# Design of an On-Line IEEE Floating-Point Addition Unit for FPGAs

Steven D. Krueger and Peter-Michael Seidel

**Abstract**— We present the design of an on-line IEEE floating-point (FP) adder. In on-line arithmetic a result is computed as a digit serial output stream from digit serial input streams. The result digits begin to be produced a short delay after the first input digits arrive and before all the input digits have been received. On-line arithmetic proceeds from the most significant digit first through the least significant digit. Performing on-line addition on IEEE FP numbers imposes challenges beyond the challenges of conventional on-line arithmetic, including the task of normalization and IEEE rounding and its effect on the digits already output. The proposed implementations are very suitable for FPGAs, because the serial organization simplifies critical components of conventional FPGA IEEE FP addition implementations: Large alignment and normalization shifters are avoided allowing for reduced interconnection complexities and reasonable latencies at low implementation cost. The proposed implementations are fully compliant with the IEEE FP standard.

**Index Terms**—Floating-point addition, IEEE rounding, on-line arithmetic.

## 1 Introduction

The increasing use of application-specific datapaths in fixed and reconfigurable logic presents the challenge of accommodating a large number of interconnected functional units in the limited resources available. This makes considerations for cost-effectiveness a priority.

Floating-point (FP) arithmetic is considered very important in the choice of functional units on FPGAs. This and the need for compliance with the IEEE754-1985 [1] are underlined by the efforts to automate the translation from IEEE floating-point code to fixed-point code when FP hardware support does not exist [7].

The development of floating-point arithmetic implementations on FPGAs already has some tradition [5,9,10,15]. The increasing transistor budgets on FPGAs allow the consideration of alternative design options and make implementations feasible that were not practical in the early days. Current focus can be seen in parameterizable FP implementations [2,6,8] that allow for trading computation accuracy (precision) with implementation latency and cost and make corresponding implementation sources available.

Following our initial motivation for cost-effectiveness, we are also interested in enabling reduced implementation cost, but without reducing precision. For the importance of compliance with the IEEE FP standard, we focus on implementations that are fully compliant and that can support conventional IEEE precisions (single and/or double precision FP numbers).

Our approach for the design of cost-effective FP arithmetic implementations is the consideration of on-line arithmetic. In on-line arithmetic a result is computed as a digit serial output stream from digit serial input streams. Thereby, on-line arithmetic proceeds from the most significant digit first through the least significant digit and outputs the most significant result digit only a few cycles $\delta$ after the input of the most significant input digit. The value of $\delta$ defines the on-line delay of the implementation in cycles. On-line arithmetic modules for fixed-point arithmetic are smaller and can be clocked at higher frequencies than their parallel couterparts, but they have latencies related to the number of digits, and more significantly, have lower throughput than parallel functional units. For fixed-point numbers on-line arithmetic implementations are very cost-effective, because the hardware involved needs to process only a very few digits concurrently.

Although the motivation to apply on-line arithmetic to floating-point implementations is quite natural, there has been little consideration of on-line floating-

* *S.D. Krueger is with Texas Instruments, Inc., Dallas, TX 75243. E-mail: stevek@ti.com.*
* *P.-M. Seidel is with the Computer Science and Engineering Department of Southern Methodist University, Dallas, TX 75275. E-mail: seidel@acm.org.*

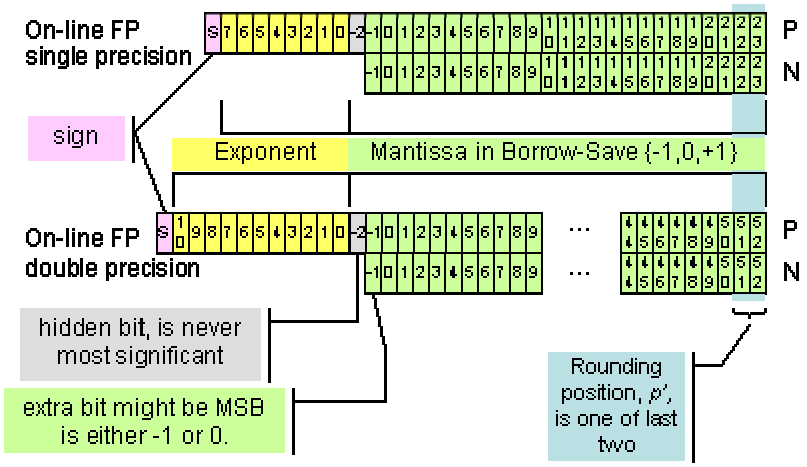


**Fig. 1. Proposed representation of IEEE single and double precision FP numbers.**

point arithmetic in the literature. The description in [16] presents a discussion of on-line computations for floating-point, but the computations considered are approximate with an error analysis described in [17,18]. This work was published before the IEEE FP standard even existed and the algorithms do not work for considering the on-line digit radix of 2. In [11,12] some suggestions about FP number representations for on-line arithmetic are given.

