A Synthesis Method of General Floating-Point Arithmetic Units by Aligned Partition

Liangwei Ge[1], Song Chen[1], Yuichi Nakamura[2], Takeshi Yoshimura[1]

ABSTRACT

Floating-point arithmetic units (FPU) have paramount importance in applications that involve intensive mathematic operations. However, previous implementations of FPU either require much manual work or only support special functions (e.g. reciprocal, square root, logarithm, etc.). In this paper, we present an automatic method to synthesize general FPU by aligned partition. Based on the novel partition algorithm, our method supports functions of wide, irreducible domain. The synthesized FPU achieves smaller area, higher frequency, and greater accuracy. Experimental results show that our method obtains 1) on average 90% smaller and 2.1 times faster indexer than the conventional automatic method; 2) on the hyperbolic functions, 20k times smaller error rate and 50% use of LUTs and flip-flops than the conventional manual design.

Keywords

Synthesis, floating-point arithmetic, polynomial approximation.

1. INTRODUCTION

Many applications involve intensive floating-point arithmetic. The traditional software emulation, like FdLibM [1], is slow and consumes lots of the CPU/DSP computing power. As the scaling technology constantly reduces the transistor cost, there is a growing demand to evaluate arithmetic functions by floating-point unit (FPU). However, the use of FPU has not been very successful in some hardware systems either because of the implementation difficulty [14] or the unsatisfactory FPU quality (larger than the fixed-point unit, low throughput [2][3], the limited function types supported [4]-[12], etc.).

In this paper, we present an automatic method to synthesize FPU of general functions based on aligned partition. The contributions of our works are as follows:

- An automatic FPU synthesis method for general functions of wide input domain, which generates compact, fast, and high throughput (pipelined) implementation;
- The proposal of *zones* that facilitate the grouping of floating-point numbers;
- The aligned domain partition algorithm that handles excessive floating-point segment boundaries at high speed and low hardware cost;

The rest of the paper is organized as follows: Section 2 summarizes previous works. Section 3 explains the proposed method in detail. And Section 4 shows the experimental results.

2. SUMMARY OF PREVIOUS WORKS

Currently, FPGA manufacturers start to provide IP cores to synthesize pipelined/serial arithmetic units using CORDIC algorithm [15]. However, these IP cores only support fixed-point units of limited function types. For floating-point units or ASIC designs, a lot of manual work is required.

Piecewise polynomial approximation [16][17] based synthesis methods have simple architecture, which ensures fast, pipelined designs [4]-[12]. It works as follows: assume the domain X of function $f(x)$ is partitioned into k segments and $f(x)$ is approximated by an mth-order polynomial in each segment:

$$P_j(x) = \sum_{i=0}^{m} c[j][i] \times x^i, 0 \le j \le k-1 \qquad (1)$$

Then, $f(x)$ can be evaluated as Fig. 1 shows: 1) for any $x \in X$, the *indexer* decides the segment j that x falls in; 2) retrieve the $(m + 1)$ polynomial coefficients of segment j from memory; 3) evaluate $P(x)$ by adders and multipliers according to Eq. (1).

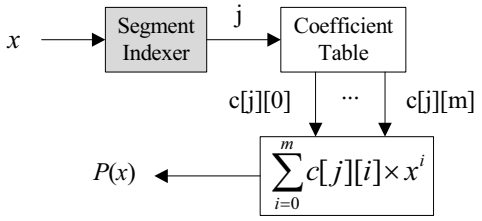


Fig. 1 Architecture of piecewise polynomial approximation.

The challenge in piecewise polynomial approximation is designing a compact, pipelined indexer that compares given x with excessive segment boundaries at high speed. Previous methods [4]-[12] assume the boundaries to be fixed-point numbers and use fixed-point adders and multipliers to implement $P(x)$, which have very limited dynamic. Therefore, they only support functions of narrow, reducible domain as Table 1 shows. These functions can have the mantissa and exponent calculated separately [13] (a brief introduction of the IEEE 754 floating-point number is given in Section 4.2). Therefore, domain X is generally reduced to the mantissa $1.M$ that has a range of [1, 2). In [9][11], Sasao further improved the indexer implementation by using less wires but more memory for the FPGA platform.

