Available online at www.sciencedirect.com
ScienceDirect
Microprocessors and Microsystems 31 (2007) 537–545
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

Parameterized floating-point logarithm and exponential functions for FPGAs

Jérémie Detrey *, Florent de Dinechin

Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon, France

Available online 28 February 2006

Abstract

As FPGAs are increasingly being used for floating-point computing, the feasibility of a library of floating-point elementary functions for FPGAs is discussed. An initial implementation of such a library contains parameterized operators for the logarithm and exponential functions. In single precision, those operators use a small fraction of the FPGA's resources, have a smaller latency than their software equivalent on a high-end processor, and provide about ten times the throughput in pipelined version. Previous work had shown that FPGAs could use massive parallelism to balance the poor performance of their basic floating-point operators compared to the equivalent in processors. As this work shows, when evaluating an elementary function, the flexibility of FPGAs provides much better performance than the processor without even resorting to parallelism. The presented library is freely available from http://www.ens-lyon.fr/LIP/Arenaire/.

Keywords: Elementary functions; Parameterized operators; Logarithm; Exponential; Floating-point; FPGA; FPLibrary

1. Introduction

A recent trend in FPGA computing is the increasing use of floating-point. Many libraries of floating-point operators for FPGAs now exist [18,8,1,11,6], typically offering the basic operators $+, -, \times, /$ and $\sqrt{}$. Published applications include matrix operations, convolutions and filtering. As FPGA floating-point is typically clocked 10 times slower than the equivalent in contemporary processors, only massive parallelism (helped by the fact that the precision can match closely the application's requirements) allows these applications to be competitive to software equivalent [13,5,10].

More complex floating-point computations on FPGAs will require good implementations of elementary functions such as logarithm, exponential, trigonometric, etc. These are the next useful building blocks after the basic operators. This paper describes both the logarithm and exponential functions, a first attempt to a library of floating-point elementary functions for FPGAs.

Elementary functions are available for virtually all computer systems. There is currently a large consensus that they should be implemented in software [17]. Even processors offering machine instructions for such functions (mainly the x86/x87 family) implement them as microcode. On such systems, it is easy to design faster software implementations: Software can use large tables which would not be economical in hardware [19]. Therefore, no recent instruction set provides instructions for elementary functions.

Implementing floating-point elementary functions on FPGAs is a very different problem. The flexibility of the FPGA paradigm allows to use specific algorithms which turn out to be much more efficient than a processor-based implementation. We show in this paper that a single precision function consuming a small fraction of FPGA resources has a latency equivalent to that of the same function in a 2.4 GHz PC, while being fully pipelinable to run at 100 MHz. In other words, where the basic floating-point operator ($+, -, \times, /$, and $\sqrt{}$) is typically 10 times slower on an FPGA than its PC equivalent, an elementary function will be more than 10 times faster for precisions up to single precision.

* Corresponding author.
E-mail addresses: Jeremie.Detrey@ens-lyon.fr (J. Detrey), Florent.de.Dinechin@ens-lyon.fr (F. de Dinechin).

0141-9331/$ - see front matter
doi:10.1016/j.micpro.2006.02.008

Writing a *parameterized* elementary function is a completely new challenge: to exploit this flexibility, one should not use the same algorithms as used for implementing elementary functions in computer systems [19,15,14]. This paper describes an approach to this challenge, which builds upon previous work dedicated to fixed-point elementary function approximations (see [7] and references therein).

The authors are aware of only two previous works on floating-point elementary functions for FPGAs, studying the sine function [16] and studying the exponential function [9]. Both are very close to a software implementation. As they do not exploit the flexibility of FPGAs, they are much less efficient than our approach, as Section 4 will show.

1.1. Notations

The input and output of our operators will be $(3 + w_E + w_F)$-bit floating-point numbers encoded in the freely available FPLibrary format [6] as follows:

- F_X: the w_F least significant bits represent the fractional part of the mantissa $M_X = 1.F_X$.
- E_X: the following w_E-bit word is the exponent, biased by $E_0 = 2^{w_E-1} - 1$.
- S_X: the next bit is the sign of X.
- exn_X: the two most significant bits of X are internal flags used to deal more easily with exceptional cases, as shown in Table 1.

