FPGA IMPLEMENTATION OF 64-BIT EXPONENTIAL FUNCTION FOR HPC

Ernest Jamro[1,2], Kazimierz Wiatr[1,2]

1. AGH University of Science and Technology,
al. Mickiewicza 30, 30-059 Kraków
email: jamro/wiatr@agh.edu.pl

Maciej Wielgosz[2]

2. ACK Cyfronet AGH,
ul. Nawojki 11, 30-950 Kraków
email: wielgosz@agh.edu.pl

ABSTRACT

Most presented implementations of the exponential function confine to the single precision format. Increasing data width to the double precision format requires a different approach. The presented novel architecture employs three independent Look-Up Tables (LUTs) together with a short Taylor expansion $exp(x) \approx 1+x$. Implementation results show that the double precision *exp()* function implementation achieves huge performance with satisfactory accuracy, latency and FPGA area consumption.

1. INTRODUCTION

Numeric analysis of computationally intensive physicians or chemical models (performed e.g. by Gaussian application) is a challenging task. It is worth to focus on an exponential operation since it appears frequently in many tasks and it is not directly supported by general-purpose processors. Performing a single *exp()* operation by a general-purpose processor consumes many clock cycles.

Nowadays, there are a few implementations of single precision floating-point *exp* operations in FPGAs [4,5,6]. Since the proposed *exp()* module aims to speed-up HPC chemical and physics calculations, it has to be compatible to the double precision data format. Up to the authors knowledge none double precision efficient implementation of exp() function has been published. The paper [1] presents double precision implementation of exp() function ,proposed algorithm occupies almost all resources of Virtex2 Pro (55 616 Slices).

The outline of this paper is as follows. Section 2 describes the previous work on the *exp* function and mathematical background for the proposed novel architecture. Section 3 presents the block diagram of the proposed module. As the multipliers significantly influence the implementation results, the optimized multiplier architectures are proposed in Section 4. Section 5 contains calculation error analysis. Section 6 shows implementation results.

2. CALCULATION ALGORITHMS

Functions evaluation methods can be separated into two groups: iterative [2] and non-iterative algorithms. Since iterative methods have longer computation latency, non-iterative methods are considered hereby. Non-iterative methods are classified as follows:
- direct Look-Up Table (LUT)
- polynomial approximations
- a mixture of LUT and polynomial approximation

To evaluate *exp* function the following commonly known mathematical identities are employed:

$$e^x = 2^{x \cdot \log_2 e} = 2^{x_i} \cdot e^{x - x_i / \log_2 e} \quad (1)$$

and

$$e^{x+y} = e^x \cdot e^y \quad (2)$$

where x_i is an integer part of $x \cdot log_2 e$.

Conversion to base *2* instead of *e* simplifies significantly evaluation of 2^x for integer part of x [6]. Consequently an input value x is separated into a fractional x_f and integer part x_i. Integer part x_i is fed almost directly to the exponent of the resultant word (64-bit word comprised of a sign, exponent and mantissa fields). Fractional part x_f is further processed to evaluate the mantissa. It should be noted that according to (2) the input argument can be split into two or more independent bit-sections (e.g. most and less significant bits parts) and each section can be evaluated separately. The final result is obtained by multiplying the partial results.

An alternative solution to the LUT-based method is the polynomial approximation which is commonly employed in hardware [3]. Usage of these methods for higher precision functions imposes a much higher degree of the approximating polynomial. Summing-up, for higher precision data (e.g. 64 bit) polynomial approximation methods loose their primary advantages.

The proposed architecture combines positive features of both above-mentioned techniques: LUT-based method and

1-4244-1060-6/07/$25.00 ©2007 IEEE.

polynomial approximations. Many implementations of mixed methods were introduced for the single precision format [4,5,6].

3. PROPOSED ARCHITECTURE

According to (2) the input argument can be split into several bit sections. These separate bit-sections can be calculated using either LUT-based methods or polynomial approximation. In the case of the LUT-based method, increasing the bit-width for each section reduces the number of multiplications but the size of the LUT memory increases rapidly. The analysis of the resources occupied by multipliers and LUT memories led to the following conclusions: Xilinx Virtex II provides 512×32 bit (9-bit address bus) BRAMs therefore 9-bit input LUT is an optimal solution.

