An improved convergence algorithm to compute $ln(x)$ – FPGA implementations

Géry J. A. Bioul, Martín Vázquez, Héctor N. Acosta, Martín Oriol*
Universidad Fasta, Facultad de Ingeniería
Gascón 3145 – B7600FNK – Mar del Plata. Argentina.
www.ufasta.edu.ar

Abstract

This paper presents FPGA implementations of classical algorithms for computing $ln(x)$ with some improvement at the level of the multiplication steps, and step skipping techniques. One starts from a practical implementation of $ln(x)$ computation using a convergence method. The function is approximated by a multiplicative normalization technique, however, thanks to the peculiarity of the multiplicative factor, namely $(1 + a_i .2^{-i})$, with $a_i \in \{-1, 0, 1\}$, the successive multiplications have been replaced by additions. Doing so, one saves the use of LUT's and eventually reduces processing time, as addition is generally faster than multiplication. Further, the acceleration technique, based on skipping trivial steps, improves performances. Implementations for FPGA are presented with time and slice cost evaluations. The Xilinx Virtex IV has been used for comparative analysis of 8 to 64-bit logarithm computing devices.

Keywords: FPGA, $ln(x)$, convergence method, multiplicative normalization, Xilinx Virtex IV.*

* This project is supported by FASTA University, Faculty of Engineering, B7600FNK Mar del Plata, Argentina.

1. INTRODUCTION

Most often the computation of functions such as logarithms, and exponential or trigonometric functions are made through software-implemented algorithms applied to floating-point representations. Hardware or micro-programmed systems are mainly justified for special-purpose computing devices such as ASIC or embedded systems. As it is generally not possible to get an exact result, approximation methods have to be used together with error estimation techniques. Newton-Raphson, Goldschmidt algorithm, Taylor MacLaurin series or Polynomial approximations are the most common approaches to compute these functions. For trigonometric functions, *CORDIC* (linear convergence) algorithms are well suited. Arguments included in the range [1, 2[- floating-point IEEE standard - are suitable for most approximation methods that need to limit the range of the argument. Whenever a specific range is imposed on the operands, a pre-scaling operation may be necessary: so an initial step may be included in the algorithmic procedure. Crucial questions for approximation methods are error estimation and effective rounding techniques; these problems start from tables design (first approximation LUT) up to the final result. Numerous methods, algorithms and implementations are proposed in the literature [1, 2, 3, 4, and 5]; the choice will depend upon the speed/cost compromises and other constraints imposed on the designer. Approximations methods usually assume available the four basic operations as arithmetic primitives at hand, together with look-up tables for a first "reasonably good" approximation to start from. This paper presents a practical implementation of $ln\ (x)$ computation using a convergence method [2]. The function is approximated by a multiplicative normalization technique, however, thanks to the peculiarity of the multiplicative factor, namely $(1 + a_i .2^{-i})$, with $a_i \in \{-1, 0, 1\}$, the successive multiplications have been replaced by additions. Doing so, one saves the use of LUT's and eventually reduces processing time, as addition is generally faster than multiplication. Furthermore an acceleration technique, based on skipping trivial steps, has been taken into account to improve performances. Implementations for FPGA are presented with time and slice cost evaluations. The Xilinx Virtex IV [6, 7, 8, 9] has been used for comparative analysis of 8 to 64-bit logarithm computing devices.

2. THEORETICAL BACKGROUND - LOGARITHM FUNCTION APPROXIMATION BY A CONVERGENCE METHOD USING MULTIPLICATIVE NORMALIZATION

Convergence methods consist in two parallel processes on two related sequences; typically, one sequence converges to 1 (*multiplicative normalization*) or 0 (*additive normalization*) while the other one converges to the function to approximate. Division using *Goldschmidt's* algorithm is an example of multiplicative normalization: while the divisor sequence converges to 1, the dividend converges to the desired quotient.

