Microprocessors and Microsystems 35 (2011) 535–546

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Customizing floating-point units for FPGAs: Area-performance-standard trade-offs

Pedro Echeverría *, Marisa López-Vallejo

Departamento de Ingeniería Electrónica, Universidad Politécnica de Madrid, Spain

ARTICLE INFO

Article history:
Available online 7 May 2011

Keywords:
Floating-point arithmetic
FPGAs
Library of operators
High performance

ABSTRACT

The high integration density of current nanometer technologies allows the implementation of complex floating-point applications in a single FPGA. In this work the intrinsic complexity of floating-point operators is addressed targeting configurable devices and making design decisions providing the most suitable performance-standard compliance trade-offs. A set of floating-point libraries composed of adder/subtracter, multiplier, divisor, square root, exponential, logarithm and power function are presented. Each library has been designed taking into account special characteristics of current FPGAs, and with this purpose we have adapted the IEEE floating-point standard (software-oriented) to a custom FPGA-oriented format. Extended experimental results validate the design decisions made and prove the usefulness of reducing the format complexity.

1. Introduction

Current deep sub-micron technologies allow manufacturing of FPGAs with extraordinary logic density and speed. The initial challenges related to FPGAs programmability and large interconnection capacitances (poor performance, low logic density and high power dissipation) have been overcome while providing attractive low cost and flexibility [1].

Subsequently, use of FPGAs in the implementation of complex applications is increasingly common but relatively new when dealing with floating-point applications ranging from scientific computing to financial or physics simulations [2–4]. This is a field of increasing research activity due to the performance and efficiency that FPGAs can achieve. The peak FPGA floating-point performance is growing significantly faster than the CPU counterpart [5] while their energy efficiency outperforms CPUs or GPUs [6]. Additionally, FPGA flexibility and inherent fine-grain parallelism make them ideal candidates for hardware acceleration improving GPUs capabilities for a particular set of problems with complex datapaths or control and data inter-dependencies [7]. FPGAs flexibility also allows the use of tailored precision, what can significantly improve certain applications. Furthermore, new FPGA architectures have embedded resources which can simplify the implementation of floating-point operators.

However, the large and deeply pipelined floating-point units require careful design to take advantage of the specific FPGA features. Designing this kind of application from scratch is almost impossible or makes the design cycle extremely long. Thus, the availability of complete and fully characterized floating-point libraries targeting FPGAs has become a must.

The IEEE standard for binary floating-point arithmetic was conceived to be implemented through custom VLSI units developed for microprocessors. However, if the target hardware is an FPGA, the internal architecture of these operators must be highly optimized to take advantage of the FPGA architecture [8]. Furthermore, implementations with slight deviations from the standard could be of great interest, since many applications can afore some accuracy reduction [3,9], given the important savings that can be achieved: reduced hardware resources and increased performance.

Several approaches have addressed the hardware implementation of a set of floating-point operators [10–12], but none of them includes the wide and general analysis carried out here. Some works only include the basic operators (adder/subtracter, multiplier, divider and square root) [10]. Other works focus on the implementation of particular floating-point operator implementations [11–13]. Regarding advanced operators (exponential, logarithm and power functions) few works can be found, standing out [14–16], with implementations of the exponential and logarithm functions.

In [8], the potential of FPGAs for floating-point implementations is exploited focusing on the use of internal fixed formats and error analysis what requires specific analysis for every application and supporting tools. Therefore, floating-point operators are mainly replicated in hardware without tailoring the format to the applications. In this scenario is where we have focused on, improving the performance of the floating-point units taking advantage of FPGA flexibility. We have tuned the architecture of floating-point operators to get the best performance-cost trade-off with slight deviations from the

* Corresponding author.
E-mail addresses: petxebe@die.upm.es (P. Echeverría), marisa@die.upm.es (M. López-Vallejo).

0141-9331/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2011.04.004

standard. This approach was also discussed in [17] but it is extended here in several ways:

- We have included advanced operators.
- A more complete set of deviations is studied.
- We perform an in-depth analysis of the implications of the deviations.
- We study the replicability of the operators.
- We provide a set of recommendations to achieve the resolution and accuracy of the standard with high performance.

Our proposed libraries include both conventional and advanced operators. Starting by an almost fully-compliant library[1] (`Std`), we have made several design decisions that allow clear improvements in terms of area and performance. These design decisions include the substitution of denormalized numbers by zero, the use of truncation rounding or the definition of specific hardware flags that allow the use of extended bit width internally.

The interest of this work is focused on the impact of those decisions over floating-point operators and not in presenting new architectures. Thus, the main contributions of this work are the following:

- A thorough analysis on the implications of the proposed design decisions has been carried out focusing on the performance-accuracy trade-offs. This is the base of a set of recommendations that can be considered when a complex floating-point application is implemented in configurable hardware.
- A complete set of varying accuracy floating-point libraries has been developed including both conventional (adder/subtracter, multiplier, divider and square root) and advanced (exponential, logarithm and power functions) operators.
- A systematic approach based on specific interfaces has been adopted allowing the use of extended bit widths. It simplifies the implementation of complex applications and reduces the resources needed for chained operators fitting in a single FPGA with better performance.
- Two final FPGA oriented libraries with significant hardware improvements have been implemented, taking advantage of the proposed design decisions.

The paper structure is as follows: Section 2 summarizes the floating-point format. Section 3 presents the key design decisions that are proposed while Section 4 describes the particular architecture of each operator. Experimental results are thoroughly discussed in Section 5, paying special attention to the influence of the proposed design decisions. Finally, Section 6 introduces a hardware library specially designed for FPGAs and Section 7 draws some conclusions.

2. Floating-point format IEEE 754

The IEEE Standard [18] is mainly designed for software architectures, usually using 32 bit words (single precision) or 64 bit words (double precision). Each word (Fig. 1) is composed of a sign (s, 1 bit), a mantissa (mnt, m_b bits) and an exponent (exp, e_b bits), being the value of a number:

$$s \times mnt' \times 2^{exp'} = s \times h \cdot mnt \times 2^{exp-bias} \quad (1)$$

where h is an implicit bit known as the hidden bit and the *bias* is a constant that depends on e_b being its value $2^{e_b-1}-1$. With this number representation the floating-point format can represent

[1] Only some software issues as exception handing with additional flags and signaling NaNs (not a number) are not implemented.

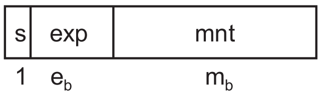


Fig. 1. Floating-point word.

zeros, infinities, exceptions (Not a Number, NaN) and two number types, normal ones (normalized) and numbers very close to zero (denormalized). The differentiation among these five types is based on the exponent and mantissa values. Table 1 depicts all possible combinations of exponent and mantissa values.

The standard is specifically designed to handle these five number types sharing a common format while maximizing the total set of numbers that are represented. Combining these two facts increases the complexity of the arithmetic units because, in addition to the calculation unit itself, it is needed a preprocessing (also known as prenormalization) of the inputs numbers and a postprocessing (also known as postnormalization) of the output numbers, see Fig. 2.

Therefore, when implementing a floating-point operator, the hardware required is not only devoted to the calculation unit itself, additional logic is needed just to handle the complexity of the format. This logic represents a significant fraction of the area of a floating-point unit, a 48% of logic in average for the studied operators, as will be shown in Section 5. In a general way preprocessing logic includes:

- Analysis of the number type of the inputs, which includes exponent and mantissa analysis.
- Determination of operation exceptions due to the number type or sign of the inputs (square root, logarithm).
- Normalization of inputs.
- Conversion of inputs to the format required by the calculation unit.

