Efficient Floating-Point Representation for Balanced Codes for FPGA Devices (preprint version)

Abstract—**We propose a floating–point representation to deal efficiently with arithmetic operations in codes with a balanced number of additions and multiplications for FPGA devices. The variable shift operation is very slow in these devices. We propose a format that reduces the variable shifter penalty. It is based on a radix–64 representation such that the number of the possible shifts is considerably reduced. Thus, the execution time of the floating–point addition is highly optimized when it is performed in an FPGA device, which compensates for the multiplication penalty when a high radix is used, as experimental results have shown. Consequently, the main problem of previous specific high-radix FPGA designs (no speedup for codes with a balanced number of multiplications and additions) is overcome with our proposal. The inherent architecture supporting the new format works with greater bit precision than the corresponding single precision (SP) IEEE–754 standard.**

Keywords—Floating Point representation; FPGA devices; variable shifts; high radix arithmetic;

I. Introduction

The use of FPGAs to accelerate high-performance computing applications has been increasing rapidly in recent years. These applications are generally implemented utilizing standard floating-point number representation, specifically IEEE–754 standard [1][2]. Consequently, research is currently underway to develop functional units and libraries which provide an efficient implementation of IEEE-754-compliant operations on FPGAs [3][4][5].

The IEEE–754 standard was defined in relation to a general purpose processor implemented using ASIC technologies. A broad overview of various implementations and their optimizations is given in reference [6]. Thus, some requirements of this standard may not match the typical characteristics of FPGA devices. A more efficient implementation of floating-point arithmetic for FPGAs could be achieved if the architecture was defined taking into account the resources available in FPGA devices [7], [8], [9], [10], [11], [12].

In this study, we attempted to a find a floating-point format which fits some of the characteristics of FPGA devices and provides better results for actual implementations for codes with balanced number of multiplications and additions. Given the importance of having a standard, our aim was not to replace it but to complement it when an FPGA is used as an accelerator. In these kinds of applications, data being processed is more likely to stay on chip until the application has finished processing it [13].

The aim of our proposal is to define a format which allows us to convert the IEEE–754 floating-point input numbers to the new internal format, operate with this new representation inside the FPGA accelerator (which has more precision bits than IEEE) and return the output numbers in IEEE–754 format, while keeping accuracy (note that the IEEE specification does not require identical output from all IEEE compliant operators).

Floating-point addition involves two variable shifters and a leading zero detector (for alignment and normalization), which are very slow when they are implemented in FPGAs. If the maximum number of digits to be shifted is reduced, the penalty due to this operation also decreases. This idea was used in [7] which investigated high radix floating–point representations for FPGA-specific devices. The authors found that the best performance was achieved by the radix–16 implementation for FPGA devices, obtaining an improvement in execution time of about 7% for additions and a loss of about 6% for multiplications. Thus, in codes with a balanced number of additions and multiplications (i.e. digital filters) the gain is negligible. This is a serious handicap since the most frequent operation in digital signal processing (DSP) is filtering.

We overcome the problem of having balanced code, obtaining a clear net gain. To achieve this, we design an efficient format conversion modules (IEEE to/from internal representation) which allows us to use a carefully selected higher radix (64 versus 16) which fits the hardware and considerably improves on previous results (68% for additions and 23% for balanced codes). The global behavior presents clear advantages in execution time, unlike the previous specific FPGA designs. We provide a detailed study and proof of the accuracy of the proposed format, which fulfills the requirements of IEEE-754 single precision standard.

Thus, the main contribution of our paper is that the problem of the lack of speedup of balanced codes in previous specific high-radix FPGA designs has been overcome. This is due to the design of an efficient format conversion circuit which allows us to deal with a higher radix, unlike previous implementations.

II. Floating-Point Format Proposed

Without any loss of generality, we focus on an initial format similar to the IEEE–754 single-precision floating-point representation. The representation of an IEEE–754 floating-point number x is

$$x = (-1)^{S_x} M_x 2^{E_x} \tag{1}$$

where $S_x \in \{0,1\}$ is the sign, M_x is the magnitude of the significand (24 significant bits, also called mantissa) and E_x is the exponent (8 bits). The significand is a radix–2 normalized number with one integer bit $M_x = 1.F$ (F is the fraction).

We propose using a radix–64 for the representation of a floating-point number instead of radix–2.