Define

$$c(i) = 1 + a_i .2^{-i}, \quad a_i \in \{-1, 0, 1\} \tag{1}$$

as the *multiplicative normalizing function*, where a_i is selected in such a way that the sequence

$$x(i+1) = x(i).c(i) \text{ (}auxiliary\ sequence\text{)} \qquad x(i) \in B(2^n) \tag{2}$$

converges towards 1. Then, the sequence

$$y(i+1) = y(i) - ln\ c(i) \tag{3}$$

can be set to converge toward the result $ln\ (x)$. If $y(0)$ and $x(0)$ are respectively set to 0 and to the argument x, and assuming $x(p) \cong 1$, one can write

$$x(p) = x.\Pi_i\, c(i) \cong 1 \rightarrow 1/x \cong \Pi_i\, c(i);\ \ y(p) = y - \Sigma_i\, ln\ c(i) = -\ ln\ \Pi_i\, c(i) = ln\ (x). \quad (4)$$

To make the convergence of (2) possible, the argument x needs to be in a range such that

$$x.min\ (lim_{p \rightarrow \infty} \Pi_{1 \le i \le p}\, c(i)) \le 1 \quad \text{and} \quad x.max\ (lim_{p \rightarrow \infty} \Pi_{1 \le i \le p}\, c(i)) \ge 1$$

that is

$$x \le 1/lim_{p \rightarrow \infty} \Pi_{1 \le i \le p}\ (1 - 2^{-i}) \text{ and } x \ge 1/lim_{p \rightarrow \infty} \Pi_{1 \le i \le p}\ (1 + 2^{-i}),\ \text{that is}\ \ 0.42 \le x \le 3.45. \quad (5)$$

This means that the argument x could need to be pre-scaled to fall in the range (5). An argument x in the range [1, 2[(such as e.g. a floating-point mantissa) fits perfectly; otherwise use a straightforward pre-scaling operation that replaces x by x' such that $x = x'.2^s$ (x' in [1, 2[); the algorithm computes $ln\ (x')$, then a final additive correction of $s.ln\ (2)$ is completed. Observe that the lower bound of (5) can be lowered to 0.21, as $(1+2^0)$ can be accepted as a first normalizing factor for computing $x(1)$.

In practical implementations of this algorithm, look-up tables are used to read out the successive values of $ln\ (1 \pm 2^{-i})$, needed to compute $y(i+1)$ of (3). For x in [½, 2[, a_i can be selected according to the following rules:

$$a_0 = 0, \quad (6)$$

$$\text{if } x(i) > 1,\ \ a_i = -\ x_{-i}\ (i), \qquad i \ge 1 \quad (7)$$

$$\text{if } x(i) < 1,\ \ a_i = +\ x_{-i}(i).not\ (x_{-i-1}(i)). \qquad i \ge 1 \quad (8)$$

The above rules are justified by the following two lemmas, also showing that the convergence rate reaches precision p after p steps (linear convergence).

Lemma 1.

Let

$$x(k) = 1 + 2^{-k} + \varepsilon, \qquad 0 \le \varepsilon \le 2^{-k} - 2^{-n}, \quad k \le n, \quad (9)$$

be the n-bit auxiliary sequence vector at step k; then

$$1 - 2^{-2k} \le x(k).(1 - 2^{-k}) < 1 + 2^{-k}. \quad (10)$$

Proof

The left inequality is trivial, it corresponds to $\varepsilon = 0$. The right inequality is deduced from the computation of $x(k).(1 - 2^{-k})$ for ε maximum, i.e. $2^{-k} - 2^{-n}$.

The practical interpretation of (10) is the impact of rule (7) on $x(k+1)$ whenever $x(k)$ is greater than one with a fractional part made up of a $(k-1)$-zero string and a one at position k. $x(k+1)$ will be either greater than one, exhibiting a similar pattern with at least one zero more, or inferior to one ($x_0(k+1) = 0$) with at least $2k$ one's as the header of the fractional part. In both cases, the target value $x(p) = 1$ is approximated by $x(k+1)$ with at least one bit more.

Lemma 2.

Let

$$x(k) = 1 - 2^{-k} + \varepsilon, \qquad 0 \le \varepsilon \le 2^{-k} - 2^{-n}, \quad k \le n, \tag{11}$$

be the n-bit auxiliary sequence vector at step k, then

$$1 - 2^{-2k} \le x(k).(1 + 2^{-k}) < 1 + 2^{-k}. \tag{12}$$

Proof

The right inequality is trivial, it corresponds to $\varepsilon = 0$. The left inequality is deduced from the computation of $x(k).(1 + 2^{-k})$ for ε maximum, i.e. $2^{-k} - 2^{-n}$.