Table 1 Example of special functions with reducible domain

$f(x)$	Calculation method	Reduced domain
$1/x$	$\frac{1}{(-1)^S \times 1.M \times 2^E} = (-1)^S \frac{1}{1.M} 2^{-E}$	$1.M \in [1,2)$
$\log_2(x)$	$\log_2(1.M \times 2^E) = \log_2(1.M) + E$	$1.M \in [1,2)$
$\sqrt{x}$	$\sqrt{1.M \times 2^E} = \sqrt{1.M} \times 2^{E/2}$, if E even $= \sqrt{2 \times 1.M} \times 2^{(E-1)/2}$, if E odd	$1.M \in [1,2)$ or $2 \times 1.\mathrm{M} \in [2,4)$

In this paper, we extend the piecewise polynomial approximation [4]-[12] by supporting functions with wide, irreducible domain, like *tanh*(x), *sigmoid*(x), etc. We systematically analyze the placement

The authors are with:
1. The Graduate School of IPS, Waseda University.
Yoshimura Lab, 2-7 Hibikino Wakamatsu-ku
Kitakyushu 808-0135, Japan.
Tel & Fax: +81-93-692-5364. Email: liangwei_ge@ruri.waseda.jp
2. NEC Central Research Lab. Kawasaki-shi 216-8555, Japan.

of segment boundaries and its impact on the implementation. The proposed aligned partition is capable of handling excessive (2,000) floating-point boundaries at higher speed (2.1 times faster) and lower hardware cost (90% smaller).

3. ZONE-BASED ALIGNED PARTITION

Partition of domain X into segments is critical in piecewise polynomial approximation. Traditional partitions [4]-[8] generate width-fixed segments (uniform partition), which produce unnecessary segments. The recent non-uniform partitions [9]-[12] reduce segments by supporting variable segment width. Fewer segments mean smaller coefficient tables. However, over-reducing segments will greatly complicate the indexer. In this study we systematically analyze the domain partition and its impact on hardware. The resultant *zone*-based, *aligned* partition guarantees a high-speed, fully pipelined implementation that handles excessive floating-point segment boundaries at low hardware cost.

3.1 Segment Boundary Placement and Its Impact on Hardware

Definition 3.1 (partition). *The partition of domain $X = [x_{min}, x_{max}]$ is a set of segments: $PT=\{[x_0, x_0`], [x_1, x_1`], ..., [x_{k-1}, x_{k-1}`]\}$, where $x_0 = x_{min}$, $x_{k-1}` = x_{max}$, and $x_j \le x_j`$ for $0 \le j < k$. For segment $[x_j, x_j`]$, denoted by sg_j, $x_{j-1}` < x_j$, where $x_{j-1}`$ and x_j are two adjacent numbers in X.*

Definition 3.2 (segment indexer). *The indexer is an integer function of the input $x \in X$ and the partition $PT = \{[x_0, x_0`], ..., [x_{k-1}, x_{k-1}`]\}$. indexer(x, PT)=$\sum_{j=0}^{k-1} j \times T_j(x)$, where $T_j(x)$ is a Boolean function that decides whether x falls in sg_j. If $x \in [x_j, x_j`]$, $T_j(x) = 1$; else $T_j(x) = 0$.*

As Definition 3.2 shows, the indexer strongly depends on the partition *PT*. An indexer of k segments can be implemented by the Verilog code of Fig. 2 in combinational circuit, where $t = \lceil log_2 k \rceil$ is the bit-width of the segment index. To support pipelined processing, Fig. 2 adopts a fast, fully paralleled architecture that calculates index j for given x within one clock cycle. The implementation cost can be roughly estimated as follows:

$$size(\text{indexer}) < \sum_{j=0}^{k-1} \left[size\left(T_j(x)\right) + \text{wired signal of } j \right] + (k \cdot log_2 k)\ \text{OR gates} + (k \cdot log_2 k)\text{AND gates}$$

```
assign indexer[t-1:0] =
  {t{T0(x)}}  & t'd0     |
  {t{T1(x)}}  & t'd1     |
      ..........
  {t{Tk-2(x)}} & t'd(k-2) |
  {t{Tk-1(x)}} & t'd(k-1) ;
```

Fig. 2 RTL (Verilog) implementation of the segment indexer.

Usually, the segment number k is not extremely large (within 2k). The size of the indexer is mainly decided by $T_j(x)$. Thus, optimizing $T_j(x)$ is more effective than minimizing k to simplify the indexer.