Let us define the representation of a floating-point number x as follows:

$$x = (-1)^{S'_x} M'_x 64^{E'_x} \quad (2)$$

where S'_x is the sign bit, M'_x is the magnitude of the significand and E'_x is the exponent (6-bit width, base 2), which has a bias of 32. The significand is a radix–64 normalized number composed of five 6-bit digits with one integer digit and four fractionals such that $1 \leq M'_x < 64$. Thus,

$$M'_x = D0.D1D2D3D4 \quad (3)$$

where D0 through D5 are digits of 6 bits (Di represents the digit of relative weight 64^{-i}). Basically, this means that the integer part of the significand is the MSD whereas the fractional part is composed of the remaining digits. For example, the mantissa $110.110011010010011010111$ has five digits: the MSD (D0) is 000110 and the rest are $110011(D1), 010010(D2), 011010(D3)$ and 111000 (D4, LSD) (6 bits per digit). The dynamic range is $[1 \cdot 2^{-192}, (64 - 2^{-24}) \cdot 2^{186}]$, which is wider than that of the IEEE-754.

The five digits used and the proposed normalization ensure that the minimum number of significant bits is 25, which corresponds to an integer part equal to one ($1 \leq \lfloor M' \rfloor < 64$). In fact, the number of significant bits ranges from 25 to 30, depending on the magnitude of the integer part of M'. Thus, the proposed format always has a number of bits of precision greater than that of IEEE 754.

A. *Format conversion*

In this subsection we analyze the conversion of a normalized single precision IEEE–754 number (equation (1)) to the new format indictaed by equation (2). Three parameters have to be obtained (S'_x, M'_x, E'_x) from those of the IEEE–754 representation of a number (S_x, M_x, E_x). The sign is trivial $(S'_x = S_x)$. The exponent is obtained as

$$E'_x = \left\lfloor \frac{E_x - 127}{6} \right\rfloor - sign(E_x - 127) + 32 \quad (4)$$

where $sign$ is the sign function ($sign(a) = 0$ if $a \geq 0$ and $sign(a) = 1$ if $a < 0$). To obtain the new mantissa M'_x from M_x we need to align it. The new mantissa requires the following operation:

$$M'_x = M_x 2^{(E_x - 127)\ mod*\ 6} \quad (5)$$

where mod* is an operator such as

$$A\ mod*\ B = \begin{cases} A\ mod\ B & if\ A \geq 0 \\ -(|A|\ mod\ B) + B & if\ A < 0 \end{cases} \quad (6)$$

Figure 1 shows a possible architecture to convert a single-precision IEEE-744 floating-point number to the proposed format. Due to the fact that our final device is an FPGA, the conversion is aided by means of a small look–up table. The input is the exponent of the IEEE–754 number E_x (8 bits, 256 inputs) and the outputs are the new exponent E'_x (6-bits width) and the number of bits to be right shifted for the mantissa (the exponent of equation (5), 3-bits width). A binary variable shifter is required for this operation.

On the other hand, figure 2 shows the block diagram to convert a number from our internal format to the IEEE–754 format. The sign has a direct conversion.

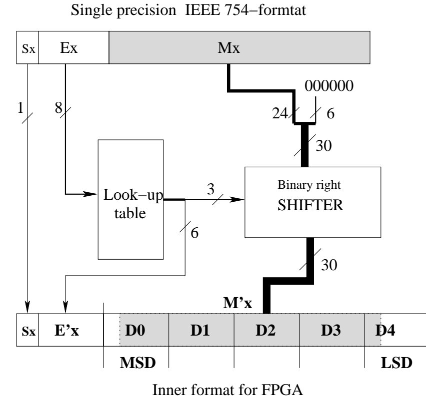

Fig. 1. Format conversion

To obtain the significand, the module "Leading One Detector" in figure 2 is in charge of obtaining the number of leading zeros of the MSD (we call this number k). A right shift of 5-k bits is required for the significand of 30 bits, which results in a number with 24 bits. This is done by the shifter called "2^{5-k}" in figure 2.

Rounding is required since there may be some shifted–out bits in the LSD. The module "Rounding" is in charge of obtaining the value 1 or 0 to be added to the shifted significand. This requires analyzing the LSD as well as the value $5 - k$ (given by the Leading One Detector module).

The exponent is obtained by

$$E_x = 6(E'_x - 32) + 127 + 5 - k \quad (7)$$

We can use a small 6-input bits (64 words) look-up table, which gives the value $6(E'_x - 32) + 127$, as shown in figure 2. If $E_x > 255$ or $E_x < 0$, we have a special value ($+\infty$ and $-\infty$, respectively). Special cases are not shown in the figure for simplicity (but are actually implemented, see section V).

Finally, if an overflow is produced due to rounding, the significand would be of the form 10.000...0. The exponent is updated by connecting the overflow signal, as shown in figure 2.