The practical interpretation of (12) is the impact of rule (8) on $x(k+1)$ whenever $x(k)$ is less than one with a fractional part made up of a k-one string and a zero at position $k+1$. $x(k+1)$ will be either less than one, exhibiting a similar pattern with at least $2k$ one's as the header of the fractional part, or greater than one ($x_0(k+1) = 1$) with at least $k+1$ zero's as the header of the fractional part. In both cases, the target value $x(p) = 1$ is approximated by $x(k+1)$ with at least one bit more.

3. COMMENT

1 The selection (6) is justified by the fact that a decision about multiplying by $a_i .2^{-i} + 1$ (1) cannot be made before knowing the next bit. Actually, considering bit x_0 only (either 1 or 0) one cannot know whether the sequence $x(i)$ is already 1 (end of convergence process) or not.

2 When $x(i) > 1$, the strategy described by (7) consists in detecting the first non-zero bit of $x(i)$ then multiplying by $(-2^{-i} + 1)$. When $x(i) > 1$, lemma 1 shows that, at step i, bits $x_{-k > -i}$ (i) are all zero's.

3 When $x(i) < 1$, the strategy described by (8) consists in detecting the last non-zero bit of $x(i)$ then multiplying by $(2^{-i} + 1)$. When $x(i) \le 1$, lemma 2 shows that, at step i, bits $x_{-k > -i}$ (i) are all one's.

4. ALGORITHMS

4.1. Algorithm 1 - Logarithm computation by multiplicative normalization

The argument x is in [½, 2[: $x = x(0).x(1)\ x(2) \ldots x(n)$. Let $xx(i,j)$ be the component j of $xx(i) = xx(i,0).xx(i,1)\ xx(i,2) \ldots xx(i,n)$. Let $lut(i) = ln\ (1+a(i).2^{-i})$ read from the table.

a(0):= 0; c(0):= 1; xx(1):= x; yy(1):= 0;
for** i **in** 1 .. p-1 **loop

```
if xx(i)= 1 then exit; end if;
if xx(i)> 1 then a(i):= -xx(i,i) else a(i):= xx(i,i)*not(xx(i,i+1)); end if;
c(i):= 1+a(i)*2**(-i); xx(i+1):= xx(i)*c(i); yy(i+1):= yy(i)-lut(i);
end loop;
```

4.2. Algorithm 2 - Logarithm computation by multiplicative, one-shift and add, normalization

The argument x is in [½, 2[: $x = x(0).x(1)\ x(2) \ldots x(n)$. Let $xx(i,j)$ be the component j of $xx(i) = xx(i,0).xx(i,1)\ xx(i,2) \ldots xx(i,n)$. Let $lut(i) = ln\ (1+a(i).2^{-i})$ read from the table.

```
a(0):= 0; xx(1):= x; yy(1):= 0;
for i in 1 .. p-1 loop
    if xx(i)= 1 then exit; end if;
    if xx(i)> 1 then a(i):= -xx(i,i) else a(i):= xx(i,i)*not(xx(i,i+1)); end if;
    xx(i+1):= xx(i)+a(i)*xx(i)*2**(-i); yy(i+1):= yy(i)-lut(i);
    end loop;
```

4.3. Example 1

In the following example the auxiliary sequence x(i) is computed in the binary system, while, for readability, the sequence $y(i)$ is computed in decimal; the precision is then readily verified. The functional values $ln\ (1\pm2^{-i})$ are assumed given by look-up tables. x is in [1, 2[.