Table 1
Types of floating-point numbers.

Type	Exponent	Mantissa	h	value
Zero	0	0	–	±0
Denormalized	0	≠0	0	Eq. (1)
Normalized	1 to $2^{e_b}-2$	–	1	Eq. (1)
Infinities	2^{e_b-1}	0	–	±∞
NaN	2^{e_b-1}	≠0	–	–

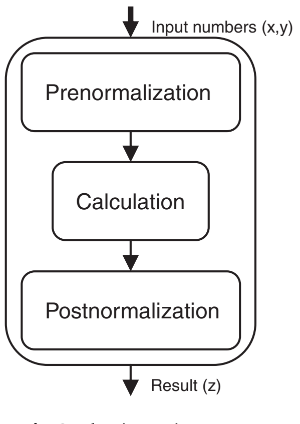


Fig. 2. Floating-point operator.

In the same way, postprocessing logic includes:

- Rounding of the result to adjust it to the format precision (mantissa and exponent correction).
- Determination of result exceptions.
- Format of the output to fit the result number type.

3. Floating-point units for FPGAs: Adapting the format and standard compliance

The implementation of a floating-point application in hardware using FPGAs requires the adaptation of the software-oriented floating-point format to this particular context to obtain the best performance. This can be accomplished by simplifying the complexity of the format so its associated processing is reduced. Following this idea, we have developed four different floating-point custom formats where three key design decisions have been gradually introduced:

1. Simplification of denormalized numbers.
2. Limitation of rounding to just truncation towards zero.
3. Introduction of specific hardware representation for the number type.

These decisions affect three features of the format that are responsible of most of the preprocessing and postprocessing overhead: handling denormalized numbers (1), rounding (2) and handling the five number types with a common format (3). Next, the implications of these decisions are studied.

3.1. Simplification of denormalized numbers

The use of denormalized numbers is responsible of most logic needed during preprocessing and postprocessing. However, most floating-point arithmetic algorithms work with a normalized mantissa (leading bit equals 1) in the calculation unit. Thus, denormalized numbers have to be converted to normalized ones during preprocessing. This requires the detection of leading one bit of the mantissa, left shift of the mantissa and an adjustment of the exponent value depending on the position of the leading one.

During postprocessing the result of the calculation unit is analyzed to determine the number type it corresponds to and to make some adjustments to its exponent and mantissa. Again, most logic related to these two tasks is required for handling the case of results that are denormalized numbers.

Consequently, the use of denormalized numbers requires of significant resources, negatively affecting the performance of the arithmetic units as usually the slowest paths are related to the handling of that denormalized numbers. However their use does not contribute substantially to most applications because denormalized numbers (i.e. single precision $e_b = 8$ and $m_b = 23$):

- Represent a small and infrequent part of the format: most of the floating-point format is reserved for normalized numbers being $2^{m_b} \times (2^{e_b} - 2)$ different normalized numbers to only 2^{m_b} denormalized ones (i.e. $\rightarrow 2^{23} \times 254$ normalized to 2^{23} denormalized for single precision). Furthermore, their value, $< 2^{e_b} - 2$ (i.e. $\rightarrow < 2^{-126}$), also makes their use very infrequent except for some special applications.
- Compromise the accuracy of results: while normalized numbers have a precision of $m_b + 1$ bits the denormalized precision varies from m_b bits to 1 depending on its leading one position. This compromises the accuracy of the result of any operation where denormalized numbers are involved.

Therefore, one way to simplify the format and reduce logic is handling denormalized numbers as zeros so all the related logic is eliminated. Actually, all commercial FPGA floating-point libraries [19,20] follow this scheme and previous works as [17,12] also deviate this way from the standard.

Deviation from the Standard. The standard cost of this solution is related to resolution and accuracy. First, when denormalized numbers are replaced by zeros we are losing resolution around the zero, as now 2^{m_b} (i.e. $\rightarrow 2^{23}$) numbers with values between $2^{-(2^{e_b-1}-2)} - 2^{-(2^{e_b-1}-2+m_b)}$ and $2^{-(2^{e_b-1}-2+m_b)}$ have been removed (i.e. $\rightarrow 2^{-126} - 2^{-149}$ and 2^{-149}).

Second, we are losing accuracy because we operate with zeros at the inputs or obtain a zero result when we have denormalized input or output. However, this loss of accuracy is relative as the resolution of a denormalized number depends on the position of its leading 1. Due to the lack of resolution or due to a rounding from a previous operation, the maximum relative error a denormalized number can reach is even the 100% of its value,[2] being much bigger than the maximum relative error for a normalized number, 2^{-m_b}.

3.2. Truncation rounding

The exact result of any operation can need more mantissa bits than the bits provided by the format. In this case the result needs to be rounded to fix with the format as its value will be comprised between two consecutive representable floating-point numbers. In the IEEE standard we can find four different rounding methods:

- Nearest: rounding to the nearest value.
- Up: towards $+\infty$.
- Down: towards $-\infty$.
- Zero: towards 0.

To implement these methods the result is generated with additional mantissa bits. Then, in the postprocessing stage, the extra bits of the result (guard bit when necessary, round bit and sticky bit) and the sign of the result (for rounding methods towards up and down) are analyzed to carry out the rounding. Rounding to nearest, provides the most accurate rounding, as ensures a maximum error of $\frac{1}{2}$ *ulp* (unit in the last place, that is the least significant mantissa bit, *lsb*). The other three methods have a maximum error of 1 *ulp*.

Meanwhile, rounding to zero is the method that needs less logic because it is equivalent to truncating the result without taking into account the round bit and the sticky bit. This feature is useful for hardware floating-point units, specially for units that require of iterative algorithms like division or square root. If only rounding towards zero is implemented, these units can be reduced as they do not need to generate the round bit and the sticky bit, consuming less cycles and resources and, in case of pipelined architectures, less stages.[3]

Deviation from the Standard. Truncation rounding slightly affects the accuracy of a single operation, and this only happens if we compare with round to nearest, which ensures a rounding error of up to half *ulp* while truncation rounding ensures up to 1 *ulp*. If we just take into account one operation, this small loss of accuracy can be considered negligible as we are in the range of a relative error of 2^{-m_b}.

However, for applications with a large number of chained operators, the error propagation must be considered, as the errors

[2] When the leading 1 of the mantissa corresponds to the *lsb* and rounding mode is up or down.

[3] Always for algorithms computing one bit per cycle, and depending on the precision and number of bits obtained for other algorithms.

introduced in the first operations spread along the chain while the following operators also introduce error in the partial results. In these cases the error in the final result can be higher when using truncation rounding instead of round to nearest.

Additionally, with truncation rounding the results are biased as truncation always rounds to the same direction, diminishing the absolute value of the result for each operation.

3.3. Hardware representation

In any operation using a floating-point number the first task is to analyze the values of the exponent and the mantissa of the operands to determine the number type. Meanwhile, the last task of the operation is to compose the exponent and mantissa of the result taking into account the number type of the result.

These two tasks are necessary to make compatible the software architecture requirements of a fixed word length and the use of the floating-point standard implying different number types.

However, in an FPGA architecture the word length used is flexible and configurable by the designer and can be extended with some flags to indicate the number type. To represent the five number types three flag bits would be necessary. However, if the previous mentioned simplification of denormalized numbers is applied, two flag bits are enough.