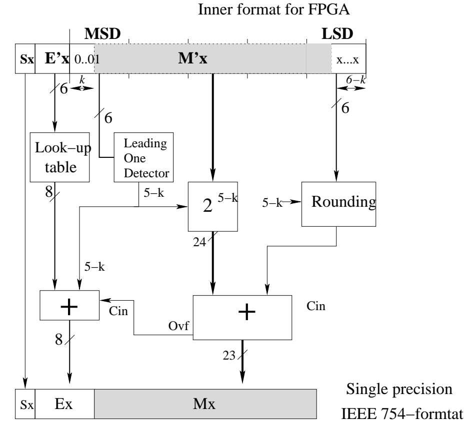

Fig. 2. Format conversion

III. ADDITION WITH THE NEW FORMAT

The addition (subtraction) of two standard IEEE floating-point numbers x and y, which are represented by (S_x, E_x, M_x) and (S_y, E_y, M_y), respectively, is done by means of the following six steps:

1) Subtraction of exponents. Let x and y be two floating-point numbers according to the format of expression (2). We analyze all the possible cases:
- $|E'_x - E'_y| > 4$. The sticky bit of the second operand (y') has to be computed for rounding operations as well as a borrow in the subtraction. This is shown in figure 3.c.
- $0 < |E_x - E_y| \leq 4$. The addition is carried out in the next step by aligning the significand. The sticky bit of the shifted–out digits is required for further operations (see figure 3.b).
- $|E_x - E_y| = 0$. The significands are aligned and the sticky bit is required only in case of an overflow (see figure 3.a).

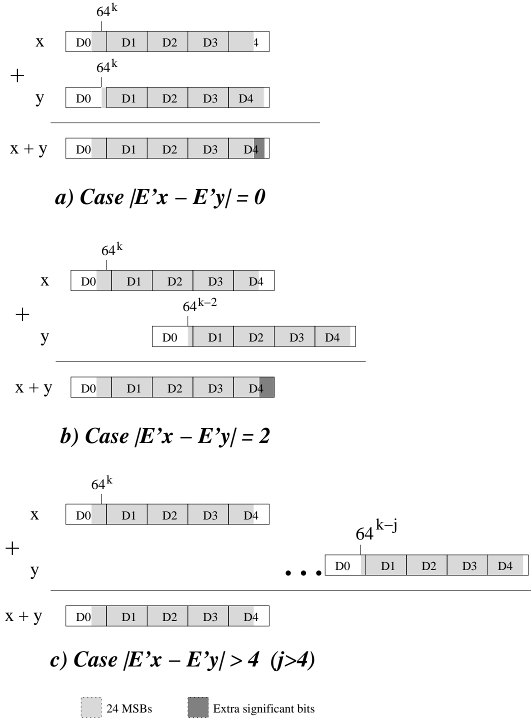

Fig. 3. Addition of two numbers

2) Align significands: we have to shift right $d = |E_x - E_y|$ positions (digits) the significand of the operand with the smallest exponent and select the largest exponent as the exponent of the result. Note that $|E_x - E_y| = \{0, 1, 2, 3, 4\}$ is a reduced set of shifts compared to the standard case ($\{0, 1, ..., 23\}$). Figure 3 shows all the possible shifts. The sticky bit of the shifted–out digit has to be calculated.

3) Add (subtract) significands. The sign of the result depends on the sign of the operands and the relative magnitude of the operands. Five digits are involved in each operation.

4) Normalization of the result. Due to the characteristics of the proposed format, a non-normalized result is only possible under overflow of the significand (addition) or if the integer part of the result is zero (substraction). Thus,
- A leading-one detector is not needed, but an overflow and leading digit zero detector is needed.
- If the result of the addition of the significands is greater than or equal to 64 there is an overflow. The normalization is carried out by setting the MSD to 1 and performing a right shift of one digit (D0.D1D2D3D4 ovf $\rightarrow$ 1.D0D1D2D3 plus sticky bit).

Similarly, if the result of a subtraction of the significands is less than 1, the MSD of the result is 0 (integer part). The normalization requires a left shift of up to four digits depending on the number of leading zero digits. For example, if there are two leading zero digits we perform a left shift of two digits ($\hat{0}.\hat{0}D2D3D4 \rightarrow D2.D3D4\hat{0}\hat{0}$, where $\hat{0}$ means an all-zero digit).
- The exponent of the result is updated by adding 1 if there is an overflow in the addition. Regarding substraction, it is updated by subtracting as many units as the number of leading zero digits of the result.

5) Rounding. The aim of our proposal is to maintain precision greater than or equal to that of IEEE–754 (24 significant bits) for all internal FPGA operations. Thus, after any rounding we must have a numerical value with a precision greater than or equal to the precision obtained after a IEEE–754 rounding. We consider rounding to nearest, to even when tie since it is the default mode for the IEEE and its implementation is the most complex.