Let

$x = x(0) = x_0.\ x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5} = 1.10111 = (1{,}71875)_{10}$

$y(0) = 0$

Compute $ln\ (x)$ with precision $p = 8$

i	a_i	$c(i)$ $a_i.2^{-i} + 1$	$x\ (i+1)$ $x(i).c(i)$	$ln\ c(i)$	$y(i+1)$ $y(i) - ln\ c(i)$
-	-	-	$x(0) = 1.10111$	-	$y(0) = 0$
0	$a_0 = 0$	$0.2^{-0} + 1$ $c(0) = 1$	(1.10111).(1) $x(1) = 1.1011100$	0	0
1	$a_1 = -1$	$-2^{-1} + 1$ $c(1) = 0.1$	(1.1011100).(0.1) $x(2) = 0.11011100$	-0.69314718	0.69314718
2	$a_2 = 1$	$2^{-2} + 1$ $c(2) = 1.01$	(0.11011100).(1.01) $x(3) = 1.00010011$	0.223143551	0.470003628
3	$a_3 = 0$	$0.2^{-3} + 1$ $c(3) = 1$	(1.00010011).1 $x(4) = 1.00010011$	0	0.470003628
4	$a_4 = -1$	$-2^{-4} + 1$ $c(4) = 0.1111$	(1.00010011).(0.1111) $x(5) = 1.00000010$	-0.064538521	0.534542149
5	$a_5 = 0$	$0.2^{-5} + 1$ $c(5) = 1$	(1.00000010).1 $x(6) = 1.00000010$	0	0.534542149
6	$a_6 = 0$	$0.2^{-6} + 1$ $c(6) = 1$	(1.00000010).1 $x(7) = 1.00000010$	0	0.534542149
7	$a_7 = -1$	$-2^{-7} + 1$ $c(7) = 0.1111111$	(1.00000010).(0.1111111) $x(8) = 1$ (rounded up)	-0.007843177	0.542385326

The actual decimal value of *ln* (1.71875) is 0.541597282 ± 10^{-9}, the difference from the computed result is less than $8.10^{-4} < 2^{-10}$.
As it appears in the preceding example, whenever $a_i = 0$, the only effect of step *i* on the computation process consists in incrementing the step number; both sequences $x(i)$ and $y(i)$ remain unchanged. So, by detecting strings of 0 or 1 in $x(i)$, one could readily jump to the next non trivial computation step. The following example illustrates this feature.

4.4. Example 2

As in the preceding example 1, the auxiliary sequence x(*i*) is computed in the binary system, while sequence $y(i)$ is computed in decimal. The functional values $ln\ (1 \pm 2^{-i})$ are given by look-up tables. x is now in [½, 2[. Strings **00..** or **11..** are highlighted. The multiplications by (1 ± 2^{-i}) have been replaced by additions: $x(i) \pm x(i)*2^{-i}$.

Let

$x = x(0) = x_0.\ x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5} = 0.10011 = (0{,}59375)_{10}$

$y(0) = 0$

Compute *ln* (x) with precision $p = 10$

i	a_i	$c(i)$ $a_i.2^{-i} + 1$	$x\ (i+1)$ $x(i).c(i)$	***ln c*(*i*)**	$y(i+1)$ $y(i) - ln\ c(i)$
-	-	-	$x(0) = 0.1001100000$	-	$y(0) = 0$
0	$a_0 = 0$	$0.2^{-0} + 1$ $c(0) = 1$	$(0.10011).(1)$ $x(1) = 0.1001100000$	0	0
1	$a_1 = 1$	$2^{-1} + 1$ $c(1) = 1.1$	$(0.1001100000).(1.1)$ $x(2) = 0.1001100000$ $+ 0.0100110000$ $= 0.1\mathbf{11}0010000$	0.405465108	0.405465108
2	$a_2 = 0$	-	- $x(3) = x(2)$	-	-
3	$a_3 = 1$	$1.2^{-3} + 1$ $c(3) = 1.001$	$(0.111001).(1.001)$ $x(4) = 0.111001$ $+ 0.000111001$ $= 1.000\mathbf{00000}10$	0.117783035	0.523248143
4→8	$a_4 \rightarrow a_8$ $= 0$	-	- $x(9) = x(4)$	-	-
9	$a_9 = -1$	$- 2^{-9} + 1$ $c(9) =$ 0.111111111	(1.0000000010) $.(0.111111111)$ $x(10) = 1.0000000010$ $- 0.0000000010\ ..$ $= 1$ (rounded up)	- 0.001955035	0.521293108

The actual decimal value of *ln* (0.59375) is - 0.521296923 ± 10^{-9}, the difference from the computed result is less than $4.10^{-6} < 2^{-10}$.

