoded and passed on to a leading zero estimator and through to the second cycle of the algorithm.

Since we are dealing with the case of small exponent difference, a non-zero sum may be very close to zero. However, the rounding unit requires that the output non zero significand belong to the range $(1, 4)$, and hence, a normalization shift may be required. 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 compression 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 recodings have a key role in restricting this adverse situation so that the shifted sum belongs to one of two possible binades, as formalized in Lemma 2. This completes the description of the first clock cycle.

In the second clock cycle, we first add the carry-round packet, denoted by c, to the sum, denoted by 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 significand is not shifted, the position of the carry-round packet is fixed. Hence, the redundant addition simply computes: $h = g + (-1)^{s_1 \oplus s_2} \cdot c \cdot 2^{-63}$. Claim 3 shows that this redundant addition can be performed by a redundant adder of 4 digits such that this addition does not generate a carry that changes the 64 most significant digits of 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 g; and "k" - the number of leading zeros in g in case there is a non-zero digit among the 64 most-significant digits of 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 the 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 4; and (b) adjust the two most significant digits of the sum so that the representation complies with the packet forwarding format.

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 [5]. Note, the description of the exponent and sign-bit data-paths is omitted.

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 2 *Suppose the sum g is non-zero and let $0^k \cdot \sigma \cdot t$ denote the representation of the recoded sum, where $\sigma \in \{-1, 1\}$ and $t \in \{-1, 0, 1\}^{p-k-1}$. Then $\sigma.t$ (where the dot between σ and t is a radix point) is in the range $(\frac{9}{32}, \frac{7}{8})$ for $\sigma = 1$ and $(-\frac{23}{16}, -\frac{9}{16})$ for $\sigma = -1$.*

Proof: Since the borrow-save number $f_2 = f_2^+ - f_2^-$, with fraction range $(-1, 1)$, is PN-recoded and added with a 3-2 adder to the binary number f_1 with fraction range $[0, 1)$, the output of the second PN-recoder $g = g^+ - g^-$ has fraction range $(-\frac{23}{32}, \frac{7}{16})$.

Denote the fraction range of the borrow save number $0^k \cdot \sigma \cdot t$ by (c, d). If $\sigma = 1$, then $0.1t \in (c, d)$, and thus, $1.t \in (1+c, 1+d) \cap (2c, 2d)$. Substituting $(c, d) = (-\frac{23}{32}, \frac{7}{16})$ we get $1.t \in (\frac{9}{32}, \frac{7}{8})$. Similarly we deduce $\bar{1}.t \in (-\frac{23}{16}, -\frac{9}{16})$. □

The sum 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. A more detailed look shows that the two least significant digits (i.e. $g[63 : 64]$) originate uninterrupted from the aligned non-redundant significand, and therefore, $g[63 : 64] \in \{0, 1\}^2$.

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

Claim 3 *Let $g[61 : 64]$ denote the 4 least-significant borrow-save digits of the redundant sum that are input to the second pipeline stage, and let $c[63 : 62]$ denote the 2 borrow-save digits of the carry round packet. Then,*

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

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

As stated above, the sum g has 7 digits to the left of the radix point and 64 digits to the right of the radix point. Let g be of the form $0^k \cdot \sigma \cdot t$, as in Lemma 2. The normalization shift shifts 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 $h = (g + c \cdot 2^{-63}) \cdot 2^{k-6}$. 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 4 *If $k \leq 63$, then normalized sum h satisfies:*

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

Proof: If $\sigma = 1$, then Lemma 2 implies that

$$h \in (\frac{9}{32}, \frac{7}{8}) + c \cdot 2^{k-69}$$

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

$$h \in (-\frac{23}{16}, -\frac{9}{16}) + c \cdot 2^{k-69}$$

Our assumption that $k \leq 63$ implies that

$$|c \cdot 2^{k-69}| \leq \frac{1}{32}.$$

and the claim follows. □

Claim 4 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.

4 Adder Datapath: Large Exponent Difference

In this section we describe 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. 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 alignment of the late arriving carry-round packet.

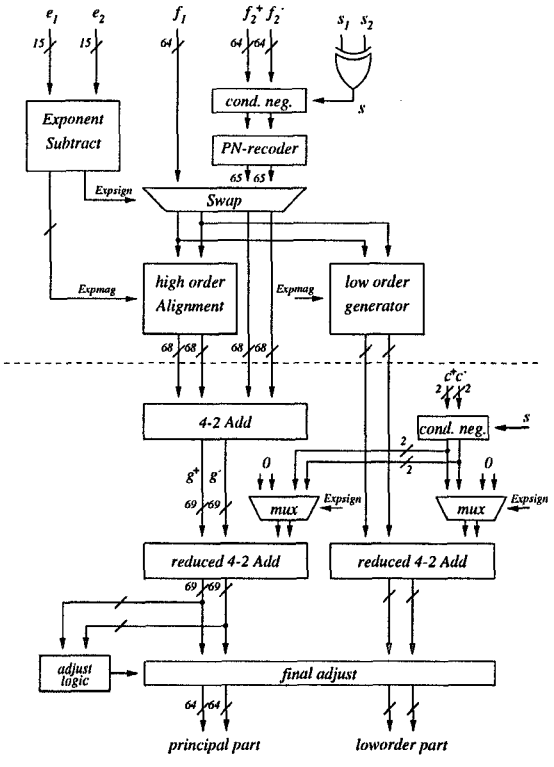

Figure 5. Adder datapath for $e_2 - e_1 \geq 2$ or $e_1 - e_2 \geq 5$.