Let us decompose the final significand of the result of one operation considering all the digits M' involved:

$$M' = M'^{(24)} + M'^{(d)} 2^{-24} \tag{8}$$

where $M'^{(24)}$ is composed of the integer part of M' plus the 24 most significant fractional bits, and $0 \leq M'^{(d)} < 1$. Namely, $M'^{(24)}$ has the precision of the significand in our floating-point system and $M'^{(d)}$ represents the rest. For example, if we use the symbol "$\wedge$" to indicate a 6-bit digit and we obtain the result of an operation like $\hat{1}.\hat{2}\hat{3}\hat{4}\hat{5}\hat{6}\hat{7}$, the value of $M'^{(24)} = \hat{1}.\hat{2}\hat{3}\hat{4}\hat{5}$ and $M'^{(d)} = \hat{6}\hat{7}$.

Let G denote the bit with weight 2^{-24} (that is, the LSB of the LSD). We define the rounding to nearest of the significand $r(M')$ as follows:

$$r(M') = \begin{cases} M'^{(24)} + G \cdot 2^{-24} & if\ M'^{(d)} \neq 0 \\ M'^{(24)} & if\ M'^{(d)} = 0 \end{cases} \tag{9}$$

In comparison with the corresponding IEEE 754 round to nearest, the final rounded value of our approach always has a precision greater than or equal to that of IEEE since IEEE defines a significand of 23 fractional bits, whereas we have 24 fractional bits. In other words, the round to nearest mode (not tie) is quite similar to that of the IEEE, but uses 24 fractional bits instead of 23.

In case of a tie, the IEEE propose the tie to even solution. In our case, since we have at least 25 significant bits instead of 24 fixed significant bits, the middle value $M'^{(d)} = 0$ can be covered by the bit G. For example, for IEEE 754, the value 1.100 1000 1110 1001 1001 100x 10000000 is rounded to even so that if bit x=0 the rounded value is 1.100 1000 1110 1001 1001 1000 (24 bits, error=-0.0...010000000=-2^{-24}) and if x=1 the rounded value is 1.100 1000 1110 1001 1001 1010 (24 bits, error=+0.0...010000000=$+2^{-24}$). Nevertheless, according

to equation (9), our rounded value is 1.100 1000 1110 1001 1001 100x 1 (25 bits, error=0), which is the exact value. Thus, like IEEE 754, the proposed solution is unbiased (recall that our aim is to maintain precision greater than or equal to that of the rounded IEEE 754).

Let us address the subtraction operation. Since our format has 5 digits (30 bits), in most cases the result of a subtraction has more than 24 significant bits and rounding can be carried out using the G bit, as proposed in equation (9).

Nevertheless, there are some extreme cases requiring some extra guard bits. Having analyzed all these cases, we conclude that two extra bits are required for rounding: a guard bit and the sticky bit (see [14] for details). This conclusion is in line with that of the IEEE in which three guard bits are required: in our case, since our minimum size for the significand is 25 bits, one of these three bits belongs to the significand (the G bit). Thus, the path for the actual implementation of the addition/subtraction operation is not 30 bits but 32 bits and rounding is carried out based on the final result by applying equation 9.

6) Determine exception flags and special values. We consider the same cases as the IEEE (implementation details are presented in section V).

A. Comparison of rounding

In this subsection we analyze the rounding for our format and compare it with the corresponding case of the IEEE.

In general, since our format has a precision of between 25 and 30 bits (due to normalization) and the single precision IEEE has 24 bits, our rounding is performed using more significant bits and has a greater precision.

We now analyze the worst case in our proposal, which takes place when the result of an operation has 25 significant bits. Taking into account that we add two extra bits in the path for rounding purposes, the result of any addition/subtraction is 32-bits wide (five 6-bit digits plus two guard bits). Consider that the result of an operation is

```
  D0   .  D1      D2      D3      D4
000001.xxxxxx xxxxxx xxxxxx xxxxLG RT
```

where L and G corresponds to the two LSBs of the digit D4, R is a guard bit and T is the sticky bit. Note that the weights of the bits L,G,R and T are $2^{-23}, 2^{-24}, 2^{-25}$ and 2^{-26}, respectively. The final rounded value for the IEEE reaches the bit of weight 2^{-23}, whereas the final rounded value in our format has a weight of 2^{-24}.

Table I presents a detailed comparison of the 16 different cases of the bits L,G,R,T and the corresponding rounded values. The LSB of the rounded value for the IEEE has a weight of 2^{-23} (24 bits, second column), whereas the weight of the LSB of our result is 2^{-24} (25 bits, one more precision bit, fourth column). Thus, the bits of weight 2^{-24} through 2^{-26} are discarded after rounding for the IEEE format whereas the bits 2^{-25} and 2^{-26} are discarded in our case.