4.5. Implementation schemes

4.5.1. Algorithm 1

Figure one displays a possible implementation scheme for algorithm 1. The auxiliary sequence is computed through successive multiplication by 1, $(1+2^{-i})$ or $(1-2^{-i})$ according to the values of a_i .

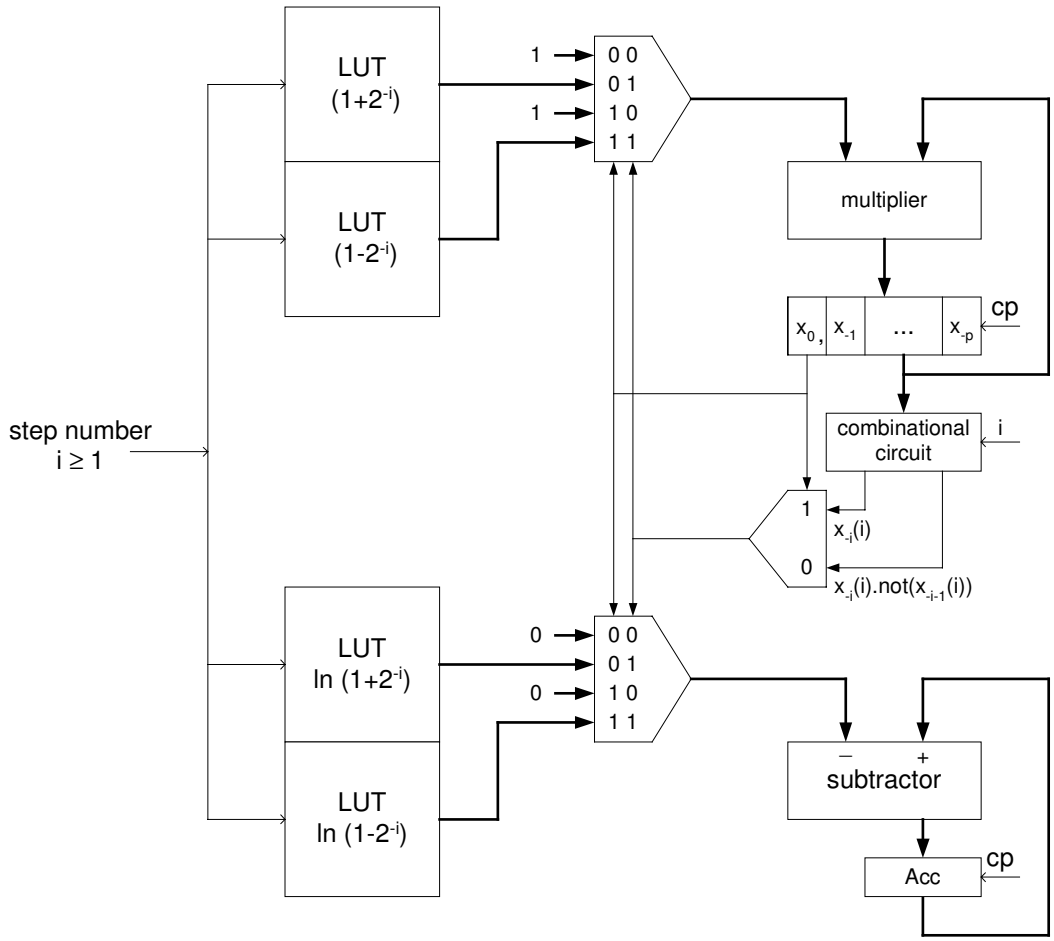

Figure 1. Logarithm computation circuit using multiplicative normalization

Figure 2 displays a possible implementation scheme for algorithm 2. The auxiliary sequence is computed through successive additions of 0, $x(i)*2^{-i}$ or $-x(i)*2^{-i}$, according to the values of a_i . Actually the algorithm materialized by figure two is a slight modification of algorithm 2, as follows.