This scheme follows the internal use of flags presented in previous works [14,15] of the FPLibrary [21] that we have extended with a systematic approach using interface blocks. Two interface blocks are required: first, the input interface that calculates the value of the flags from a standard floating-point number; second, the output interface that composes a standard floating-point number from our custom floating-point number and its corresponding flags.

The two interfaces carry out some of the preprocessing and postprocessing tasks so the logic needed in the arithmetic unit can be reduced. When a datapath involves chained operations, the advantages of this scheme are clear as the input interface is only needed for each input number while the output interface is found only at the final result. All intermediate operations need no interfaces so all the operators involved have been reduced.

Deviation from the Standard. This design decision has no cost in terms of resolution or accuracy. Meanwhile, the use of interfaces makes transparent the transformation between the standard format and the extended FPGA format.

3.4. Global approach analysis

The three design decisions just explained insist on the same point, the simplification of the floating-point format complexity. Handling complexity requires logic resources, thus as the complexity is reduced so are the logic resources needed for an operator. Additionally, as less resources are used, operators implementations become more efficient, and speed or the number of pipeline stages is improved. Therefore our approach has focused on determining those features of floating-point arithmetic that require a heavy processing overhead while their relevance in the format is minimum.

However, while the use of dedicated flags does not have any effect on the standard compliance, the other design decisions affect compliance in terms of accuracy and resolution around zero.

Finally, and regarding floating-point standard compliance, one last issue should be addressed when using FPGAs: the non-associativity of floating-point operations. FPGA datapaths are commonly designed taking advantage of FPGAs intrinsic parallelism placing parallel operators in the datapath. However the results obtained with parallel operations may differ from the ones obtained with an equivalent sequential implementation [22]. Therefore, two issues affect standard compliance: how the floating-point operators are implemented and how the datapath is designed. The second issue is architecturally dependent and cannot be analyzed in a general way, being out of the scope of this work.

4. Arithmetic units

To study the impact of the three design decisions on floating-point operators we have implemented a set of libraries where those decisions have been gradually applied, from standard operators to operators including the three simplifications. Table 2 reviews the tasks that are eliminated by each design decision and for each operator.

The libraries are composed of seven operators: addition–subtraction, multiplication, division and square root, exponential, logarithm and power functions, while the precision selected has been single precision. Although double precision presents higher accuracy, the accuracy required by many applications can be achieved with single precision or a tailored precision which is much closer to single than to double precision [23]. Furthermore, the conclusions of the study we perform here for single precision can be easily generalized to double precision.

Following, we briefly analyze how we have implemented the calculation stage of each operator.

4.1. Adder–subtracter

The calculation stage of a floating-point adder–subtracter unit does not present any particular complexity and is just compounded of a fixed arithmetic adder/subtracter (preprocessing aligns the mantissas) that calculates the mantissa of the result and the sign calculator that takes into account the input signs, if it is an addition or a subtraction and which operand is bigger.

The exponent of the result is considered equal to the exponent of the biggest operand, and then it is adjusted during postprocessing if the mantissa result presents a carry (addition) or a cancelation of its most significant bits (subtraction).

4.2. Multiplication

Nowadays FPGAs include embedded multipliers that can be directly used to multiply the input mantissas. Since current embedded multipliers (18×18 multipliers for Virtex 4 and Stratix III and

Table 2
Logic reduction due to design simplifications.

	Preprocessing	Postprocessing
Denormalized Numbers Simplification	Leading one detection ($*,/,\sqrt{},\ln,x^y$) Mantissa left shifting ($*,/,\sqrt{},\ln,x^y$) Exponent correction ($+,*,/,\sqrt{},\ln,x^y$)	Mantissa shifting ($+,*,/,e^x,x^y$) Exponent correction ($+,*,/,e^x,x^y$)
Truncation Rounding	–	Rounding bits evaluation (*All*) Rounding (*All*)
Hardware Flags	Mantissa analysis (*All*) Exponent analysis (*All*)	Format exponent (*All*) Format mantissa (*All*)

IV, 25×18 multipliers for Virtex 5 and 6) have at least one of their two operand inputs with a bit width smaller than the 24 bits mantissas of single floating-point precision, our operator gets advantage of the multiplication distribution property:

$$(a+b)\times(c+d) = a\times c + a\times d + b\times c + b\times d$$

and splits de input mantissas into two parts, one corresponding to the most significant bits (upper parts, x_u and y_u) and the other part corresponding to the least significant bits (lower part, x_l and y_l) by multiplying these subparts in parallel. Afterwards, the results of the partial multiplications are added taking into account the necessary alignment between operands[4]:

$$(x_u * 2^{12} + x_l)\times(y_u * 2^{12} + y_l) = x_u * y_u 2^{24} + (x_u * y_l + x_l * y_h) * 2^{12} + x_l * y_l$$

The sign of the result is calculated with an XOR gate while the exponent is obtained by adding the input exponents (we will not go into the details of handling the exponent bias; the same will be done in the rest of the units). As in the adder–subtracter, a carry may be obtained in the mantissa result so an additional exponent adjustment is needed in postprocessing.

4.3. Division

Divisor and multiplier architectures are similar as division is the multiplication inverse function (the exponent result is now calculated by subtraction). However since there are no embedded divisors in current FPGAs, mantissas division has to be implemented with logic.

Exact binary division can be implemented by several algorithms [24], like digit recurrence methods as restoring, non-restoring or SRT algorithms [25]. From those algorithms, the most common implementations are the digit recurrence methods where one (restoring, non-restoring and radix-2 SRT algorithms) or more bits (radix-4 SRT, radix-8 SRT, etc. algorithms) of the mantissa result are calculated per division step.

Other implementations rely on some kind of approximation as algorithms based on Taylor series [26], Goldshmidt algorithm [27] or the Newton–Raphson method, where the mantissa result is computed with several extra bits to minimize the error introduced by the approximation.

For our libraries we have selected the non-restoring algorithm due to its better performance when compared to other digit recurrence methods, and due to its logic requirements when compared to approximation algorithms as these methods require of multiplications and even look-up tables in the case of the Taylor series [12]. This selection implies a division with the highest number of division steps needed to obtain the result, and therefore more clock cycles than with SRT algorithms with a radix bigger than two [28]. However it presents the highest performance per each division step (mainly an XOR for a conditional negation, and an adder) being the algorithm most suited for high clock rates.

4.4. Square root

Square Root can be computed very similarly to division when a digit recurrence algorithm is implemented and thus a non-redundant algorithm has been selected again.

The calculation unit follows the non-restoring algorithm presented in [29]. This algorithm is especially well suited for FPGAs as it works with bit-width reduced operands. As in the divisor each step is composed of a conditional negation and an addition, but now, the bit width of each step (and of that operations) is determined by the bit width of the partial result calculated before that step.

The exponent of the result is obtained just dividing by two the input exponent (shifting right one position) and a preprocessing of the mantissa in case the input exponent was odd (one shift left). No sign calculation is needed as calculated square roots are always positive.

4.5. Exponential, logarithm and power functions

The exponential and logarithm operators are based on the previous work of Detrey and Dinechin [14,15]. The technique used in both works is to reduce the input range and then use table-driven methods to calculate the function in the reduced range. This type of algorithm has been selected instead of iterative algorithms [16], due to better performance in terms of frequency. For our operators we have redesigned the exponential and logarithm datapaths, introducing several changes to improve performance while reducing resources [30].