The third column shows the difference between the IEEE rounded value (24 bits) and the pre-rounded value (32 bits). Similarly, the fifth column gives the difference between our rounded value (25 bits) and the pre-rounded value (32 bits). We can see that the difference is exactly the same for both roundings except for the case NGRT=0100 and NGRT=1100 (boldface in table I). These cases correspond to the tie cases for IEEE in such a way that a round to even is carried out. It generates a bias which is compensated in the IEEE case (difference +4 and -4). Nevertheless, in our case we do not need to round to even since we have one more precision bit and we take the exact value with no bias (difference = 0). In other words, the tie case of the IEEE does not need any rounding in our case since we use the exact value due to fact that we have one more precision bit.

We also present the error regarding the actual value (which is given by the bits LGRT)

LSBs (32 bits) LGRT	IEEE rounding (24 bits) $2^{-23} 2^{-24}$	diff. $(\times 2^{-26})$	Our rounding (25 bits) $2^{-23} 2^{-24}$	diff. $(\times 2^{-26})$
0000	0 -	0	0 0	0
0001	0 -	-1	0 0	-1
0010	0 -	-2	0 0	-2
0011	0 -	-3	0 0	-3
0100	0 -	**-4**	0 1	**0**
0101	1 -	+3	1 0	+3
0110	1 -	+2	1 0	+2
0111	1 -	+1	1 0	+1
1000	1 -	0	1 0	0
1001	1 -	-1	1 0	-1
1010	1 -	-2	1 0	-2
1011	1 -	-3	1 0	-3
1100	←0 -	**+4**	1 1	**0**
1101	←0 -	+3	←0 0	+3
1110	←0 -	+2	←0 0	+2
1111	←0 -	+1	←0 0	+1

← Carry propagation

TABLE I. ROUNDING FROM THE FOUR LSBS OF THE RESULT (NGRT)

Since we can perform intermediate calculations with more precision bits, the output to be returned might have a numerical value different from that obtained if all the intermediate calculations were carried out with 24-bit precision. This is not a problem since the IEEE specification does not require identical output from all IEEE-compliant operators [1], [2].

We developed a C simulation program to check our algorithm. We used a random pattern of about $3 \cdot 10^{11}$ calculations, 81% with exactly the same precision as the IEEE 754 and 19% with enhanced precision.

B. Architecture

Figure 4 shows the architecture proposed for a single–path implementation (the results can be extended to other alternative architectures like double-path). First, the exponents are compared to place the largest number on the left operand and the smallest one on the right (module ”swap”).

The difference of exponents (d) is used to shift the operand y' (module 64^{-d}). In a substraction the module ”sticky1” is in charge of obtaining the two extra guard bits required in a substraction (bit R and the sticky bit T). Thus, the first adder has inputs of 32 bits.

On the other hand, after the first adder, a right shift of one digit is required in case of overflow (only for the addition operation, module called ”64^{-1}” in figure 4). The extra bits

required (due to the shift-out of the LSD) are obtained by the module "sticky2" and generates the bits R and T. In case of subtraction, the bits R and T do not need to be re-computed. The module "sticky2" controls the final value of R and T depending on the operation.

The "rounding calculus" module performs the computation of a 0 or 1 according to equation (9). In this equation the value of $M'(d)$ corresponds to the bits R and T (a logic OR). The adder carries out the final rounding. If an overflow is produced after rounding the result is 1000000.000...000. This requires updating the exponent and forces the value $\hat{1}.\hat{0}\hat{0}\hat{0}\hat{0}$ as the final result.

After a substraction it is possible to have several leading zero digits. The module "Digit leading zero detector" is in charge of this operation and the corresponding digit left shift is performed in the module "64^{-lzd}" (ldz is the number of digits to be shifted). If all the digits of the result are zero, it is thus a special value (the corresponding hardware is not shown in the figure for clarity).

Finally, the exponent has to be updated. The module entitled "Update Exp." is in charge of adding 1 if an overflow is produced or subtracting the value "lzd" in case of having a left shift.