Definition 3.3 (*L*-bit aligned segment). *A segment sg_j is L-bit aligned when (1) sg_j contains 2^L consecutive numbers; (2) for the 2^L numbers, the least significant L bits change from 00...0 (L-bit wide) to 11...1 (L-bit wide) and the higher bits remain constant.*

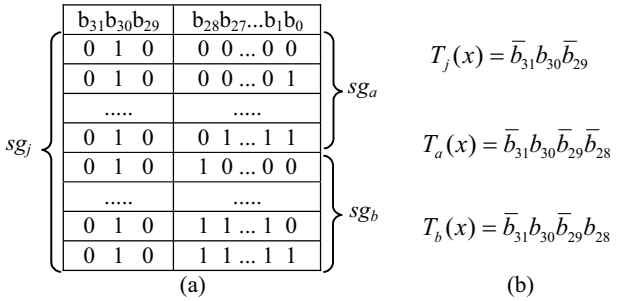
Theorem 3.1: *For any segment sg_j that contains 2^L numbers, the corresponding $T_j(x)$ has the simplest implementation when sg_j is L-bit aligned.*

Generally, two comparators and one AND gate are needed to decide whether a given x falls in a segment $sg_j = [x_j, x_j`]$: $T_j(x) = (x_j \le x)$ AND $(x \le x_j`)$. Theorem 3.1 shows how to simplify $T_j(x)$. By making sg_j aligned, $T_j(x)$ becomes independent of the least significant L bits of x.

In Fig. 3, the most significant 3 bits, $b_{31}b_{30}b_{29}$, of all the 2^{29} numbers in segment sg_j are constantly '010'. Thus, $T_j(x)$ is simplified to one product term of '$\bar{b}_{31}$ AND b_{30} AND $\bar{b}_{29}$', which requires only two AND gates without any comparator.

Theorem 3.2: *Assume an L-bit aligned segment sg_j is partitioned into two segments $\{sg_a, sg_b\}$. $T_a(x)$ and $T_b(x)$ have minimum hardware cost when sg_a and sg_b are two (L-1)-bit aligned segments.*

Fig. 3(a) shows a 29-bit aligned segment sg_j partitioned into two 28-bit aligned segments: sg_a and sg_b. Either $T_a(x)$ or $T_b(x)$ can be implemented by three AND gates. Any other partition of sg_j will greatly complicate both $T_a(x)$ and $T_b(x)$.



$$T_j(x) = \bar{b}_{31} b_{30} \bar{b}_{29}$$

$$T_a(x) = \bar{b}_{31} b_{30} \bar{b}_{29} \bar{b}_{28}$$

$$T_b(x) = \bar{b}_{31} b_{30} \bar{b}_{29} b_{28}$$

Fig. 3 Example of aligned segments. (a) sg_j is 29-bit aligned, sg_a and sg_b are 28-bit aligned; (b) corresponding $T_j(x)$, $T_a(x)$, and $T_b(x)$.

3.2 Grouping Floating-Point Numbers into Zones

The IEEE 754 single-precision floating-point number [18] has 32 bits: $b_{31}b_{30}...b_0$, where $S = b_{31}$ is the sign, $E = b_{30}...b_{23}$ the exponent, and $M = b_{22}...b_0$ the mantissa, which is a 23-bit fixed-point number with the point placed before b_{22}. For discussion convenience, we assume *normalized* numbers [18] (there is always a default '1' before the point). Thus, number x has a dynamic range of ($2^{-126} \le |x| < 2^{128}$) with its value given by: $x = (-1)^S \times (1.M) \times 2^{E-127}$.

Definition 3.4 (zone). *A zone, denoted by Z(S, E), is a set of 2^{23} consecutive floating-point numbers, which have the same sign ($b_{31} = S$) and the same exponent ($b_{30}...b_{23} = E$).*

Fig. 4 shows the grouping of all positive floating-point numbers into 254 zones. Each horizontal line represents a zone. The dots on a line stand for floating-point numbers with the same exponent. $Z(0, E)$ contains 2^{23} consecutive numbers:

$$\underbrace{1.00\cdots0}_{23\text{ bits}} \times 2^{E-127} \le x \le \underbrace{1.11\cdots1}_{23\text{ bits}} \times 2^{E-127}$$

Note that the 2^{23} floating-point numbers in a zone are virtually 2^{23} evenly spaced, fixed-point numbers, since their points are aligned at

the same digit. Therefore, the *zone* is a kind of bridge between floating-point and fixed-point numbers. Zones around zero have smaller rounding error (up to $2^{E-127-24}$). As E grows, $Z(0, E)$ becomes wider and the numbers become sparser. Precision is thus sacrificed for range.