The basis for our power unit can also be found in our previous work [30]. The power unit derives from the exponential and logarithm operators as x^y can be reduced to a combination of other operations and calculated straightforward with the transformation:

$$z = x^y = e^{y\times\ln x} \tag{2}$$

5. Libraries evaluation and comparison

The evaluation of our libraries has been carried out in a Xilinx Virtex-4 XC4VF140-11 FPGA with the ISE 10.1 environment [20]. Results obtained are post place & route with balanced mapping, and place & route with high effort.

To make a fair study and comparison of the impact of each design decision four libraries have been developed:

- `Std`: operators without any significant change with respect to the floating-point standard (supporting the four rounding methods and denormalized numbers). Only software issues as exception handling and signaling NaNs are not implemented, just providing quiet NaNs.
- `Norm`: operators handling denormalized numbers as zeros.
- `O_Rnd`: operators with only truncation rounding mode and also handling denormalized numbers as zeros.
- `HP` (High Performance): operators designed including all three design decisions; denormalized numbers as zeros, truncation rounding and use of flags.

If performance is the design goal, floating-point operators require deeply pipelined implementations. The criterion followed to determine the number of pipeline stages of the operators has been achieving a high clock frequency with a reasonable number of stages. Thus, we have determined which basic calculation or iteration in any of the operators of the `HP` library is in the critical path, being its delay the maximum delay for all other operators of `HP`.

For `Std` and `Norm` libraries, the same design criterion has been followed, while for `O_Rnd` we have chosen the same number of pipeline stages as the operators of the `HP` library to study the benefits of the use of internal flags without any other changes.

The experimental results obtained are summarized in Table 3 where each operator is characterized by the number of logic resources (**Slc**, slices), the number of pipeline stages (**Stg**, stages) and their clock frequency (**MHz**), and in Table 4 where are detailed the operators' embedded resources, which are common for all the libraries, Block RAMs (**BRAM**) and embedded DSPs (**DSP**).

[4] For Virtex 5 and 6, this scheme will correspond to the equation $a \times (b + c)$ where only one input mantissa needs to be split.

Table 3
Operators results.

	Std			Norm			0_Rnd			HP		
	Slc	Stg	MHz	Slc	Stg	MHz	Slc	Stg	MHz	Slc	Stg	MHz
+/−	414	9	250.9	402	8	252.9	369	6	267.5	344	6	286.4
$*$	471	9	250.9	148	6	253.5	109	5	242.7	102	5	250.0
/	1044	29	253.4	802	26	276.9	753	24	274.3	733	24	280.5
$\surd$	515	20	250.0	411	18	250.2	342	16	250.4	328	16	256.8
	554	17	250.0	482	16	244.4	463	15	258.6	449	15	253.9
ln	878	18	234.0	777	16	250.6	737	14	250.3	732	14	250.3
x^y	1734	37	210.0	1472	34	209.4	1457	33	220.2	1433	33	214.8

Table 4
Libraries' common resources.

	$*$	e^x	ln	x^y
DSP	4	4	5	13
BRAM	–	1	2	3

Table 5
Operators results. Commercial library.

		Xilinx [31]		Norm		NormXlx	
	Stg	Slc	MHz	Slc	MHz	Slc	MHz
+/−	8	418	256.1	402	252.9	395	257.9
$*$	6	150	246.9	148	253.5	144	255.0
/	26	668	175.7	802	276.9	777	276.9
	27	868	290.9	821	293.7	781	296.8
$\surd$	18	418	190.7	411	250.2	389	250.6

Our evaluation has been carried out in three main parts.

1. Comparison of operators with a commercial library to verify the quality of our libraries.
2. Study and comparison of the results for each component and for each of the libraries, analyzing the impact of each design decision on preprocessing and postprocessing logic. As x^y is composed of a chain of other operators, we have not included this unit in this study as it will distort this global analysis.
3. Analysis of the capabilities of current FPGAs to implement floating-point operators.

5.1. Comparison with respect to a commercial library

As reference commercial library we have selected Xilinx floating-point operators (logic core Floating-Point Operator v4.0 [31]). This library is parameterizable (variable exponent and mantissa bit widths, number of pipeline stages), denormalized numbers are not supported and rounding is restricted to round to nearest.

Consequently, Xilinx operators are almost equivalent to our `Norm` ones, only differing in the calculation of the rounding as Xilinx does not support the four floating-point standard rounding modes, see Section 3.2. Therefore, to make an exact comparison we have tuned our `Norm` operators restricting rounding to round to nearest, `NormXlx` operators. Finally we have configured Xilinx and `NormXlx` operators with the same number of stages as the ones of `Norm`.

For the four basic operators some common features can be observed in Table 5. Our operators achieve better frequencies, with a remarkable increase in both divisor and square root. The 100 MHz increase of the divisor is due to a design improvement as one stage is removed making a dual first division step[5] so the calculation of the guard bit is unnecessary. This makes that the Xilinx operator is configured with one stage less than its optimum. However, if divisors are configured with Xilinx optimum number of stages, 27, our operator is still faster.

[5] *Dividend* − *Divisor*, for the cases where mantissa dividend is bigger or equal than divisor one, and (*Dividend* $*$ 2) − *Divisor* for cases where divisor mantissa is bigger.

Regarding the square root, the 60 MHz increase is due to the algorithm selected or how it is implemented, as the Xilinx operator is below 200 MHz until it is configured with 26 stages.

With respect to the resources used, all our `Norm` operators are slightly smaller although implementing the four rounding methods (except division with 26 stages, which is not comparable as ours is 100 MHz faster).

When comparing Xilinx operators with `NormXlx` ones, the improvements are increased as we are removing logic from our operators. The biggest impacts can be observed at the divisor and at the square root due to the removal of part of the sticky bit calculation logic. This can be possible as in both operators the calculation of the sticky bit differs between rounding to nearest and rounding up or down methods, and requires independent logic.

For the three advanced operators no comparison is possible as the Xilinx library only provides the four basic operators.

5.2. Operators evaluation

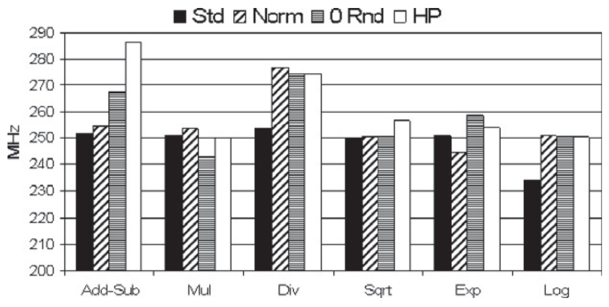
In Fig. 3 the results of Table 3 (clock frequency, number of pipeline stages and number of logic resources) are graphically depicted.

Regarding all figures of merit the final library, `HP`, outperforms the standard library `Std` in all metrics, being each operator faster while requiring less resources and less pipeline stages. And as each design decision is introduced each library outperforms or equals the previous one. Only one exception can be found, the clock frequency.

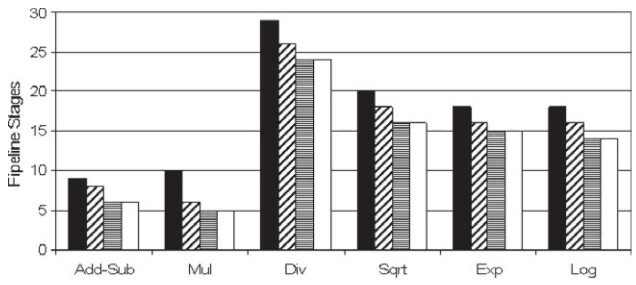
For each design decision, we are removing logic (completely or partially) and therefore the clock frequency should be increased if the pipelined architectures are not changed, as it is our case. However there are several exceptions to this general trend and mainly in operators using DSPs. Analyzing these exceptions we have found it is due to the placement and routing heuristic algorithms which do not ensure achieving the optimum implementations. In the exceptions found, the stages determining the clock frequency were exactly the same (or even with less logic) that in previous faster operators, being the slower frequency achieved due to a different placement or routing.