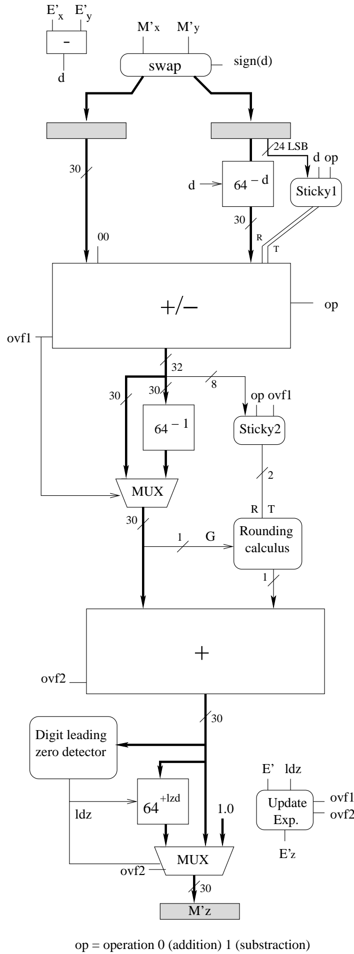

Fig. 4. Architecture for addition/subtraction

IV. MULTIPLICATION

This work focuses on the addition operation, and the proposed architectures are tailored to optimize the execution time in an FPGA device. Floating point multiplication does not involve shift operations. The improvement achieved for addition (due to the improvement in the shift operation) does not speed up multiplication; in fact it slows it down, as shown in [7]. This is due to larger word length, the larger multiplier, and longer execution time.

On the other hand, the most extended operation in DSP is filtering, which involves multiplications and additions. In [15] the authors propose a high radix double-precision format based on two 53-bit digits, in such a way that the size of the significand is 106 bits instead of 53. In this way, the shifts are reduced to only one position (to left and to right), which improves the speed of the addition considerably. Nevertheless, there are important implications for the multiplication due to having such large digits since the size of the multiplier increases by a factor 4, which increases the area and time of this operation, and it becomes impractical for digital filtering.

Due to the fact that many applications use multiplication and addition, we consider multiplication with the new format. In our case, the significand has 30 bits instead of 24.

Let x' and y' be the operands of two floating-point numbers using our format, which are represented by (S'_x, E'_x, M'_x) and (S'_y, E'_y, M'_y), respectively. The result $z' = x' \times y'$ is represented by (S'_z, E'_z, M'_z). The resulting sign is $S'_z = s'_x \oplus s'_y$, the exponent is $E'_z = E'_x + E'_y$ and the new significand is $M'z = M'_x \times M'_y$, where a 30×30 unsigned multiplier is required (five digits each operand).

Let us address the normalization. The result of the multiplication has 10 digits and we have to select the five most significant non-zero digits. Let us call the ten digits of the result $D0, D1, ..., D9$ ($D0$ is the most significant). Since M'_x and M'_y are both greater than or equal to one (that is, the MSD of both operands is greater than or equal to one), the normalized significand of the result is

$$M'_z = \begin{cases} (D0.D1D2D3D4) & if \ D0 \neq 0 \\ (D1.D2D3D4D5) & if \ D0 = 0 \end{cases} \tag{10}$$

If $D0 \neq 0$ the exponent has to be updated $E'_z = E'_z - 1$.

Regarding the rounding, the same solution as the addition operation is used since the normalized result has at least 25 significant bits (see equation (9)). In this case, the value of $M'^{(d)}$ (the sticky bit) has to be computed from either D5 through D9 or D6 through D9 (see equation 10).

V. IMPLEMENTATION RESULTS AND COMPARISON

To measure the effectiveness of the new floating-point format presented in this paper, we have developed some VHDL modules implementing the proposed architectures to perform the main operations related to it for single precision. These are addition/subtraction, multiplication and conversions. For comparison purposes, the corresponding modules generated by the open source library FloPoCo [5] have been used as representative of the floating-point standard core for FPGAs. Since these circuits are easily pipelined, for the sake of simplicity

all the designs are compared using the fully combinational version. All these modules were simulated using Modelsim SE 6.3f and they were synthesized using Xilinx ISE 13.4, targeting a Virtex6 device (xc6vlx240t-1). TableII summarizes the results obtained for area and delay.

TABLE II. IMPLEMENTATION RESULTS

	Delay (ns)	Area (LUTs/mult)
FloPoco radix-2 Adder	21.234	344/-
Our radix-64 Adder	12.659	282/-
FloPoCo radix-2 Multiplier	9.989	60/2
Our radix-64 Multiplier	12.613	55/4
IEEE→Ours	2.355	98/-
Ours→IEEE	5.750	104/-

According to [7], a high radix speeds up addition and slows down multiplication. Thus, a trade–off solution has to be found since the faster the adder, the slower the multiplier. The trade–off solution proposed in this paper is a radix-64 since it provides considerably speed-ups in balanced codes (codes with a similar number of multiplications and additions). We now present the implementation results and compare them to the standard radix–2 implementation and with the radix–16 implementation presented in [7].

i) Comparison with radix–2 implementation. In our case, as expected, there is a strong improvement in the adder unit (68% faster with 18% less area), whereas there is a degradation in the multiplier (20% slower and doubling the embedded multipliers; for applications involving multiplications and additions, the slowdown of the multiplication is satisfactorily compensated by the high speedup of the addition, as shown later). The cost of format conversion is not excessive, since they use about a third of the area utilized by an adder and the delay is a little less than the improvement achieved using the proposed adder. In fact, the proposed adders including both conversions has roughly the same performance as a radix–2 one.