4.5.2. Modified algorithm 2

a(0):= 0; xx(1):= x; yy(1):= 0;

```
for i in 1 .. p-1 loop
    if xx(i)= 1 then exit; end if;
    if xx(i)> 1 then a'(i):= xx(i,i); xx(i+1):= xx(i)-a'(i)*xx(i)*2**(-i);
    else a'(i):= xx(i,i)*not(xx(i,i+1)); xx(i+1):= xx(i)+a'(i)*xx(i)*2**(-i); end if;
    yy(i+1):= yy(i)-lut(i);
    end loop;
```

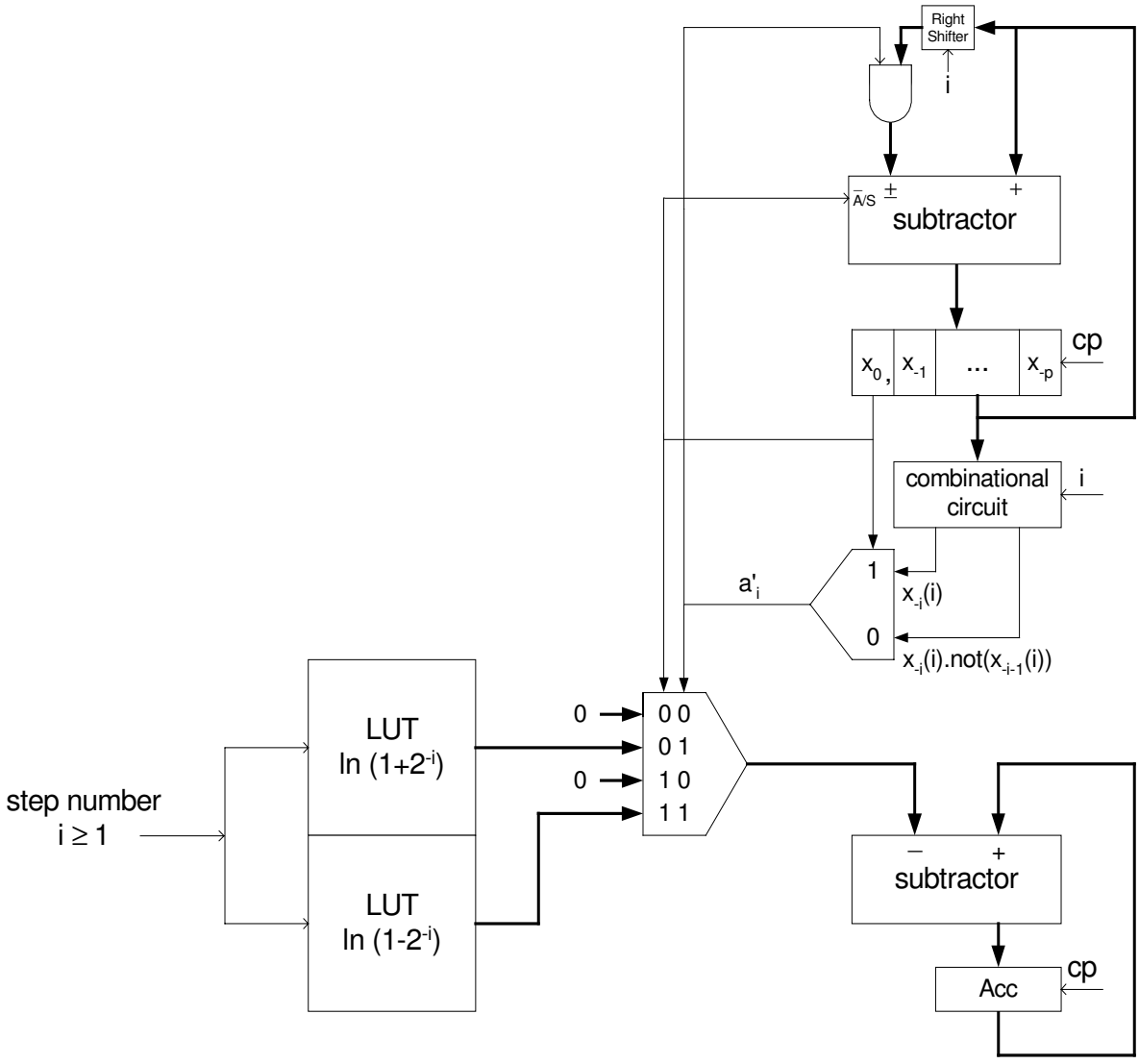

Figure 2. Logarithm computation circuit using a multiplicative normalization circuit made of a shifter and an adder/subtractor