The importance of all the improvements can be analyzed together comparing `HP` operators from `Std`. The reduction of logic resources is between a 78.3% (multiplier) and a 16.6% (logarithm), while pipeline stages are reduced between 5 (divisor) and 2 (exponential) stages.

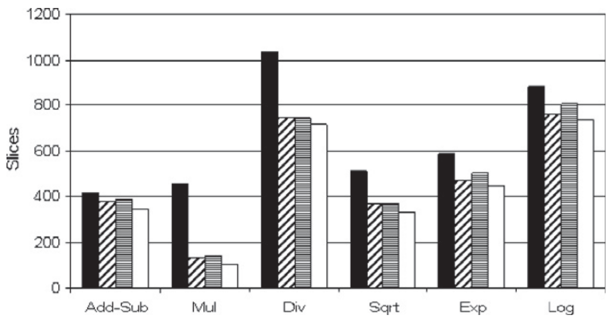
To analyze the impact of preprocessing and postprocessing overheads on the different libraries we have separately analyzed each stage: preprocessing (pre), operator calculation (cal) and postprocessing (pst). Two metrics have been analyzed, the number of resources needed, see Table 6 and the number of pipeline stages required, see Table 7.



(a) Clock Frequency.



(b) Pipeline Stages.



(c) Logic Resources.

Fig. 3. Operators evaluation.

Table 6
Split slice comparison.

	Std			Norm			0_Rnd			HP		
	Pre.	Cal.	Pst.	Pre.	Cal.	Pst.	Pre.	Cal.	Pst.	Pre.	Cal.	Pst.
+/−	146	128	198	146	128	192	146	124	155	138	124	134
$*$	262	75	186	43	75	113	43	63	51	4	63	50
/	252	729	122	47	729	64	47	695	46	4	695	12
$\sqrt{}$	150	355	49	39	355	39	39	311	16	29	311	0
e^x	124	301	169	124	301	105	124	301	74	119	301	38
ln	130	581	231	25	581	231	25	581	178	3	581	173

Table 7
Split stages comparison.

	Std			Norm			0_Rnd			HP		
	Pre.	Cal.	Pst.	Pre.	Cal.	Pst.	Pre.	Cal.	Pst.	Pre.	Cal.	Pst.
+/−	2	2	5	2	2	4	2	2	2	2	2	2
$*$	2	4	3	1	4	3	1	4	1	1	4	1
/	2	25	2	1	25	2	1	24	1	1	24	1
$\sqrt{}$	2	17	1	1	17	1	1	16	1	1	16	0
e^x	1	13	4	1	13	3	1	13	2	1	13	2
ln	2	12	4	1	12	4	1	12	2	1	12	2

5.2.1. Denormalized numbers

Comparing `Std` operator with `Norm` ones we can observe a reduction on the number of slices required from a 68.5% (multiplier) to a 11.5% (logarithm). The adder can be considered a special case as almost all the extra logic needed in `Std` is subsumed in the logic of `Norm`, and only the exponent correction due to a denormalized number is simplified, see Table 2.

`Std` overheads can be mainly found in the preprocessing stage where almost every operator has several tasks related only to denormalized numbers as these numbers need to be converted into normalized numbers, see Table 2. Results in Table 6 show that these tasks have a major impact in the resources required, mainly in the multiplier and the divider, as for both the resources for these tasks have to be replicated for both inputs.

On the other hand, the impact in the postprocessing stage is much smaller due to two facts. Firstly, there is only one output to handle so the logic is not replicated. And secondly, part of the logic of the tasks needed for handling the denormalized number is shared with the logic required for handling the result.

With respect to pipeline stages, not handling denormalized numbers reduces the required pipeline stages (between 1 and 3) in two ways:

- Elimination of stages; as some tasks are removed with dedicated logic in the datapath.
- Overlap of stages; when denormalized numbers are no longer handled, the calculation stage of some operators can directly work with the input numbers. In parallel, the preprocessing stage analyzes the inputs number type and processes the exceptions due to not normalized input.

Furthermore, it can be also observed that the handling of denormalized numbers also affects the clock frequency of the operators. Although there are specific pipeline stages for this handling, these stages become the slowest stages harming the global speed of most units (see Fig. 3a), requiring even more additional pipeline stages.

5.2.2. Rounding

Introducing the limitation of rounding methods to just rounding towards zero, `O_Rnd`, implies an additional reduction of slices, up to 69 for the square root, and between 1 and 2 stages when comparing `O_Rnd` operators with `Norm` ones.

Rounding mainly affects to the postprocessing stage and to the calculation stage as the rounding bits need to be calculated. With respect to postprocessing, the resources needed are required to evaluate the rounding bits, the rounding that implies two adders (exponent and mantissa), and the logic needed for cases where a carry can be obtained after rounding the mantissa (a multiplexor).

Regarding the calculation stage the major impact is in the operators with digit recurrence methods as extra iterations are needed for computing those bits.

5.2.3. Number types

Again, as with denormalized numbers, when applying this design decision the major impact can be found in the operators with two inputs. The logic needed (comparators) to determine the number type by analyzing the mantissa and exponent values needs to be replicated. Meanwhile, in the postprocessing the result is formatted (using a multiplexor). When this logic is removed, the saved resources are equivalent to the resources needed by the interface units. Therefore, the reduction of logic resources respect `O_Rnd` (up to a 6.8% in the multiplier) will only be effective when implementing chained operators.

5.3. Replicability

Replicability is a key issue to address when analyzing the suitability and capacity of modern FPGAs to implement floating-point applications. We can define replicability in an easy way as the number of operators that can be implemented in an FPGA considering 100% of resources available and taking as base reference the resources used by one operator. The results obtained following this definition are depicted in Fig. 4 for `HP` operators and for four diffent FPGAs which present a good mix of elements, two Virtex-4 (SX55, FX140) and two Virtex-5 (FX200, SX240).

From Fig. 4, it can be deduced the great capacity of modern FPGAs to implement complex single floating-point algorithms, even involving hundreds of operators. However, in a real scenario other facts have to be taken into account as:

- Routing stress: for large implementations routing congestion can affect very seriously the performance or make impossible to route already mapped operators.
- Use of logic resources instead of embedded elements: when no more embedded elements are available, they will be substituted by slices.
- The datapath: for operators sharing the same inputs there could be replicated logic removed by implementation tools. Additionally, for chained operators the input or output registers should be removed.

Taking all these features into account, we have designed a synthetic datapath, Fig. 5, to analyze how many times an operator can be replicated. The datapath is composed of *n* levels of 10 operators each while another 9 operators provide the final output. For coarse grain configurability, *n* can be increased adding more levels, while for fine grain configurability, operators can be added at the output. The datapath has been designed to prevent implementation tools removing duplicated logic: there are no two equal inputs, using Z_{ij} (the output of each operator) and $\underline{Z_{ij}}$ (the output but with the bits reordered) while the operators are registered only at their outputs. Operators under study are the adder among the operators not using embedded elements and the logarithm among the ones using embedded elements (for this case, we have chained operators as there is only one input per operand). The reference FPGA used has been the Virtex-4 XC4VF140 and the design strategy has been balanced implementation.