2. A floating-point logarithm

2.1. Evaluation algorithm

2.1.1. Range reduction

We consider here only the case where X is a valid positive floating-point number (i.e., $\mathrm{exn}_X = 01$ and $S_X = 0$), otherwise the operator simply returns NaN. We therefore have:

$$X = 1.F_X \cdot 2^{E_X - E_0}.$$

If we take $R = \log X$, we obtain

$$R = \log(1.F_X) + (E_X - E_0) \cdot \log 2.$$

In this case, we only have to compute $\log(1.F_X)$ with $1.F_X \in [1,2)$. The product $(E_X - E_0) \cdot \log 2$ is then added back to obtain the final result.

Table 1
Value of X according to its exception flags exn_X

exn_X	X
00	0
01	$(-1)^{S_X} \cdot 1.F_X \cdot 2^{E_X - E_0}$
10	$(-1)^{S_X} \cdot \infty$
11	NaN (*Not a Number*)

To avoid catastrophic cancellation when adding the two terms, and consequently maintain low error bounds, we use the following equation to center the output range of the fixed-point log function around 0:

$$R = \begin{cases} \log(1.F_X) + (E_X - E_0) \cdot \log 2, & \text{when } 1.F_X \in [1, \sqrt{2}), \\ \log\left(\frac{1.F_X}{2}\right) + (1 + E_X - E_0) \cdot \log 2, & \text{when } 1.F_X \in [\sqrt{2}, 2). \end{cases} \quad (1)$$

We therefore have to compute $\log M$ with the input operand $M \in [\sqrt{2}/2, \sqrt{2})$, which gives a result in the interval $[-\log 2/2, \log 2/2)$.

We also note in the following $E = E_X - E_0$ when $1.F_X \in [l, \sqrt{2})$, or $E = 1 + E_X - E_0$ when $1.F_X \in [\sqrt{2}, 2)$.

2.1.2. Fixed-point logarithm

As we are targeting floating-point, we need to compute $\log M$ with enough accuracy to guarantee faithful rouding, even after a possible normalization of the result. As $\log M$ can be as close as possible to 0, a straightforward approach would require at least a precision of $2w_F$ bits, as the normalization could imply a left shift of up to w_F bits, and w_F bits would still be needed for the final result.

But one can remark that when M is close to 1, $\log M$ is close to $M - 1$. Therefore, a two-step approach consisting of first computing $\log M/(M-1)$ with a precision of $w_F + g_0$ bits and then multiplying this result by $M - 1$ (which is computed exactly) leads to the targeted accuracy at a smaller cost.

The function $f(M) = \log M/(M-1)$ is then computed by a generic polynomial method [7]. The order of the considered polynomial obviously depends on the precision w_F.

2.1.3. Reconstruction

As the evaluation of $f(M)$ is quite long, we can in parallel compute the sign of the result: if $E = 0$, then the sign will be the sign of $\log M$, which is in turn positive if $M > 1$ and negative if $M < 1$. And if $E \neq 0$, as $\log M \in [\sqrt{2}/2, \sqrt{2})$, the sign will be the sign of $E \cdot \log 2$, which is the sign of E.

We can then compute in advance the opposite of E and $M - 1$ and select them according to the sign of the result. Therefore, after the summation of the two products $E \cdot \log 2$ and $Y = f(M) \cdot (M - 1)$, we obtain Z the absolute value of the result.

The last steps are of course the renormalization and rounding of this result, along with the handling of all the exceptional cases.

2.2. Architecture

The architecture of the logarithm operator is given on Fig. 1. It is a straightforward implementation of the algorithm presented in Section 2.1. Due to its purely sequential dataflow, it can be easily pipelined. The values for the two parameters g_0 and g_1 are discussed in Section 2.3.