5. IMPLEMENTATIONS ON FPGA XILINX VIRTEX IV

5.1. Algorithms 1 and 2

Both *ln*(x) algorithms presented above have been implemented on a 10752-slice Virtex IV FPGA [6]. Synthesis has been achieved using Xilinx Synthesis Technology-*XST*, [7], while physical implementation used Integrated System Environment (ISE), [8]. A comparative analysis is presented for 8-bit, 16-bit, 32-bit and 64-bit *ln*(x) precisions. The corresponding precision is provided by the respective LUT's.
For implementing algorithm 1, among the available synthesis options, DSP48 units [9] have been selected for efficiency purposes. DSP48 is a dedicated unit involving an 18-bit multiplier, a 3-input 48-bit adder together with the corresponding additional logic such as multiplexers. The aimed functional precision determines the required quantity *ND* of DSP48 unit, namely

$$ND = (N/16)^2, \qquad (13)$$

where *N* stands for *N*-bit precision.

Table 1 displays the comparative values of cycle time *T* (single-step time), operating frequency *F*, *ln*(*x*)-operation frequency *FOP*, number of required DSP's *ND*, and number of slices *NS* for 8-bit, 16-bit, 32-bit and 64-bit precision and for implementations of algorithms 1 and 2.

Table 1: Performances of algorithms 1 and 2 implemented on Xilinx Virtex IV (device xc4vlx25-12ff668)

N	**Algorithm 1 (multiplier)**						**Algorithm 2 (one-shift & Add)**				
	T *(ns)*	***F*** *(Mhz)*	***FOP*** *(Mhz)*	***ND***	***NS***		***T*** *(ns)*	***F*** *(Mhz)*	***FOP*** *(Mhz)*	***ND***	***NS***
8 bits	6.2	161	23.0	1	44 (0.4%)		4.5	222	31.746	-	52 (0.5%)
16 bits	6.7	149	9.95	1	92 (0.9%)		5.8	172	11.494	-	115 (1%)
32 bits	12.9	77	2.50	4	221 (2%)		7.2	138	4.480	-	290 (2.7%)
64 bits	20	50	0.793	16	639 (5.9%)		8.3	120	1.912	-	707(6.6%)

ln(x) operation frequency FOP is computed as

$$FOP = F/(N-1) \qquad (14)$$

Table 2 enhances the improvements of *FOP*'s for Algorithm 2 with respect to Algorithm 1 and the related increases of slice costs. It can be observed that the operating speed-up is paid by some additional slice cost but for $N = 32$ and $N = 64$, the overall performance is reached at a very reasonable cost. It can be observed that the improvements are more significant for higher values of *N*, due to the quadratic increase of needed DSP48's.

Table 2: Comparative *FOP* and slice costs of algorithms 1 and 2 implementations

N	***FOP*(2) / *FOP*(1)**	***NS*(2) / *NS*(1)**
8 bits	1.38	1.18
16 bits	1.15	1.25
32 bits	1.79	1.31
64 bits	2.41	1.11

5.2. Algorithm 3 - acceleration of algorithm 2.

An important feature of the *ln*(x) algorithms presented in section 4, is the possibility to skip steps according to eventual values zero of a_i's. As a matter of fact whenever $a_i = 0$, neither *c(i)* nor *y(i)* have to be modified. According to rules (7) and (8), one can skip as many steps (say *s*) between position *-i* and position *-i-s*, as allowed by the length of eventual 0-strings (within *x(i)*>1) or 1-strings (within *x(i)*<1). One can shows statistically that the average quantity of steps to be skipped, using this acceleration procedure, is superior to 50 % of what would be required by the straight *p*-step procedure for precision *p* calculation. Exhaustive tests have been handled up to 16-bit operands, while statistic experiments (for normalized [1,2[operands) exhibited Gaussian distributions. Table 3 displays the experimental average latency (***L***) in terms of the average numbers of required cycles for 8-bit, 16-bit, 32-bit and 64-bit operands. The drawback of step skipping is the rise of cycle length, due to the need of an additional circuit to set the skip length (*s*). Actually the hardware at hand plays a key role in additional costs and subsequent performance improvements. Table 4 displays the overall performances of the accelerated algorithm 2 while table 5 displays the comparative *FOP* and slice costs of algorithm 2 with respect to the accelerated version (algorithm 3).