In Fig. 6 the results obtained for the adder operator are shown. The expected number of operators obtained extrapolating the results for one operator is widely exceeded. Instead of the 183 operators expected (first vertical line starting from the left in Fig. 6) it is possible to implement up to 241 adders. The first reason for this increase, is that only the outputs are registered in these tests. The second reason is the way the implementation tools work. When reaching the usage limits of the FPGA, the implementation tools focused their work on area optimization (although results are obtained with balanced goal). Therefore, and for operators without input registers, two more theoretical limits have been calculated: the first one with balanced implementation (203, second vertical

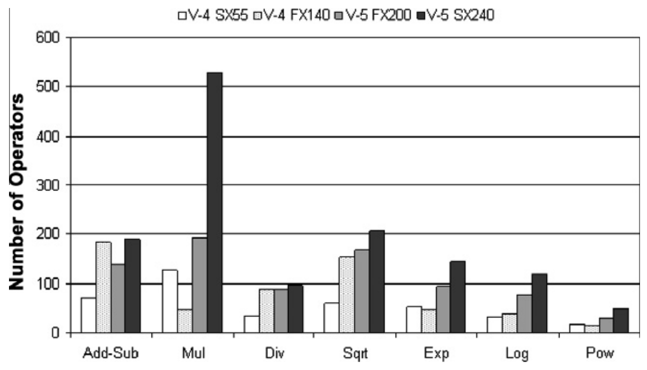


Fig. 4. Operators replicability.

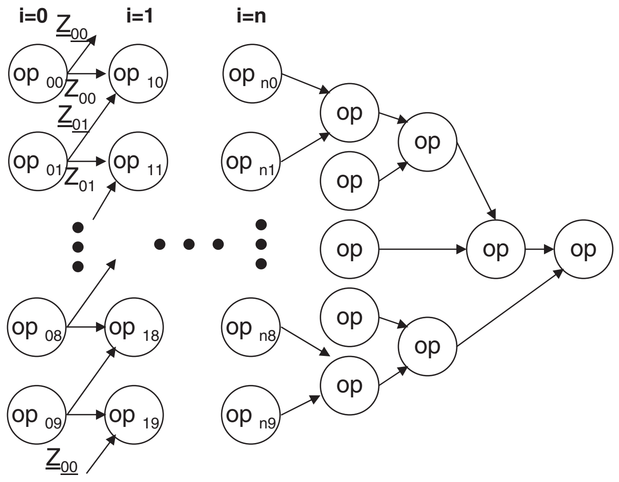


Fig. 5. Synthetic datapath.

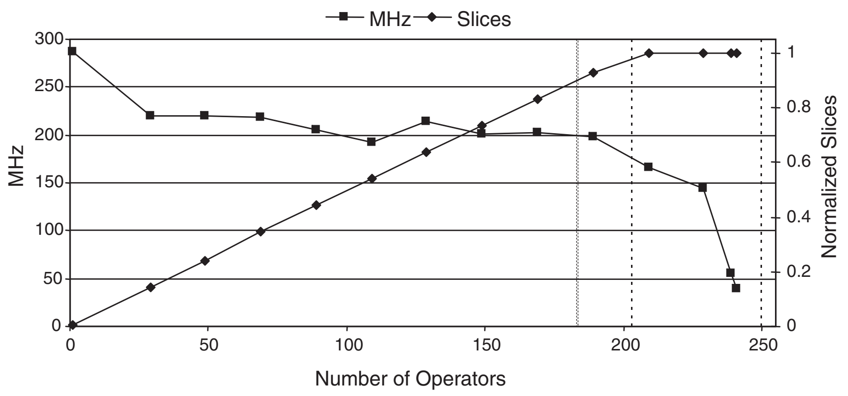


Fig. 6. Adder replicability results.

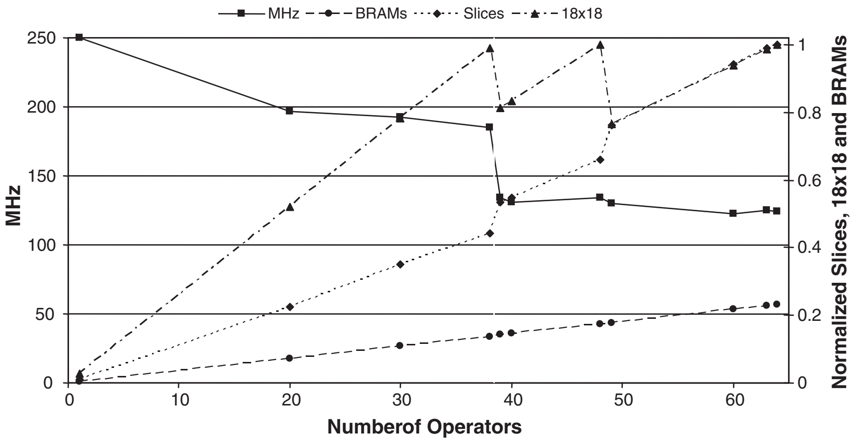


Fig. 7. Logarithm replicability results.

line) and the second with area oriented implementation (250, third vertical line). For the adder, this area oriented limit has proved to be the one closest to the results obtained in the experimental tests, where we have obtained just 3.6% fewer operators than this limit.

In the Slices graph (results are normalized, being 1 the total number of slices of the FPGA) we can see that 100% of resources of the FPGA are achieved before the 241 operators. The number of slices used grows linearly with the operators until the 100% is achieved. Then, the implementation tools reduce the slices needed per operator as the resources limit has been achieved. In this point, the speed of the datapath, MHz graph, is seriously affected as for each new operator introduced routing becomes more difficult, harming the performance.

When the embedded elements are the limiting components the situation changes, as can be seen in Fig. 7 for the logarithm. When the limit is going to be achieved (between 38 and 39 logarithms) the implementation tools replace one 18×18 multipliers of each logarithm by logic. Therefore, for 39 multipliers, all the operators now have four 18×18 multipliers instead of the previous five, when it would be strictly necessary 36 logarithms with five 18×18 and three with four 18×18. Consequently there is a big increase in the number of slices used while a big decrease is observed in the number of 18×18 multipliers. The same circumstance happens for 48 operators. However, the next frontier (64 operators) becomes the final limit as with 64 logarithms we have reached the 100% of slices used, making impossible to replace 64 18×18 multipliers with logic.

6. Towards standard compliance and high performance

As seen in Section 5, performance of FPGA floating-point operators can be considerably improved when introducing a few modifications to obtain substandard operators. However, how can we obtain a standard compliant library while trying to preserve improvements of sub-standard libraries?

Since using dedicated flags for the number type has no cost in terms of standard compliance, we can always apply it by using interfaces. The two other decisions cannot be directly applied if compliance with the standard is a must, so trying to fulfill the standard will require additional techniques.

6.1. Simplification of denormalized numbers: one bit exponent extension

A number is denormalized for a given floating-point precision depending on the exponent bitwidth for that precision, e_b. Thus, it will be denormalized if the value of its leading one corresponds to a 2^x with x in $[-2^{e_b-1} - m_b, -2^{e_b-1} + 1]$ while it will be normalized if x is in $[-2^{e_b-1} + 2, 2^{e_b-1})$.

Consequently, a denormalized number for a given precision will be a normalized number for a precision with one extra exponent bit, $e'_b = e_b + 1$ as now the x corresponding to the leading one will be in the new normalized range of $[-2^{e'_b-1} + 2, 2^{e'_b-1}) = [-2^{e_b} + 2, 2^{e_b})$.