In on-line arithmetic the implementation of IEEE FP arithmetic imposes several major challenges over customized floating-point number representations (as in [16]). For on-line FP addition this is particularly the consideration of results that are simultaneously: a) normalized and b) correctly rounded, because rounding and normalization affect each other and need to be dealt with at different ends of the FP number representation.

For the implementation of on-line FP addition we propose a representation for IEEE FP numbers that allows an effective solution with respect to the challenge mentioned above. We propose radix-2 representations with non-redundant n bit representation of the exponent and with p+1 digit pseudo-normalized borrow-save representation of the significand. The borrow-save representation helps in absorbing carrys and borrows from lower order digits that have not yet been considered. The one additional digit in the significand allows the representation of floating-point numbers in pseudo-normalized form, where the significant has a value in the range [1,4).

To control rounding, we must detect when the sum is in the range [1,2) and when it is in the range [2,4). When the result value is in the former range, the rounding position is at the conventional place. When it is in the range [2,4), the rounding position is one more position to the left. Because carries and rounding can affect the final significand result the distinction between the two cases requires some further consideration to show the correctness of our implementation.

Finally, it should be noted that alignment shifting in a serial system does not require any expensive hardware, but just involves delays for one of the operands. Similarly, the normalization shift is also implemented as a delay. However, in normalization, the result's exponent can't be known until the most significant digit of the result is found. This needs to be done by inspecting the sum digits as they are generated. The delays implementing the normalization shift cause the implementation to have a variable latency for the case of gradual underflow.

## 2 Number Representation

We represent a floating-point number with a sign bit, followed by an n-bit non-redundant exponent representation, followed by a p+2 digit redundant borrow-save representation of the significand. The most significant digit is a hidden bit and does not appear in transmissions. The significand is pseudo-normalized and has the value range of [1,4).

For the redundant representation, we choose the borrow-save representation. This representation can be easily negated and compressed. We utilize known recoding schemes [3] to reduce the fraction range of the sum representation to slightly more than one binade, sufficient to simplify range testing and to allow feedback of results into inputs.

When a compressed IEEE floating-point number needs to be input, it can be converted by replacing the hidden one with a hidden one in our representation, followed by two -1 digits. The fractional part follows, where each binary digit is encoded as the corresponding borrow-save digit. Such a representation conver-

sion may be performed just by wiring without requiring any gates.

Figure 1 illustrates the proposed on-line representations for single and double precision IEEE FP numbers.

## 3 Notation

We use the notation developed in [14].

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

**IEEE FP-numbers.** In double precision, IEEE FP-numbers are represented by three fields (S, E[10:0], F[0:52]), with sign bit S ∈ {0, 1}, exponent string E[10:0] ∈ $\{0, 1\}^{11}$ and significand string F[0:52] ∈ $\{0, 1\}^{53}$. In a normalized IEEE number F[0] is always 1. The values of exponent and significand are defined by

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

Single precision IEEE floating-point numbers represented by (S, E[7:0], F[0:23]) where the values of the exponent and significand are defined by

$$e = \sum_{i=0}^{7} E[i]\cdot 2^i - 127 \qquad \& \qquad f = \sum_{i=0}^{23} F[i]\cdot 2^{-i}\,.$$

The value represented by an FP-number (S, E, F) is

$$fp\_val(S,E,F) = (-1)^S \cdot 2^e \cdot f$$

Normalized IEEE FP-numbers have significands in the range $f \in [1,2)$, however, our redundant representation will be pseudo-normalized and allow the significand to be in one of two binades of the range $f \in [1,4)$ representing the same values as normalized IEEE FP-numbers by adjusting the exponent correspondingly.