Table3: Average latency for algorithm 3

N	***L***
8 bits	3.4
16 bits	7.1
32 bits	15.1
64 bits	31.1

Table 4: Performances of algorithm 3 implemented on Xilinx Virtex IV (device xc4vlx25-12ff668)

	Accelerated algorithm 2 (Algorithm 3)			
	T *(ns)*	***F*** *(Mhz)*	***FOP*** *(Mhz)*	***NS***
8 bits	6.4	156	45.955	59 (0.5%)
16 bits	8.9	112	15.825	147 (1.4%)
32 bits	11.5	86	5.758	397 (3.7%)
64 bits	16.3	61	1.978	1217 (11.3%)

Table 5: Comparative *FOP* and slice costs of algorithms 3 and 2 implementations.

N	***FOP*(3) / *FOP*(2)**	***NS*(3) / *NS*(2)**
8 bits	1.45	1.13
16 bits	1.37	1.27
32 bits	1.28	1.38
64 bits	1.03	1.72

Table 5 shows that the performances (*FOP*) are decreasing as *N* increases. This means that as *N* increases the additional step delay is overcoming the reduction in the number of steps. The *NS*

factor is directly related to the step complexity. In order to take a better profit of the acceleration technique, some optimization technique is needed and remains an open question.

6. CONCLUSION

Starting from a classical $ln(x)$ computation algorithm using convergence method with multiplicative normalization, some FPGA implementations have been carried out. The used FPGA device belongs to the Xilinx Virtex4 family.
On first noticed that the multiplication steps may be usefully replaced by a shift and add procedure using a shifter and an adder-subtractor. This alternative has proved to be faster and cheaper. To emphasize this point, one first implemented algorithm 1 (multiplication) using dedicated DSP48 multiplier cells embedded in the device at hand. Then the algorithm 2 (shift and add) has been implemented and provided up to 140 % performance improvements (*FOP*) - for 64-bit operand, while the hardware cost augmented by 11 % only. Finally the acceleration of the process (skipping trivial steps) has been taken into account to improve performances, but the reduction in number of steps appeared to be partially compensated by an additional delay generated by step length. So, the advantages vanish as N increases.

7. BIBLIOGRAPHY

[1] J. Cao, B.W. Wei, and J. Cheng, "High-Performance Architecture for Elementary Functions Generation," *Proc. 15th IEEE Symp. Computer Arithmetic,* pp.136-144, 2001.
[2] J-P. Deschamps, G. Bioul, and G. Sutter, *Synthesis of Arithmetic Circuits, FPGA, ASIC, and Embedded Systems,* John Wiley Interscience, New York 2006.
[3] M.D. Ercegovac, "FPGA Implementation of Polynomial Evaluation Algorithms," *Proc. of SPIE Photonics East '95 Conference,* Vol. 2607, pp177-188,1995.
[4] V. Paliouras, K. Karagianni, and T. Stouraitis, "A Floating-point Processor for Fast and Accurate Sine/Cosine Evaluation," *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal processing,* Vol. 47 n°5, pp. 441-451, May 2000.
[5] P.K. Tang, "Table Look-up Algorithms for Elementary Functions and their Error Analysis," *Proc. 10th IEEE Symp. Computer Arithmetic,* pp.232-236, 1991.
[6] Xilinx inc., Virtex-4 User Guide, http://www.xilinx.com, April 2007.
[7] Xilinx inc., XST User Guide-82i, http://www.xilinx.com, 2007.
[8] Xilinx inc., ISE 8.2 documentation, http://www.xilinx.com, 2007.
[9] Xilinx inc., Xtreme DSP for Virtex-4 FPGA's User Guide, http://www.xilinx.com, June 2007.