For the LSBs, the LUT-based approach is inefficient and can be easily replaced by the polynomial approximation. According to Taylor-Maclaurin expansion:

$$e^x = 1 + x + x^2/2 + x^3/6 + \ldots. \quad (3)$$

In order to disregard $x^2/2$ and higher degree expressions, the input argument must be very small to satisfy maximum mantissa error $< 2^{-54}$ for double precision format. This is satisfied for $x<2^{-27}$. Consequently the most significant 27 bits of input x_f (fractional part of x) are calculated employing LUT-based methods, the reset less significant bits are calculated using Taylor-Maclaurin expansion limited to: $e^x \approx 1 + x$. The authors considered also limiting the expansion to: $e^x = 1 + x + x^2/2$, nevertheless it would require more hardware resources.

To conclude, the input argument x after conversion to fix-point format is divided into 5 sections:

1. integer part (11-bit), x_I, which evaluates 2^{Xi} (exponent),
2. fractional MSB part, x_M, bits $2^{-1} \div 2^{-9}$,
3. fractional miDdle part, x_D, bits $2^{-10} \div 2^{-18}$,
4. fractional LSB part, x_L, bits $2^{-19} \div 2^{-27}$,
5. fractional Taylor part, x_T, bits $2^{-28} \div 2^{-60}$

Summing up, the following mathematical operations are employed:

$$x_I = \lfloor x \cdot \log_2(e) \rfloor \quad (4)$$

$$x_F = x - x_I \cdot (\log_2(e))^{-1} = x_M \,\&\, x_D \,\&\, x_L \,\&\, x_T \quad (5)$$

$$y = 2^{x_I} \cdot e^{x_M} \cdot e^{x_D} \cdot e^{x_L} \cdot (1 + x_T) \quad (6)$$

where: & is bit concatenation, $\lfloor x \rfloor$ is rounding to the greatest integer x_I satisfying $x_I \leq x$.

Employing (1), (4) and (5) enables separation of the integer and fractional part. In should be noted that the fractional part is obtained as the result of two multiplications (see (4), (5)). Multiplication by the constant $\log_2(e)$ (4) is carried out with lower precision as it is described in Section 5. Furthermore, for the second multiplication by $1/\log_2(e)$ (5), the x_I is a small integer (11-bit). Consequently, this approach, replaces a large multiplier (required by the identity $e^x = 2^{X/\ln(2)}$) with two smaller multipliers. This results in significant decrease of area consumption.

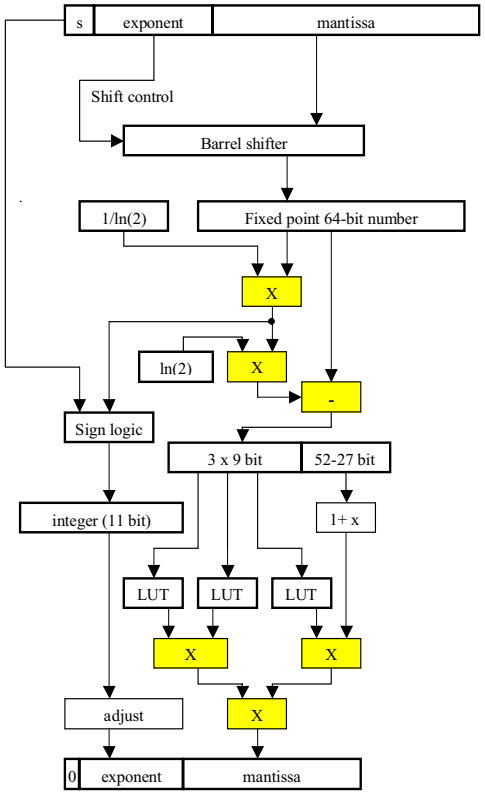

Fig. 1. Architecture of the proposed *exp()* module

The proposed *exp()* module is presented in Fig. 1 and comprises the following modules:

- exceptional states (*inf*, *NaN*) detection logic for input data; this unit also converts input data to internal fixed point standard (barrel shifter)
- exponent evaluation module which separates fractional and integer part (which corresponds to exponent field of the result); this module includes also sign migration from fractional to integer part (described below)
- LUTs which store fractional elementary values of *exp()*, polynomial approximation and multipliers
- the internal format to IEEE-754 standard conversion logic.