Thus, the benefits of using the proposed format depend on the target application. Specifically, it mainly depends on the ratio between the number of additions and the number of multiplications and to a certain extent on the total number of internal floating-point operations involved in the algorithm (if the number of internal operations is large, the conversion time is negligible). Thus, many applications can take advantage of the proposed representation. Some important basic algorithms in DSP applications, such as summation and digital filters, can take advantage of our representation. In the case of summation, the improvement is close to 68% and for the case of digital filters (in which the number of additions and multiplications is similar) the improvement is close to 23%, as can be calculated using the data shown in Table II.

Due to the importance and widespread use of digital filters in DSP applications, we provide a simple example of a pipeline design, as shown in figure 5.

The multiplier and adder proposed in our implementation have a very similar delay (12.659 ns. and 12.613 ns.). This facilitates the design of pipeline architectures (like figure 5), where the stages are well-balanced. Nevertheless, the corresponding counterpart radix–2 units are not well-balanced (see table II). Since the clock cycle is imposed by the slowest unit (adder), the throughput of our pipelined implementation

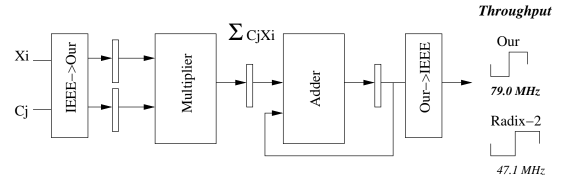

Fig. 5. Example of application: simple pipelined architecture for digital filtering

is about 79.0 MHz, whereas the counterpart radix-2 pipeline implementation is 47.1 MHz, which improves throughput by about 67%.

ii) Comparison with radix-16. In [7] the authors implement a radix-16 adder and multiplier using a Xilinx Virtex-II 6000 (4-LUT-based). Since we use a different technology (6-LUT-based), a direct comparison would not be appropriate. Therefore, we compare the improvements obtained for each design regarding the radix–2 implementation. Taking into account that there is a net gain in the addition operation and a net loss in the multiplication operation for both implementations, the global gain/loss depends on the ratio of additions/multiplications of each application. In figure 6 we present the speedup of both designs as a function of the percentage of addition/multiplications in the code:

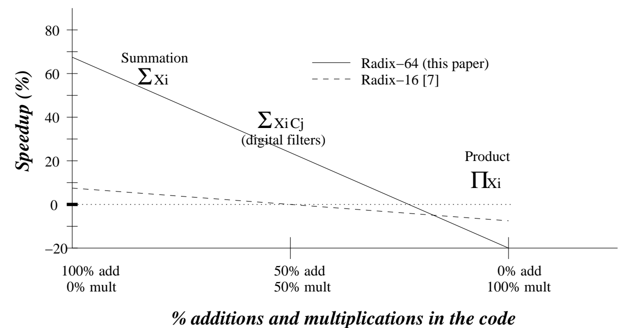

Fig. 6. Speedup as a function of % additions-multiplications

This figure shows that the speedup decreases as the proportion of multiplications increase for both designs. We can see that in applications where additions are dominant, both designs have net gains. For applications with a balanced number of additions/multiplications, our design maintains large speedups (23%), whereas no significant speedup is obtained for [7]. Finally, in applications where multiplication is highly dominant, neither of the solutions is valid. We can see a positive speedup if the percentage of additions is greater than 50% for [7] and negative speedup beyond 50%. In our case, we achieve moving this critical point to only 25%. The speedup interval is (7%,-6%) for [7] and (68%,-20%) for our case. The high speedup obtained for the addition is in line with with the fact that the set of possible shifts of our design is less than that of [7].

We can see that the high speedup of the addition in our case compensates the negative speedup of the multiplication, such that in a code with a mixture of multiplication and addition (which is the most usual case) we have a net gain. This is

exactly the main handicap of [7]: if the code has a balanced number of multiplications and additions, the gain is only about 1%, whereas our proposal has a net gain of 23%. Table III show the scores obtained by both implementations for some basic and extended algorithms such as summation and digital filtering:

TABLE III. SPEEDUP FOR COMMON ALGORITHMS

	Summation	Digital filter
Radix 16 [7]	7%	1%
Our radix 64	68%	23%

The area of the conversion format circuit is about 30% of that required for the adder (see Table II). In [7] this ratio is about 9%. This is due to the fact that they propose and implement radixes which are a power of a power of 2 (that is 2^{2^k}) which facilitates exponent conversion. The results of implementing a radix 4 (k=1) and 16 (k=2) are presented in [7] but not 256 (k=3), since it yields worse results than radix-16 and the multiplier was extremely costly.