In a binary representation, a pseudo-normalized significand requires one more bit in F, which is F[-1]. The significand string F[-2:52] ∈ $\{-1, 0, 1\}^{55}$ is required in a redundant borrow-save encoding. In a pseudo-normalized representation, F[-2] is always 1 and never significant.

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

**Inputs.** The inputs of an FP addition/subtraction are:

1. Operands denoted by (SA, EA[10:0], FA[-2:53] and (SB, EB[10:0], FB[-2:53]).
2. Operation denoted by SOP ∈ {0, 1} to indicate addition/subtraction.
3. IEEE rounding mode.

**Output.** The output is an FP number (SSUM, ESUM[10:0], FSUM[-2:53]). The value represented by the output equals the IEEE rounded value of

$$fpsum = fp\_val(SA, EA[10:0], FA[-2:53]) + (-1)^{SOP} \cdot fp\_val(SB, EB[10:0], FB[-2:53])$$

**Serial transmission.** The floating-point numbers (SA, EA, FA), (SB, EB, FB) and (S, E, F) are transmitted serially into and out of the on-line IEEE floating-point adder. The transmission order is S, E[10], E[9], … E[0], F[-1], F[0], F[1], … F[53].

For a higher radix, $r = 2^k$, a radix-$2^k$ digit from E and F will be transmitted each transmission time. However, for simplicity, we assume that the transmission of S and the transmission of E do not overlap in time, nor do the transmission of E and the transmission of F. This restriction might be easily relaxed in an implementation.

## 4 Steps in IEEE FP Addition

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

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

Let SEFF = $sa \oplus sb \oplus SOP$. The case when SEFF =0 is called *effective addition* and the case that SEFF = 1 is called *effective subtraction.*

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

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

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

The sum can be written as:

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

The result is (SSUM, ESUM[10:0], FSUM[-2:53]) with fsum ∈ [1, 4). We need to compute ssum, esum and fsum. Since our goal is minimum delay, we will need to compute ssum and esum as soon as possible, but neither can be output until the most significant digits of fsum has been computed.

The sum is computed as follows:

1. Compute SEFF = $sa \oplus sb \oplus SOP$. These operands arrive first, so SEFF can be computed first.
2. Compute $\delta = ea - eb$. The exponent arrived MSD-first, so an on-line subtraction is used to compute δ as the digits arrive. This means that the difference is known shortly after EA[0] and

EB[0] are received.

It is critical to know which is the larger exponent and if the two exponents are equal. These two flags are computed as the exponent bits arrive so that these two decisions are known in the cycle when the EA[0] and EB[0] arrive. The sign of δ (which is larger) is used in step 3 below. If δ = 0, the alignment shift (step 8) is bypassed. If δ ≠ 0, then step 5 has at least one more cycle to complete.

3. The sign of δ is used to make the swapping decision to select (*sl, el, fl*) and (*ss, es, fs*), and to compute abs(*δ*).
4. Significand negation. The smaller significand is negated for effective subtraction. In addition, we must use signed arithmetic to comput *fsum*. In a signed digit representation, it is easy to invert the signs of each of the digits. In our borrow-save representation, the negation can be performed either by inverting the P and N bits of the *fs* operand, or by just exchanging the P and N bits.
   $fsn = (-1)^{SEFF} \cdot fs$.
5. Compute the limit of the alignment shift amount: $\delta\_lim = \min\{\alpha, \text{abs}(\delta)\}$, where $\alpha$ is a constant of the precision + rounding digits. For binary double precision, we will use $\alpha = 55$.
6. At this point we have a tentative sign and exponent for the result. *ssum.tent* = *sl*, *esum.tent* = *el*. These will be adjusted during the development of the *fsum* if necessary.
7. Let *i* = *j* = -2 and *zf* = 1.
8. Do the alignment shift. If For *c* from 0 to *δ_lim* do:
   a. FSUM[*j*++] = FL[*i*++].
   b. *zf* = 0. Have output significant digits because FL[-2] is always a 1.
9. For *i* continuing upto 53 do:
   a. DIG = FL[*i*] + FS[*i* - *δ_lim*].
   b. if *zf* = 1 and DIG = 0, let *esum.tent* = *esum.tent* – 1. This is the normalization shift.
   c. Otherwise:
      i. if *zf* = 1 and if DIG < 0, then *neg* = 1. This case detects a negative sum.
      ii. *zf* = 0. Now in significant digits.
      iii. FSUM[*j*++] = $(-1)^{neg} \cdot$ DIG.
   d. *last* if *j* > *49*.
10. If *i* = 53 and *j* < *50*, have not yet produced all digits of the fraction but have run out of FL digits. While (*j* < *50*) do:
    FSUM[j++] = FS[*i* - *δ_lim*].
    *i* = *i* + 1.
11. Now ready to produce the least significant three digits. The rounding position is either FSUM[52] or FSUM[51], depending on whether FSUM[-1] is significant or not after additional digits have been produced. Call the rounding position $p' \in \{51, 52\}$ for double precision. Inject the rounding constant according to the mode and rounding digits from FS[$p'$ - *δ_lim*:*p*]. It is not possible to perform a post-normalization shift as the exponent and higher order digits have been transmitted before the rounding is performed. Note that the rounding may be either positive or negative.