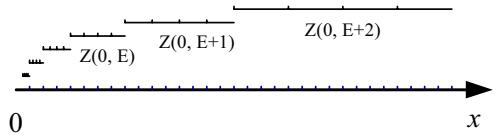


Fig. 4 Grouping floating-point numbers into zones.

Apparently, $Z(S, E)$ is a 23-bit aligned segment. This means $T_f(x)$ of $Z(S, E)$ depends only on the sign and exponent, which can be implemented by just eight AND gates without any 32-bit floating-point comparator.

3.3 Zone-Based, Aligned Partition

Algorithm 1: *Aligned non-uniform partition*

Input	$f(x)$, $X = [x_{min}, x_{max}]$, and the acceptable absolute error $e(x)$ defined on domain X
Output	Partition $PT = \{[x_0, x_0`], [x_1, x_1`], ..., [x_{k-1}, x_{k-1}`]\}$, where $x_0 = x_{min}$, $x_{k-1}` = x_{max}$

1. let z_L be the zone that x_{min} falls in: $z_L = zone(x_{min})$;
2. let $z_R = z_L$;
3. while($|P(x) - f(x)| \le e(x)$ over interval $\bigcup_{z=z_L}^{z_R} z$)
4. let z_R be the zone right of z_R: $z_R = right(z_R)$;
5. if($z_R \neq z_L$)
6. $z_R = left(z_R)$;
7. $\bigcup_{z=z_L}^{z_R} z$ forms an integral segment;
8. else
9. partition zone z_R into two aligned segments;
10. while($|P(x) - f(x)| > e(x)$ exists in any segment)
11. partition it into two aligned segments;
12. if($z_R = zone(x_{max})$)
13. exit; //partition finished
14. else
15. $z_L = z_R = right(z_R)$;
16. go to step 3;

As Fig. 4 shows, single-precision floating-point numbers on the entire X-axis, $(-2^{128}, 2^{128})$, can be grouped into 508 zones. Since each zone is 23-bit aligned, the partition of domain X into segments can be done by merging and partitioning these aligned zones.

We classify the segments into two types: 1) integral segment that spans integer number of zones; 2) fractional segment that is a fraction of a zone. If $P(x)$ well approximates $f(x)$ over some zones, these zones are merged into an integral segment. Since the corresponding $T_f(x)$ is independent of the *mantissa*, it has compact implementation. If $P(x)$ cannot well approximate $f(x)$ over a single zone, the zone is partitioned into fractional segments according to Theorem 3.2.

Algorithm 1 shows the zone-based, aligned partition algorithm. The approximating polynomial $P(x)$ is obtained by Chebyshev interpolation [16][17]. Function $zone(x)$ returns the zone that x falls in. $right(z)$/$left(z)$ returns the adjacent zone on the right/left side of zone z along the X-axis.

4. EXPERIMENT

For implementation convenience, we carried out the experiment on FPGA platform. The automatically synthesized FPUs (in Verilog) are evaluated by Xilinx ISE 9.1i [15] on a Spartan 3 xc3s4000 device.

4.1 Partition: Aligned vs. Least-Segment

Experiment 1 compares the aligned partition with Sasao's least-segment partition [12]. The *'Reduced domain X'* in Table 2 is the actual domain to be partitioned. Apparently, functions in Table 2 have wide input domains, which cannot be implemented by the traditional piecewise polynomial approximation [4]-[12]. Both partitions are based on the 2nd-order Chebyshev approximation [16][17]. Absolute error e_a is set at 2^{-22} over X.

Table 2 Memory usage under two partition methods ($e_a \le 2^{-22}$)

f(x)	Original domain	Reduced domain X	Aligned			Least-segment [12]		
			Seg #	Mem bits	Mem depth	Seg #	Mem bits	Mem depth
tanh(x)	(-∞, ∞)	[0, ∞)	113	7232	128	92	5888	128
sech(x)	(-∞, ∞)	[0, ∞)	142	9088	256	132	8448	256
sigmoid(x)	(-∞, ∞)	(-∞, ∞)	181	11584	256	136	8704	256

1. The aligned partition program is run by a Pentium 4 CPU of 2.4 GHz with runtime of less than 10 seconds.
2. ∞ denotes the largest number (around 3.4×10^{38}) the single-precision floating-point number can represent.