It should be noted how negative numbers are processed. Negative input argument $-x$ is usually split into integer part $-x_i$ and fractional part $-x_f$. This causes that the fractional part LUTs processes also negative numbers, which results in one extra address bit of the LUT reserved for the sign. A

better solution is obtained when negative numbers are constrained only to integer part (most significant bits part) [7]. Therefore for $x_f \neq 0$ and $x<0$ the following mathematical identify is employed:

$$-x = -x_i - x_f = -(x_i+1) + (1-x_f) \quad (7)$$

It should be noted that the above identity is obtained by converting input argument from the sign-and-magnitude format (floating point mantissa standard) into the two's complement format.

4. IMPROVED MULTIPLIERS ARCHITECTURE

A the most FPGA area in the proposed architecture is occupied by multipliers. Consequently, the standard multipliers architecture was improved by the followings:

a) employing constant coefficient multiplier for multiplying by constant values $log_2(e)$, and $(log_2(e))^{-1}$.
b) reducing the input width by exploiting the fact that some of the input bits are fixed to zero,
c) reduced arithmetic width

4.1. Constant multipliers

The multiplier by the constant value $log_2(e) = 1/ln(2)$ is employed to obtain the exponent of the result (see (1)). This constant coefficient multiplier uses Canonic Sign Digit (CSD) representation to limit the number of non-zero digit. To further reduce hardware resources, reduced-width multipliers [7] are employed.

4.2. Fixed to zero input bits

It can be noticed that some bits of the LUT output data are fixed to zero. These zeros bits are in the middle-part of the input arguments which further feed the multipliers. Therefore, a standard multiplier is replaced by a shorter-width multipliers and an adder. The following formulas illustrate the multiplier improvement:

$$a.000000x = a + x \quad (8)$$
$$1.000000y = 1 + y \quad (9)$$

$$(a+x)\cdot(1+y) = a + a\cdot y + x + x\cdot y \quad (10)$$

4.3. Reduced arithmetic width

Multiplier inputs are roughly 60-bit wide, therefore the product width is 120-bit wide. Such a bit-width is far beyond the required precision therefore the LSBs of the product are usually disregarded. As a result, in the proposed architecture, some of the LSBs logic is not implemented at all. Similar architecture was presented in [8]. The proposed architecture however does not use any advance error compensation logic. Instead, additional guard-bits are added which limit the maximum bit-width truncation error.

5. ERROR ANALYSIS

In the proposed architecture the following sources of errors can be distinguished:
1. Taylor series expansion,
2. multiplier (and LUT) width limitation

1) The Taylor series expansion is limited only to: $e^x \approx 1 + x$. For $x<<1$ the expansion error can be approximated by the next omitted expression, i.e. $x^2/2$. As input argument $x_T<2^{-27}$ the Taylor series expansion error is limited roughly by 2^{-55}. It should be noted that the input value $x_T \geq 0$, thus the result $y_T = 1+x_T \geq 1$. Consequently relative error is also $\leq 2^{-55}$. Summing up, Taylor series expansion error is much lower than the double precision format accuracy.

2) In order to reduce hardware requirements for multipliers, the part of the LSBs are not implemented. This results in additional calculation error. This error is limited by the number of carry paths which are cut. To limit this error additional guard bits are introduced.

5.1. Multiplier by *1/ln2*

In order to obtain integer part x_i the input x is multiplied by the constant $1/ln(2)$. This multiplication does not require a full precision arithmetic, the multiplication error is corrected by the circuit itself in a certain range.

The correction range (maximum allowed error) is limited in the presented architecture by the number of input bits to the LUT memory. In order no to increase the number of input bits the x_f should satisfy $x_f < 1$. Assuming the multiplication error is zero, then the following is satisfied: $x_f < ln(2)$. Consequently, x_f error should be less than $1-ln(2)$. Therefore the multiplier by the constant $1/ln(2)$ should not exceed the maximum error equal $(1-ln2)/ln2 = 0.442$.

6. IMPLEMENTATION RESULTS

The proposed exp() module was implemented on SGI RASC (Reconfigurable Application Specific Computing) [9] as a sub-element on a HPC system (e.g. SGI Altix 4700).

Four implementation versions are considered:
1) standard multiplier with build-in 18×18-bit multipliers
2) standard multiplier without build-in 18×18-bit multipliers
3) optimized multipliers

4) whole RASC system with external memory interface and RASC control and the single *exp* module.