Step 8 performs the alignment shift by delaying the small operand by *δ_lim* cycles. Observe that for the implementation not more than (p+1)/2 digits need to be concurrently stored.

Note that the sign and exponent have not be finally decided until step 9ci. This step can change the sign of the result. Step 9b can change the exponent as this step performs normalization. Once the exponent and sign have been determined, they can be transmitted. This sets the minimum delay of the addition unit to the transmission time of the sign and exponent plus the transmission time of one digit plus 1. In a binary, double precision system, this on-line delay is 14 cycles, in single precision, this online delay is 11 cycles.

Larger delays can occur in the case of catastrophic cancellation. In this case a maximum normalization shift is required. The latter occurs when a *fsum* = 0.

## 5 On-line Significand Addition

The computation of signifcand addition in step 9 requires adding two borrow-save numbers. We are implementing this addition using on-line arithmetic

We require the 2:1 compression of two borrow-save numbers. In borrow-save representation such compression can be achieved with constant delay using a ppm- and a mmp- recoder [3]. Each of the two recoders can be implemented by a simple full adder with inverters at some inputs and/or outputs. The fraction range of the sum representation by two borrow-save numbers at the input is [-2, 2]. The 2:1 compression based on the combination of a ppm- and a mmp-module results in a borrow-save output representation with reduced fraction range. The on-line adder including delay elements is shown in Fig. 2. In this figure the top full adder is used to implement the ppm-recoder and the bottom full adder is used to implement the mmp-recoder.

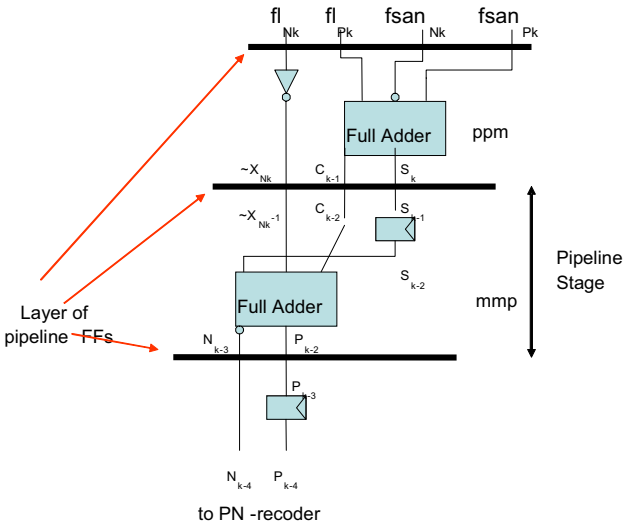


Fig. 2. On-line Borrow-Save Adder.

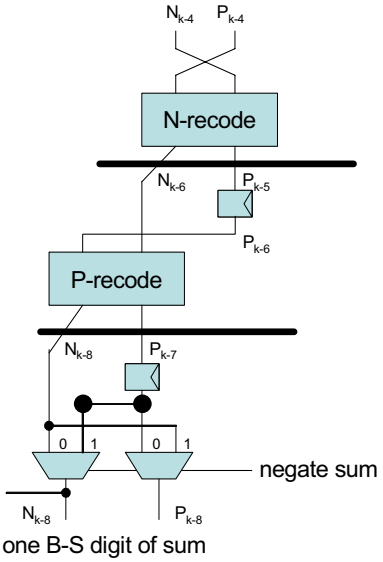


Fig. 3. On-line PN-Recoder. The latches delay earlier terms until the transfers arrive from less significant digits.