Therefore, the operators handling of denormalized numbers as zeros can be applied in combination with an extension of the num-

Table 8
Operators results with the final proposed features.

		±	*	/	√	e^x	ln	x^y
Slices	HW	378	138	742	366	470	760	1455
	HW+1	385	145	742	368	505	805	1508
Speed (MHz)	HW	270.0	250.2	286.2	250.1	246.7	258.5	213.1
	HW+1	273.9	252.3	284.4	250.6	248.3	258.5	210.2
Stages	HW	8	6	26	17	16	16	34

ber precision in one exponent bit in order to preserve the resolution of the denormalized numbers of the not extended format.

Now, as in the use of flags for the number type, the interfaces between standard numbers and hardware numbers will be in charge of transforming the numbers. Meanwhile operators will need to handle numbers with an extra bit in the exponent. However, handling exponents has a small hardware cost in the operators, much smaller than handling denormalized numbers, see Section 6.3.

Regarding the standard compliance, this solution ensures that we are not loosing accuracy nor resolution, as the denormalized numbers in the new format will always be zero in the standard precision.

Furthermore, with the extended precision more accurate results can be obtained. In datapaths involving several operations we can find that partial results that were zeros or infinities with the standard precision, with the new precision are normalized numbers. Therefore, final results that were a zero, an infinity or NaN with standard precision can have a numeric value with the new precision. Additionally, partial results with a standard denormalized number, and reduced precision, now can be normalized numbers with not reduced precision, so the subsequent operations are more accurate.

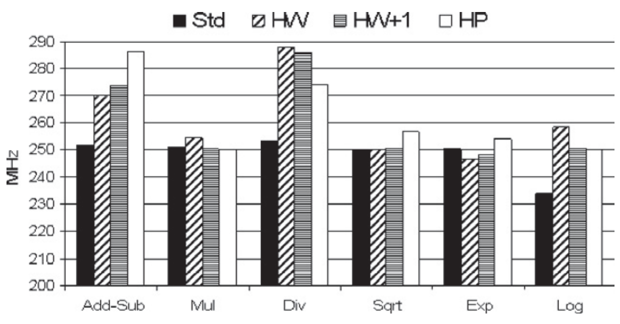
Table 9
Required interfaces.

	SW-HW		HW-SW	
	HW	HW + 1	HW	HW + 1
Slices	42	124	35	120
Speed (MHz)	434.2	323.5	700.8	295.6
Stages	1	2	1	2

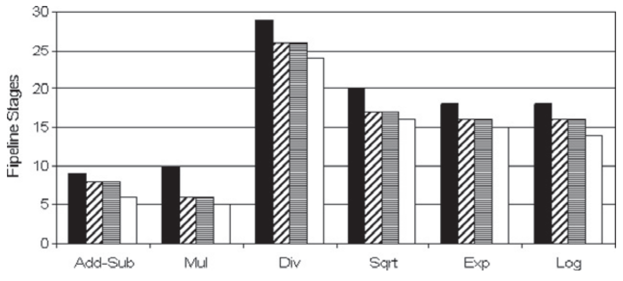
6.2. Truncation rounding: mantissa extension

To try to obtain the same accuracy with truncation rounding than with round to nearest, an extension of one bit of mantissa can be applied. The ulp of the new precision with extended mantissa, *ulp*', has a value of half of the standard ulp. So the maximum error of extended precision and truncation rounding, 1 *ulp*', will equal the maximum error obtained with standard precision and round to nearest, 0.5 ulp.

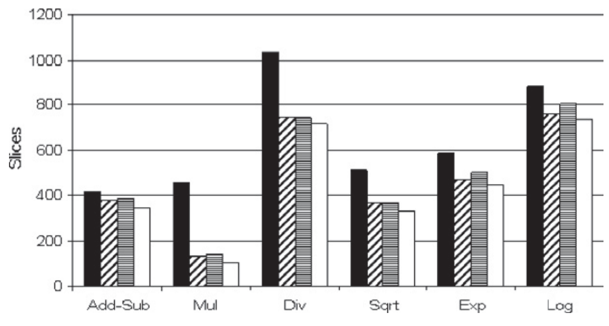
This solution will require that each operator would compute the input mantissas with one extra bit and also generate outputs with the extra bit. Part of the advantages of truncation rounding, not having to compute the rounding and sticky bit, is lost while the calculation unit has to work with input mantissas of one more bit, requiring more resources.



(a) Clock Frequency.



(b) Pipeline Stages.



(c) Logic Resources.

Fig. 8. Standard operators evaluation.

Concerning the bias introduced by truncation rounding, this source of no standard compliance cannot be corrected and we can only try to reduce the introduced bias by extending the precision with more mantissa bits, making the relative error per operation smaller and so the bias. Nevertheless, the bias can have an important impact on the accuracy of results, for example, in the statistical analysis of large quantities of data or in applications requiring a high degree of accuracy.

Therefore as bias cannot be corrected, round to nearest should be kept for standard compliance and to avoid the impact of bias in given applications.

6.3. FPGA-oriented floating-point library

From the previous analysis and discussions, we consider the following features as good options for almost standard compliance while still taking advantage of the FPGA flexibility:

- Use of dedicated flags for the number type.
- Handling denormalized numbers as zeros while exponent is extended in one bit.
- Round to nearest should be kept.

Following these recommendations we have developed two last libraries taking into account two possible scenarios: one with the one bit exponent extension, HW + 1 (Hardware library and +1 indicates the exponent extension), and another without the exponent extension, HW. Table 8 summarizes the results for the developed operators with those features while Table 9 shows the results for the required interfaces.

As can be seen in both tables, the exponent extension has impact mainly in the interfaces required, as now they have to handle the conversion between denormalized and normalized (one more stage, more resources and less frequency), and in the complex operators (up to 7.4% of slices in the exponential). Thereby, if it is known that in a given application denormalized numbers are not in the range of partial or final results, it would be better using operators with no bit extension.

When comparing the results of these final libraries, HW and HW + 1, with respect to the other libraries we can focus on Fig. 8, which shows the results of the operators of HW and HW + 1 compared to the results of the initial standard library, Std, and to the results of the sub-standard high performance library, HP.

Firstly, it can be observed that HW and HW+1 operators preserve partially the clock frequency improvements achieved with the sub-standard operators, except for the exponential operator.

If we focus on the graphics depicting the number of pipeline stages and use of logic resources it can be seen that for both features, although not all the improvements achieved with sub-standard operators can be preserved, the HW and HW + 1 results are closer to the results for HP than to the Std ones. HW and HW + 1 operators present a reduction of pipeline stages between one and three fewer stages than Std operators while the increase of stages with respect to HP operators is between one and two stages.

And finally, considering the use of slices, the improvements achieved with respect to Std operators are between 12% (exponential) and 67.2% (multiplier) for HW operators and between 10% and 66.1% for HW + 1 operators.

7. Conclusions

Current nanometer technologies allow the implementation of complex floating-point applications in a single FPGA device. Nevertheless, the complexity of this kind of operators (deep pipelines, computationally intensive algorithms and format overhead) makes their design especially long and complicated. The availability of complete and FPGA-oriented libraries significantly simplifies the design of those complex floating-point applications.

In this work we have presented several design decisions that can be made to improve the performance of floating-point operators implemented in FPGA architectures. An in depth analysis of the performance-accuracy trade-offs of those decisions has been carried out through the development and comparison of complete floating-point libraries of operators. These libraries include both conventional (adder/subtracter, multiplier, divider and square root) and advanced (exponential, logarithm and power functions) operators. Actually, the power operator is included in a floating-point library for the first time.