Table 1. Implementation results for Virtex-4 LX200

implementation	# 4-input LUTs	# flip-flops	# 18-Kb BRAMs	DSP48
1) With DSP48	1293 (0.73%)	105 (0.06%)	6 (1.8%)	71 (74%)
2) Without DSP48	13375 (7.5%)	105 (0.06%)	6 (1.8%)	0
3) optimized multipliers	5025 (3%)	5223 (3%)	6 (1.8%)	0
4) RASC system	14,521 (8%)	20,125 (11%)	29 (8%)	0
5) Single precision (Virtex-2 1000) [6]	1896 (1.044%)	1896 (1.036%)	0	0

Implementation results conclude that multipliers are the most hardware consuming part of the module and introduce the longest latency, therefore the dedicated speed optimized multipliers were designed.

It is worth to notice the large difference between resources absorbed by the standard and optimized multipliers versions. Large discrepancy in the number of flip-flops is due to pipeline mechanism employed by the module with optimized multipliers. The module achieves latency of 27 clock cycles.

Table 2. Synthesis results depending on the number guard bits

# Guard bits	Accuracy (Mean)	max. Freq [MHz]	# 4 input LUTs
0	0.5798	166.425	4584
1	0.5214	165.276	4732
2	0.4825	164.143	4823
3	0.4708	163.025	4925
4	0.4708	161.922	5023
5	0.4708	160.834	5158
6	0.463	159.760	5272
7	0.4591	158.701	5371
8	0.4591	157.656	5487

The calculation error for the designed module can be parameterized by the number of guard bits as it is presented in Tab. 2. Increase of the number of guard bits up to 4 seems to be reasonable, since the result accuracy (Mean ABS(error)) drops below 0.5 ULP and resources consumption is still not so high. The further increase of the guard-bits number results in a little accuracy improvement but rises significant the absorbed FPGA slices.

7. CONCLUSIONS

This paper describes a novel architecture of double precision exponential function implemented in FPGAs and SGI RASC platform. The presented module consumes only roughly 3% of Virtex4 4 LX200. Consequently, in a single FPGA as many as 20-30 parallel *exp* units can be implemented. Summing-up the double precision *exp* calculation performance can be significantly accelerated by FPGA and RASC platforms.

The presented *exp* architecture introduces several novel hardware solutions never used for *exp* function: a) 3 independent LUTs and Taylor series expansion for *exp* function, b) sign-migration to integer part, c) optimized multipliers. It should be noted that up to the authors knowledge, none efficient FPGA implementation of double precision *exp* function has been published.

8. REFERENCES

[1] Stanek S., Benjegerdes T., *Reconfigurable Computing for High Performance Technical Computing,* Scalable Computing Lab, Ames Laboratory, Ames, IA 50010

[2] Omondi A.R. *Computer Arithmetic Systems* Prentice Hall. Cambridge, 1994

[3] Detrey J., de Dinechin F, *Table-based polynomials for fast hardware function evaluation*, 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'05), Samos, Greece, July 2005, pp. 328-333.

[4] Doss C.C., Riley R.L., Jr., *FPGA-Based Implementation of a Robust IEEE-754 Exponential Unit*, 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), pp. 229- 238

[5] Bui H.T., Tahar S., *Design and Synthesis of an IEEE-754 Exponential Function*, 1999 IEEE Canadian Conference on Electrical and Computer Engineering Shaw Conference Center, Edmonton, Alberta, Canada May 9-12 1999, pp. 450-455 vol.1

[6] Detrey J., de Dinechin F., *A parameterized foating-point exponential function for FPGAs,* IEEE International Conference on Field-Programmable Technology (FPT'05), Singapore, December 2005, pp. 27-34.

[7] Wiatr K., Jamro E. *Constant Coefficient Multiplication in FPGA Structures*, Proc. of the IEEE Int. Conf. Euromicro, Maastricht, The Netherlands, Sep. 5-7, 2000, Vol. I, pp. 252-259.

[8] Parhi K. K., Chung J.G., Lee K.C., Cho K.J. *Low-error fixed-width modified booth multiplier*, US Patent: 6957244.

[9] SGI *Reconfigurable Application Specific Computing User's Guide* SGI RASC v2.1 Jan. 2007.