We follow this adder by an on-line adaptation of PN-recoding [3] to further limit the fraction range and to correctly determine the leading zeros and the sign of the result.

The P-carry recoding $B' = P(B)$ from borrow-save to borrow-save is given by:

$p'_{i+1} = p_i \cdot \overline{n}_i$ and $n'_i = p_i \oplus n_i$.

The N-carry recoding $B' = N(B)$ from borrow-save to borrow-save is given by:

$n'_{i+1} = \overline{p}_i \cdot n_i$ and $p'_i = p_i \oplus n_i$.

According to [3], after $PN^{\lambda}(B)$ recoding, the leading non-zero digit of a borrow-save representation indicates the sign of the number vor all non-zero values. Thus we can get an accurate sign for the significand by PN recoding of the sum and testing the N bit of the leading non-zero digit. The PN recoding network is shown in Fig. 3.

This recoding has the effect of limiting the fraction range to [-5/8, ½]. This smaller range is now small enough to allow accurate leading zero prediction, allow feedback of outputs to inputs, have an accurate sign from the leading digit, and limit the propagation of rounding.

## 6 ROUNDING

To determine the rounding position, we must determine if the value of *fsum* is in the range [1, 2) or [2, 4). The rounding position p′ is the same fractional bit as in compressed IEEE format if the range is [1, 2), but is the next more significant bit if the value is in [2, 4). To determine the position, we must interpret the digit stream to localize the value to determine if we have an underflow of FSUM[-1] or not. Table 1 contains a complete list of conditions that may occur and indicates which conditions correspond to the value underflow state. If there is a value underflow, $p' = p$, if not $p' = p - 1$.

Rounding is computed as follows:

$$frnd = \begin{cases} \langle fsum[-2:p'] \rangle + 2^{-p'} & \text{if } RINC \text{ and } rmode = RI \\ \langle fsum[-2:p'] \rangle - 2^{-p'} & \text{if RINC and rmode} \neq \text{RI} \\ \langle fsum[-2:p'] \rangle & \text{otherwise} \end{cases}$$

where

$$RINC = \begin{cases} (\sigma = -1) & \text{if } RZ \\ (\sigma = -1) \,\&\, (remain < 0) & \text{if } RNU \\ (\sigma = 1) & \text{if } RI \end{cases}$$

and $\sigma$ satisfies

$$tail = fsum[p'+1:t] = 0^k \cdot \sigma \cdot remain$$

Furthermore, the limited fraction range of the on-line sum limits the carry propagation so that the addition or subtraction of the rounding constant in postion $p'$ cannot affect digits beyond position $p'-1$.

Therefore, we will store the digits $p' - 1$ and $p'$ in the rounding unit as they are produced by the on-

TABLE 1
VALUE UNDERFLOW CONDITIONS

| *fsum*[-2:0] | Value Range | Next Non-Zero Digit | Leading Digit Under-flow |
|---|---|---|---|
| 1 -1 -1 | [1, 1½) | | Yes |
| 1 -1 0 | [1 3/8, 2) | -1 | Yes |
| 1 -1 0 | [2, 2½) | 1 or none | No |
| 1 -1 1 | [2 3/8, 3½) | | No |
| 1 0 -1 | [2 3/8, 3½) | | No |
| 1 0 0 | [3 3/8, 4) | -1 | No |

line adder. The RINC state machine will monitor the sum digits from p′ + 1 until σ is found or the end of the input is reached. Once RINC has been decided, the rounded digits can be computed by adding the rounding constant for the selected rounding mode, or left unchanged.

Note that in the case that *rmode* = RNE and *remain* = 0, the rounded value of bit $p'$ must be tested for being even (= 0). If the rounded value is not even, the unrounded value of $fsum[p'-1, p']$ is used. It is stored in the post-adder delay where it can be selected instead of the rounded value.

If $p' \neq p$, the output is one digit short for the proposed FP representation. We must insert this bit as a 0 to simulate the truncation after $p'$ that would have occurred in an IEEE floating-point result.

## 7 DESIGN OF ON-LINE IEEE FLOATING-POINT ADDER

The block diagram of the on-line floating-point adder is shown in Fig. 4. The sub-units on the left-hand side of the diagram operate using binary numbers while the sub-units on the right-hand side of the diagram operate on borrow-save digits. As input to the on-line FP adder conventional operands are allowed as well as operands in the proposed FP representation, because they can be easily converted.