As expected, the least-segment partition [12] generates slightly fewer segments, because it does not consider the alignment constraint. The aligned partition on average requires 20% more memory bits. For ASIC design, these extra memories (about 1.6 kb) are not a problem for modern manufacturing technology. On the FPGA platform, both methods virtually use the same amount of memories, since the memory depth is $2^{log_2 k}$ not the segment number k. Therefore, three block RAMs are used to respectively store the 0th, 1st, and 2nd order coefficients (c_0, c_1, c_2). Table 2 verifies that the aligned partition is almost as good as the least-segment partition on reducing memory usage. The constraint on the placement of segment boundaries is weak, which produces acceptable number of segments of nearly arbitrary width.

Table 3 Indexers under two partition methods ($e_a \le 2^{-22}$)

f(x)	Aligned		Least-segment [12]	
	Gate #	Freq. MHz	Gate #	Freq. MHz
tanh(x)	462	136	8847	53
sech(x)	870	118	12186	51
sigmoid(x)	912	116	8313	72
Average	748	123	9782	58

* The 'Gate #' is the 'equivalent gate count' estimated by Xilinx ISE.

Table 3 compares the indexers under two partition methods. On average, the aligned partition achieves 90% smaller (by 9k gates) and 2.1 times faster indexers. The least-segment partition generates some indexers as slow as around 50 MHz, which have little practical use and become the FPU bottleneck. Table 3 justifies aligning segment boundaries during the domain partitioning, which makes $T_f(x)$ independent of the least significant bits of x. The resultant indexer is simpler and faster.

4.2 Comparison with Manual Design

Hyperbolic functions, like *tanh*(x) and *sigmoid*(x), have important use in neural networks. However, traditional automatic methods [4]-[12] do not support them well due to the wide, irreducible domain. In [14], Savich et al. manually implemented *sigmoid*(x) in various number formats on FPGA, which serves as a reliable baseline for our automatic synthesis tool.

$$sigmoid(x) \approx \begin{cases} 0, & \text{if } x \le -8 \\ (8-|x|)/64, & \text{if } -8 < x \le -1.6 \\ x/4 + 0.5, & \text{if } |x| < 1.6 \\ 1-(8-|x|)/64, & \text{if } 1.6 \le x < 8 \\ 1, & \text{if } x \ge 8 \end{cases} \quad (2)$$

[14] uses five-segment linear approximation, as Eq. (2) shows. Coefficients of the 1st-order terms are restricted to powers of two (1/64 and 1/4), thus multiplication is '*simplified*' to bit shift (in fixed-point) or exponent addition (in floating-point). Such an implementation has three problems: 1) *high cost*. The segment boundaries (± 1.6) are not aligned. Comparators are needed to implement the indexer; 2) *low accuracy*. Restriction on the 1st-order coefficients increases the error; 3) *poor systematic tradeoff*. Eliminating the multiplier does not reduce cost. Any segment with a different 1st-order coefficient will need specific bit shift or exponent addition, making the area proportional to the segment number.

In Table 4, *FXD*/*FLT* denotes the fixed/floating point implementation of *sigmoid*(x) in [14]. '*a0_8/a1_18*' stands for the floating-point implementation based on the '0th/1st'-order aligned partition with the acceptable absolute error $e(x)$ set at '$2^{-8}/2^{-18}$'.

'*a0_8*' is the correct way to eliminate the multiplier. It only consists of an indexer and the 0th-order coefficient table. '*a0_8*' has a max error of 0.0039, which is 1/20 of *FXD/FLT*. Moreover, it needs no flip-flop and requires even less LUTs than the fixed-point implementation *FXD*. The ability to handle excessive (168) segments enables '*a0_8*', a look-up table (0th-order approx.), to outperform the linear (1st-order) approximation of [14].

'*a1_18*' is based on linear approximation. Though one adder and one multiplier are required, they are shared by all the 473 segments. Hence, '*a1_18*' uses flip-flops and LUTs about half of *FLT*. Moreover, '*a1_18*' has a maximum error of 0.000004, which is about 20k times smaller than that of *FLT*.