Thank to the conversion format circuits proposed in this paper, it is possible to design circuits with a radix beyond radix–16. The efficient format conversion circuits proposed in section II can be easily adapted to for any radix. In our approach, these format converters allows us to use a radix-64, which is not a power of power of 2 (2^{2^k}). Our implementation results show that the use of an intermediate radix like 64 achieves a clear speedup in the addition time (1.68) compared to radix–16 (1.07). Above all, the balanced codes achieve a clear speedup (1.23 versus 1.01)), which justifies the increase in the area of the converters. On the other hand, as in the case of [7], there are applications involving many inputs/outputs and little computation in which the high radix approach is not suitable since it carries out many conversions.

In conclusion, our design leads to a huge speed–up (68%) in additions with a similar reduction in the area compared to that presented in [7]. This high speed compensates for the multiplication penalty when a radix-64 is used in balanced codes, as experimental results have shown. Thus, we have clearly overcome the main problem involved in previous designs (that is, a negligible speedup in the case of codes with a similar number of additions and multiplications).

VI. CONCLUSION AND FUTURE WORK

We have proposed a floating–point format based on radix–64 representation to speed up the arithmetical computations for FPGA–based applications. We have focused on the addition operation in such a way that the penalty of variable shifters is considerably reduced due to the proposed high radix. The architectures proposed in this work allow us to convert the IEEE–754 floating-point input numbers to the new format, operate with this new representation inside the FPGA accelerator (at high speed), and return the output numbers in IEEE–754 format, while maintaining at least the same accuracy as the IEEE–754 standard.

Thank to the conversion format circuits proposed in this paper, it is possible to design circuits with a radix beyond radix–16. The trade–off solution proposed in this paper (radix-64) overcomes the main problem of previous high radix designs (no speedup for codes with a balanced number of multiplications and additions). In our proposal, a speedup of 23% is achieved. This widens the range of applications for which our approach is valid, in such a way that some important DSP algorithms, such as digital filters, can benefit from the proposed proposal, unlike previous high radix designs.

Based on the results of these experiments, future work could include studying higher precisions (binary 64 and 128) and decimal arithmetic, as well as other operations such as division and transcendental functions. Optimization of the multiplier (by a lower level design) is also considered in the future to reduce the area. We also consider other different FPGA families to study their behavior with the proposed format.

REFERENCES

[1] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[2] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 29 2008.

[3] *Xilinx Corporation*. http://www.xilinx.com/tools/coregen.htm, 2012.

[4] *Altera Corporation*. http://www.altera.com/, 2012.

[5] F. Dinechin and al., *FloPoCo, Floating Point Cores*. http://flopoco.gforge.inria.fr/, 2007.

[6] P.-M. Seidel and G. Even, "Delay-optimized implementation of IEEE floating-point addition," *Computers, IEEE Transactions on*, vol. 53, no. 2, pp. 97 – 113, feb 2004.

[7] B. Catanzaro and B. Nelson, "Higher radix floating-point representations for fpga-based arithmetic," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, april 2005, pp. 161 – 170.

[8] P. Karlstrom, A. Ehliar, and D. Liu, "High performance, low latency fpga based floating point adder and multiplier units in a virtex 4," in *Norchip Conference, 2006. 24th*, nov. 2006, pp. 31 –34.

[9] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. Wilton, "Floating-point fpga: Architecture and modeling," *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on*, vol. 17, no. 12, pp. 1709 –1718, dec. 2009.

[10] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An fpga-specific approach to floating-point accumulation and sum-of-products," in *ICECE Technology, 2008. FPT 2008. Int. Conf. on*, 2008, pp. 33–40.

[11] I. Ligon, W.B., S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood, "A re-evaluation of the practicality of floating-point operations on fpgas," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 1998, pp. 206–215.

[12] S. Sun and J. Zambreno, "A floating-point accumulator for fpga-based high performance computing applications," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, 2009, pp. 493–499.

[13] P. Underwood, "Trends in peak floating-point performance," in *ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA04)*, 2004.

[14] J. Villalba and J. Hormigo, "Apendix to paper efficient floating-point representation for balanced codes for fpga devices http://www.ac.uma.es/∼julio/apendix_iccd2013.pdf."

[15] P.-M. Seidel, "High-radix implementation of ieee floating-point addition," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, june 2005, pp. 99 – 106.