The on-line adder operates as described in section 5. The rounding sub-unit operates as described in section 6. The overall behavior is described in section 4, explaining the processing in figure 4. Note that most units in the block diagram can be implemented in one or two logic blocks.

There are two delays. The first is the variable alignment delay. The second is the post-adder delay. This delay is fixed and just long enough to cover the transmission of the sign and exponent once *fsum* has been normalized and *fsum*[-2] has been located. Considering radix 2, we determine an on-line delay of our FP adder implementation of 14 cycles for double precision and of 11 cycles for single precision.

## 8 CONCLUSION AND FUTURE WORK

While there are a number of challenges to the design of an on-line IEEE floating-point adder, we have shown that it can be accomplished through the use of recoding techniques to limit the fraction range and through attention to the rounding details. The result is that a compact unit is realizable that will be beneficial to FPGA implementations and other implementations that are constrained in the numbers of gates and wiring channels available to implement parallel computational systems.

The implementation described is fully compliant with the IEEE standard, can be implemented for IEEE single and double precision and considers all four IEEE rounding modes while delivering a pseudo-normalized result, which means that the significand is in the range [1,4). We determine the on-line delay of our FP adder implementation to be 14 cycles for double precision and 11 cycles for single precision operands.

At this time our algorithm has been simulated using C-programs and perl-scripts. An FPGA implementation is currently in development. We expect to have accurate delay and cost measures available for FPGA implementations of the proposed algorithm by the time of the conference. This will allow for a detailed comparison with the FP adder implementations from [2,5,6,8,9,10,15].

By scaling the radix of the implementation the thoughput of the implementation could be traded off with higher implementation costs. After exploring the low-end cost for FP addition with the proposed implementation, scalable radix implementations could allow to fill in the design space in between, so that the implementation cost and performance could be more closely adapted to the performance needs and ressource budget of an application. Future work should explore such scalable implementations to evaluate the trade-off. We are also going to consider other on-line computational units for FP multiplication, FP division, FP square-root, each working on the proposed FP representations for IEEE values and each complying in their computation and rounding with the IEEE standard.