Table 4 Different implementations of *sigmoid*(x)

Imp.	Input domain	Poly. order	Seg #	Design time	Max e_a	Flip flops	LUTs	Block RAMs	Freq. MHz
FXD	(-32, 32)	1	5	5 h	0.068	34	109	0/96	122
FLT	(-∞, ∞)	1	5	3 h	0.068	1386	2246	0/96	126
a0_8	(-∞, ∞)	0	168	5 m	0.0039	0	95	1/96	138
a1_18	(-∞, ∞)	1	473	5 m	4×10^{-6}	658	1334	2/96	118

Table 4 proved the ability of the aligned partition to handle excessive segments at low cost and high speed. The support of excessive segments greatly expands the domain X. Thus, the proposed method is able to synthesize more general functions with wide input domain.

It should be emphasized that our synthesis tool is highly automatic. FPUs are generated and verified in minutes. The implementations of [14] nevertheless require lots of manual work. In this experiment, *FXD* and *FLT* are designed and verified by an experienced RTL designer in 5 hours and 3 hours respectively.

A slight defect of piecewise polynomial approximation is the memory requirement to store polynomial coefficients. The memory can be easily implemented by register, ROM, RAM, combinational circuit, etc., which is not a problem for modern manufacturing technology. Since Spartan3 xc3s4000 provides abundant (96) block RAMs, we store the coefficients in block RAMs.

5. REFERENCES

[1] FdLibM: C math library for machines that support IEEE 754 floating-point, Available: http://www.netlib.org/fdlibm/

[2] Ray Andraka, "A survey of CORDIC algorithms for FPGA based computers," *Int. Symp. on FPGA*, 1998, pp. 191-200.

[3] X. Hu, R. G. Harber, and S. C. Bass, "Expanding the range of convergence of the CORDIC algorithm," *IEEE Trans. on Computers*, vol. 40, No. 1, 1991, pp. 13-21.

[4] J. A. Pineiro, S. F. Oberman, J. M. Muller, and J. D. Bruguera, "High-speed function approximation using minimax quadratic interpolator," *IEEE Trans. Computers*, vol. 54, No. 3, 2005, pp. 304-318.

[5] J. A. Pineiro, J. D. Bruguera, and J. M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," *IEEE Symp. on Computer Arithmetic*, 2001, pp. 40-47.

[6] V. K. Jain and L. Lin, "High-speed double precision computation of nonlinear functions," *IEEE Symp. on Computer Arithmetic*, 1995, pp. 107-114.

[7] M. J. Schulte and J. E. Stine, "Symmetric bipartite tables for accurate function approximation," *IEEE Symp. on Computer Arithmetic*, 1997, pp. 175-183.

[8] W. F. Wong and E. Goto, "Fast evaluation of the elementary functions in single precision," *IEEE Trans. Computers*, vol. 44, No. 3, 1995, pp. 453-457.

[9] T. Sasao, S. Nagayama, and J. T. Butler, "Numerical function generators using LUT cascades," *IEEE Trans. Computers*, vol. 56, No. 6, 2007, pp. 826-838.

[10] D. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "Non-uniform segmentation for hardware function evaluation," *Int. Conf. on Field Programmable Logic and Applications*, 2003, pp. 796-807.

[11] S. Nagayama, T. Sasao, and J. T. Butler, "Numerical function generators using edge-valued binary decision diagrams," *Asia and South Pacific Design Automation Conference*, 2007, pp. 535-540.

[12] S. Nagayama, T. Sasao, and J. T. Butler, "Programmable numerical function generators based on quadratic approximation: architecture and synthesis method," *Asia and South Pacific Design Automation Conference*, 2006, pp. 378-383.

[13] N. Brisebarre, D. Defour, P. Kornerup, J. M. Muller, and N. Revol, "A new range-reduction algorithm," *IEEE Trans. Computers*, vol. 54, No. 3, 2005, pp. 331-339.

[14] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: a study," *IEEE Trans. Neural Networks*, vol. 18, No. 1, 2007, pp. 240-252.

[15] Core Generator of Xilinx ISE 9.1i, Xilinx corp., Available: http://www.xilinx.com/

[16] R. Li, "Near optimality of Chebyshev interpolation for elementary function computations," *IEEE Trans. Computers*, vol. 53, No. 6, 2004, pp. 678-687.

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C++ the art of scientific computing*. 2nd ed., Cambridge University Press, 2002, ch. 3 and ch. 5.

[18] Wikipedia, *IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)*, Available: http://en.wikipedia.org/wiki/IEEE_754