Three design decisions targeting the simplification of floating-point complexity have been thoroughly analyzed. Handling complexity requires logic resources, thus as the complexity is reduced so are the logic resources needed for an operator. Additionally, as less resources are used, operators implementations become more efficient, and speed is also improved or the number of pipeline stages is reduced. Our approach has focused on determining those features of floating-point arithmetic that require a heavy processing overhead while their relevance in the format is minimum or can be neglected.

The extended experimental results validate the different decisions made to improve performance and area requirements of floating-point units and can be taken as guide by hardware designers when implementing this kind of applications.

Finally, a set of features that implies improved performance and reduced resources has been chosen to design two almost standard-compliant libraries where their tailored implementation adapts the standard format to improve performance taking advantage of FPGA flexibility.

Acknowledgements

This work has been partly funded by BBVA under Contract P060920579 and by the Spanish Ministry of Science and Innovation through the Project TEC2009-08589.

References

[1] I. Kuon, J. Rose, Measuring the gap between FPGAs and ASICs, computer-aided design of integrated circuits and systems, IEEE Transactions on 26 (2) (2007) 203–215.

[2] R. Scrofano, M. Gokhale, F. Trouw, V. Prasanna, Accelerating molecular dynamics simulations with reconfigurable computers, Parallel and Distributed Systems, IEEE Transactions on 19 (6) (2008) 764–778.

[3] G. Zhang, P. Leong, C. Ho, K. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, W. Luk, Reconfigurable acceleration for Monte Carlo based financial simulation, 2005, pp. 215–222, doi:10.1109/FPT.2005.1568549.

[4] L. Zhuo, V. Prasanna, High-performance designs for linear algebra operations on reconfigurable hardware, Computers, IEEE Transactions on 57 (8) (2008) 1057–1071.

[5] K. Underwood, FPGAs vs. CPUs: trends in peak floating-point performance, in: FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, ACM, New York, NY, USA, 2004, pp. 171–180. doi:http://doi.acm.org/10.1145/968280.968305.

[6] A. George, H. Lam, G. Stitt, Novo-g: At the forefront of scalable reconfigurable supercomputing, Computing in Science Engineering 13 (1) (2011) 82–86, doi:10.1109/MCSE.2011.11.

[7] S. Che, J. Li, J.W. Sheaffer, K. Skadron, J. Lach, Accelerating compute-intensive applications with gpus and fpgas, Application Specific Processors, Symposium on (2008) 101–107. doi:http://doi.ieeecomputersociety.org/10.1109/SASP.2008.4570793.

[8] F. de Dinechin, J. Detrey, O. Cret, R. Tudoran, When FPGAs are better at floating-point than microprocessors, in: FPGA '08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, ACM, New York, NY, USA, 2008, p. 260. doi:http://doi.acm.org/10.1145/1344671.1344717.

[9] M. Chiu, M.C. Herbordt, M. Langhammer, Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems, in: Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications, 2008, pp. 1–10.

[10] B. Lee, N. Burgess, Parameterisable floating-point operations on fpga, in: Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers, vol. 2, 2002, pp. 1064–1068.
[11] G. Govindu, L. Zhuo, S. Choi, V. Prasanna, Analysis of high-performance floating-point arithmetic on FPGAs, in: International Parallel and Distributed Processing Symposium, 2004, pp. 149–156.
[12] S.B. Xiaojun Wang, M. Leeser, Advanced components in the variable precision floating-point library, in: IEEE Field-Programmable Custom Computing Machines, 2006, pp. 249–258.
[13] J. Liang, R. Tessier, O. Mencer, Floating point unit generation and evaluation for FPGAs, in: IEEE Field-Programmable Custom Computing Machines, 2003, pp. 185–194.
[14] J. Detrey, F. de Dinechin, A parameterized floating-point exponential function for FPGAs, in: IEEE International Conference Field-Programmable Technology, 2005, pp. 27–34.
[15] J. Detrey, F. de Dinechin, A parameterized floating-point logarithm operator for FPGAs, in: Signals, Systems and Computers, 2005. Conference Record of the Thirty-Ninth Asilomar Conference, 2005, pp. 1186–1190.
[16] J. Detrey, F. de Dinechin, X. Pujol, Return of the hardware floating-point elementary function, in: Symposium on Computer Arithmetic, 2007, pp. 161–168.
[17] G. Govindu, R. Scrofano, V. Prasanna, A library of parameterizable floating-point cores for FPGAs and their application to scientific computing, in: International Conference on Engineering of Reconfigurable Systems and Algorithms, 2005.
[18] IEEE Standard Board, IEEE standard for binary floating-point arithmetic, The Institute for Electrical an Electronics Engineers, 1985.
[19] Altera Corporation, <http://www.altera.com/products/ip/dsp/arithmetic/m-alt-float-point.html>.
[20] Xilinx, <http://www.xilinx.com/products/index.htm>.
[21] J. Detrey, F. de Dinechin, Flopoco project (floating-point cores), <http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/>.
[22] N. Kapre, A. DeHon, Optimistic parallelization of floating-point accumulation, in: 18th IEEE Symposium on Computer Arithmetic, 2007, pp. 205–216.
[23] J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Dongarra, Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems), LAPACK Working Note, July 2006.
[24] S.F. Oberman, M.J. Flynn, Division algorithms and implementation, IEEE Transactions on Computers 46 (8) (1997) 833–854.
[25] C. Freiman, Statistical analysis of certain binary division algorithms, Proceedings of the IRE 49 (1) (1961) 91–103, doi:10.1109/JRPROC.1961.287780.
[26] P. Hung, H. Fhmy, O. Mencer, M.J. Flynn, Fast division algorithm with a small lookup table, in: 33rd Asilomar Conference on Signals, Systems and Computersc, 1999, pp. 24–27.
[27] R. Goldschmidt, Applications of Division by Convergence, Master's thesis, Massachusetts Inst. of Technology, 1964.
[28] K.S. Hemmert, K.D. Underwood, Floating-point divider design for FPGAs, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15 (1) (2007) 115–118.
[29] Y. Li, W. Chu, Implementation of single precision floating point square root on FPGAs, in: IEEE Symposium on FPGA-Based Custom Computing Machines, 1997, p. 226.
[30] P. Echeverría, M. López-Vallejo, An FPGA implementation of the powering function with single floating-point arithmetic, in: Conference of Real Number and Computers, 2008.
[31] Xilinx, <http://www.xilinx.com/products/ipcenter/floating_pt.htm>.

Pedro Echeverra received the M.S. degree in telecommunications with a major in electronics from the Universidad Politécnica de Madrid, Spain, in 2005. He is currently working towards the Ph.D. degree at the Department of Electronic Engineering, Universidad Politécnica de Madrid. His research interests include high-performance computing with FPGAs, application-specific high-performance programmable architectures computer arithmetic and random number generation.

Marisa Ló**pez-Vallejo** received the M.S. and Ph.D. degrees from the Universidad Politécnica de Madrid, Spain, in 1993 and 1999, respectively. She is an Associate Professor in the Department of Electronic Engineering, Universidad Politécnica de Madrid. She was with Bell Laboratories, Lucent Technologies as a member of the technical staff. Her research activity is currently focused on low-power and temperature-aware design, CAD for hardware/software codesign of embedded systems, and application-specific high-performance programmable architectures.