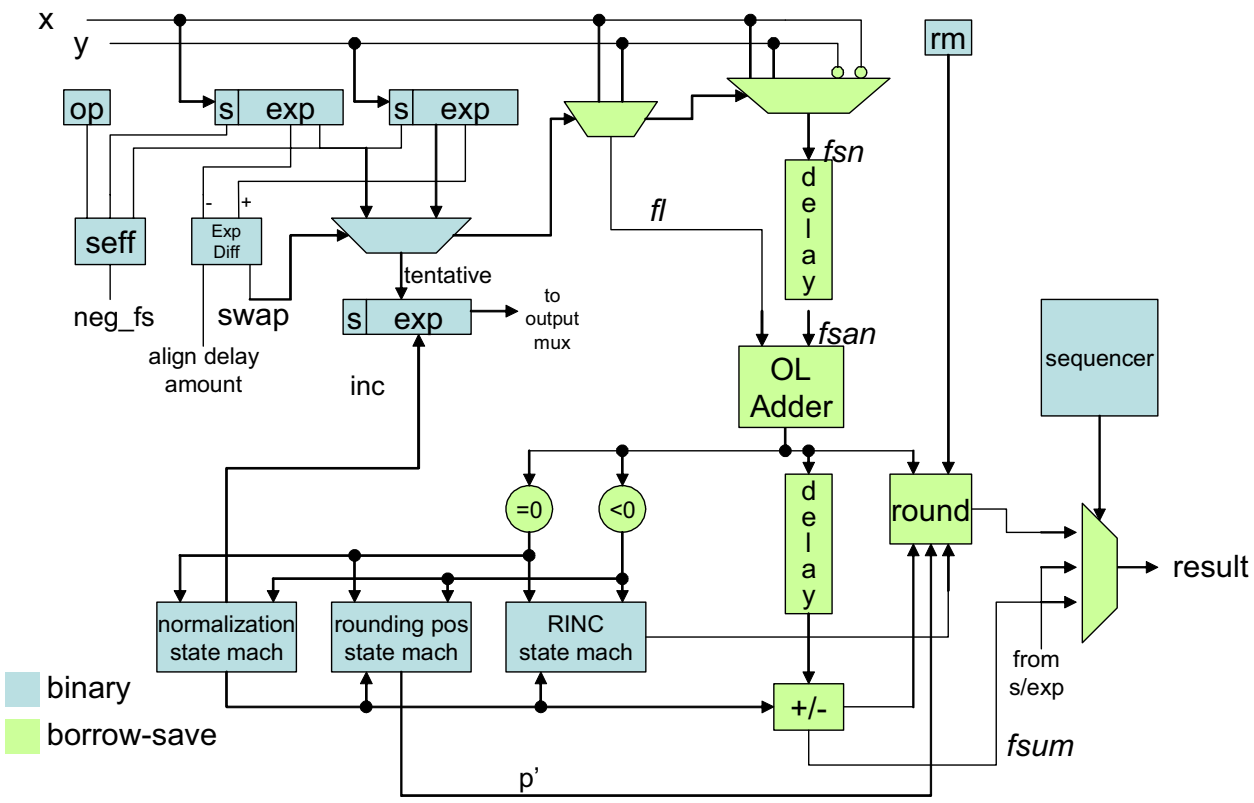


Fig. 4. Block Diagram of the On-line Floating-Point Addition Unit


## ACKNOWLEDGMENT

The authors wish to thank our families, Texas Instruments, Southern Methodist University and the students of SMU's CSE 8351 for their support.


## REFERENCES


[1] *IEEE Standard for Binary Floating-Point Arithmetic,* Institute for Electrical and Electronic Engineers, New York, NY, USA, 12 Aug. 1985.

[2] P. Belanovic, and M. Leeser, *A Library of Parameterized Floating-Point Modules and Their Use*. FPL 2002.

[3] M. Daumas, and D. Matula, *"Recoders fro Partial Compression and Rounding"*, Research Report 97-01, Ecole Normale Supérieure de Lyon, Lyon, France, Jan. 1997.

[4] M. Ercegovac, and T. Lang, Digital Arithmetic, *Morgan Kaufmann Publishers*, 2003.

[5] B. Fagin, and C. Renard, Field Programmable Gate Arrays and Floating-Point Arithmetic, *IEEE Transactions on VLSI Systems*, vol. 2, No. 3, Sept. 1994.

[6] Lee, and N. Burgess, *Parameterisable Floating-point Operations on FPGA*, Asilomar 2002.

[7] Leong et al, *Automating floating to fixed point translation and its application to post-rendering 3D warping*, FCCM 1999.

[8] Liang, Tessier et al, *Floating-Point Unit Generation and Evaluation for FPGAs*, FCCM 2003.

[9] Ligon et al, *A Re-evaluation of the practicality of floating-point operations on FPGAs*, FCCM 1998.

[10] Lourca et al, *Implementation of IEEE single precision floating-point addition and multiplication on FPGAs*, FCCM 1996.

[11] A. M. Nielsen, Digit-Serial Arithmetic, PhD thesis, *University of Southern Denmark*, 1997.

[12] A.M. Nielsen, and P. Kornerup, MSB-First Digit Serial Arithmetic, *Journal of Universal Computer Science*, Vol. 1 (7), 1995.

[13] B. Parhami, "Carry-Free Addition of Recoded Binary Signed-Digit Numbers", *IEEE Trans. Computers,* Vol. 37, No. 11, pp. 1470–1476, Nov. 1988.

[14] P.-M. Seidel, and G. Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition", *IEEE Trans. Computers,* Vol. 52, No. 12, Dec. 2003.

[15] Shirazi et al, *Quantitative analysis of floating-point arithmetic on FPGA based custom computing machines*, FCCM 1995.

[16] O. Watanuki and M.D. Ercegovac. *Floating-point on-line arithmetic: Algorithms.* Proc. 5th IEEE Symposium on Computer Arithmetic, pp. 81--86, 1981.

[17] O. Watanuki and M.D. Ercegovac. Floating-point on-line arithmetic: Error analysis. Proc. 5th IEEE Symposium on Computer Arithmetic, pp. 87--91, 1981.
[18] O. Watanuki and M. D. Ercegovac, *Error analysis of certain floating-point on-line algorithms*, IEEE Trans. Comput., C-32(4):352--358, April 1983.