Description. Figure 5 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. Bits shifted out on the right can be discarded as the low order generator computes the sticky digit. (b) The *low order generator* computes the bit string $0^{64-Expmag} . 1^{Expmag}$ and performs a bitwise AND between this string and the 64 most-significant digits of the significand (note that the PN-recoding introduces an additional digit to the left of the radix point). This pro-

duces 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. 6. This completes the description of the first clock cycle.

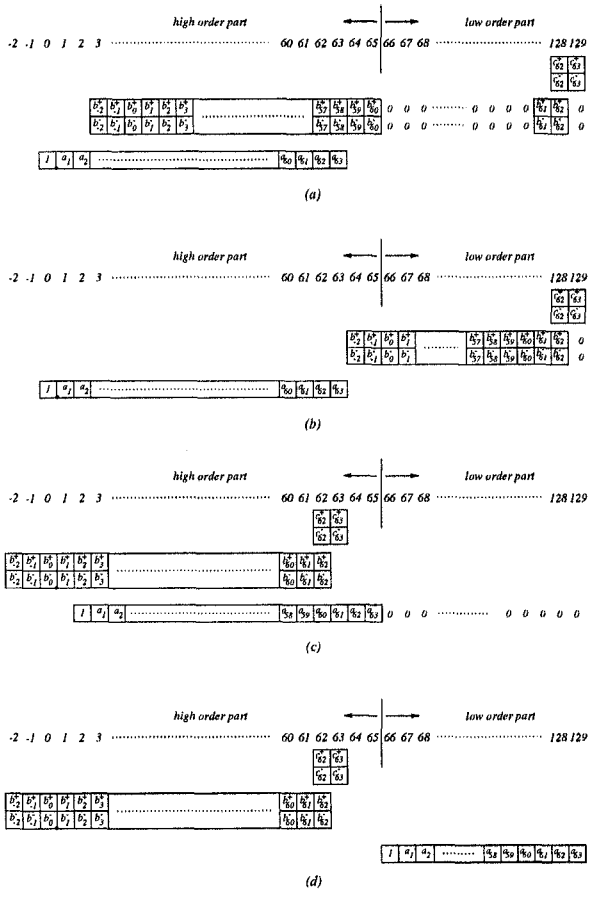

Figure 6. (a) alignment when $e_1 = e_2 + 5$; (b) alignment when $e_1 \geq e_2 + 66$; (c) alignment when $e_2 = e_1 + 2$; (d) alignment when $e_2 \geq e_1 + 66$;

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. 6. If $e_1 > e_2$, then then the redundant fraction $\boldsymbol{f_2}$ 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-round packet is added to the low order part, a ripple effect is not allowed. Claims 5 and 6 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, an adjustment takes place. The large exponent difference implies that the magnitude of the redundant sum is in the range $(\frac{1}{2}, 4\frac{1}{2})$. Therefore, the adjustment consists of: (a) sign detection and negation if necessary; (b) a shift by at most one position to the left or to the right so that the redundant sum is in the range $(1, 4)$; and (c) adjustment of the digits to the left of the radix point so that the representation complies with the packet forwarding format.

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 [5]. Note, that the description of the exponent and sign-bit data-paths is omitted.

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 5 *Let $c_{62}c_{63}$ denote the borrow-save digits of the carry-round packet. Consider the borrow-save significand $\boldsymbol{f_2'} = b'_{-2}b'_{-1}b'_0.b'_1b'_2 \cdots b'_{62}$ obtained by possibly negating the redundant significand $\boldsymbol{f_2}$ 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 6 *Let $c_{62}c_{63}$ denote the borrow-save digits of the carry-round packet. Consider the borrow-save number $g = g_{-3}g_{-2}g_{-1}g_0.g_1g_2 \cdots g_{65}$ computed by the 4-2 adder in Fig. 5. If the adder is implemented by cascading two 3-2 adders, 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

In this paper we have presented an addition algorithm according to the packet forwarding pipeline paradigm. The adder accepts one operand in the standard format and the second operand in a packet forwarding floating point format. 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 compression steps of the traditional floating point addition implementations. 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 [5], 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 compression 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 this procedure allows a sequence of additions to be fused without intermediate 2-1 additions. Only the final accumulation is subjected to a 2-1 addition for forwarding to a register. Furthermore, this accumulation agrees with the IEEE Standard sum where every intermediate result was appropriately rounded.

The algorithm employs recodings for obtaining partial compression of the redundant numbers [2]. The cost of each recoding is equivalent to that of passing every borrow-save digit through just 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 introduces a new threshold criteria for distinguishing between the cases of a small and a large exponent difference. This criteria 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 to one of two locations without variable shift delay and depending on only the sign of the exponent difference.

References

[1] "IEEE Standard for Binary Floating-Point Arithmetic"; ANSI/IEEE std 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] Farmwald, M. P.: "On the Design of High-Performance Digital Arithmetic Units"; PhD thesis, Stanford University, Aug. 1981.

[5] Matula, D. W., Nielsen, A. M.: "Pipelined Packet-Forwarding Floating Point: I. Foundations and a Rounder"; these Proceedings.

[6] Lyu, C.-N.: "Micro-Architecture of a Pipelined Floating-Point Execution Unit"; PhD thesis, SMU, Dallas, Texas, Dec. 1995.

[7] Quach, N. T., Flynn, M. J.: "An improved Floating Point Addition Algorithm"; Technical Report, Stanford University, Jun. 1990.