IPSJ Transactions on System LSI Design Methodology Vol. 1 67–77 (Aug. 2008)

Regular Paper

A Synthesis Method of General Floating-Point Arithmetic Units by Aligned Partition[⋆1]

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

Since many embedded applications involve intensive mathematic operations, floating-point arithmetic units (FPU) have paramount importance in embedded systems. 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 and the concept of grouping floating-point numbers into zones, our method supports general functions of wide, irreducible domain. The synthesized FPU achieves smaller area, higher frequency, and greater accuracy. Experimental results show that our method achieves 1) on average 90% smaller and 50% faster indexer than the conventional automatic method; 2) on the hyperbolic functions, 20 k times smaller error rate and 50% use of LUTs and flip-flops than the conventional manual design.

1. Introduction

Many embedded applications require intensive calculation of floating-point arithmetic. The traditional software emulation, like FdLibM [1], is significantly slow and consumes lots of the CPU/DSP computing power. As the scaling technology continuously increases the integration density and reduces the transistor cost, there is a growing demand to evaluate arithmetic functions by floating-point unit (FPU). However, the use of FPU in the embedded system has not been very successful either because of the implementation effort involved or the unsatisfactory FPU quality (larger than the fixed-point unit, low throughput, the limited function types supported, etc.).

Function evaluation can be roughly classified into two groups: iterative and non-iterative. The iterative method CORDIC [2),3)], widely used in coprocessors, requires many iterations to get an accurate result, which usually has poor performance. Meanwhile, CORDIC is more suitable for fixed-point rather than floating-point designs. The non-iterative methods [4)–12)] have low delay and high throughput but only support special functions with reducible domain [13)], like reciprocal, square root, and logarithm. For more general functions (e.g. $tanh(x)$ and $sigmoid(x)$), existing methods are either slow [2),3)] or require lots of manual work [14)].

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 guarantees compact, fast, and 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 FPU optimization techniques, like the automatic exploration of hardware architecture in the design space, the reduction of register usage, etc.;

The rest of the paper is organized as follows: Section 2 summarizes previous works. Section 3 shows the overview of our synthesis framework. Section 4 explains the proposed method in detail. Section 5 presents some optimization techniques. Section 6 gives the experimental results. And Section 7 draws the conclusion.

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:

†1 Graduate School of Information, Production and System, Waseda University
†2 NEC Central Research Lab.
⋆1 This work was partly supported by a grant of Knowledge Cluster Initiative implemented by Ministry of Education, Culture, Sports, Science and Technology (MEXT).

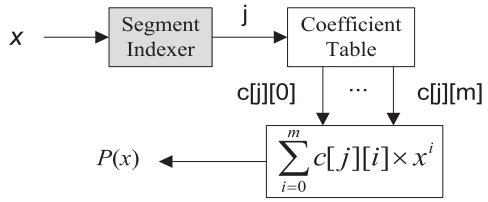

Fig. 1 Architecture of piecewise polynomial approximation.

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).

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 (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 Refs. 9) and 11), Sasao further improved the indexer implementation by using less wires but more memory for the FPGA platform.

In this paper, we extended the piecewise polynomial approximation [4)–12)] by supporting functions with wide, irreducible domain, like $tanh(x)$, $sigmoid(x)$, etc. We for the first time systematically analyzed the placement of segment boundaries and its impact on the implementation. The proposed aligned partition is capable of handling excessive (2,000) floating-point boundaries at high speed (50% faster)

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 $2 \times 1.\mathrm{M} \in [2,4)$

Table 2 Rating of different implementations of arithmetic unit.

Methods	fun types	impl. effort	number system	accuracy	pipeline	delay	types of ×, +	area
cordic_s [15]	2	3~8	fixed	3	No	5	NA	9
cordic_p [15]	2	3~8	fixed	3	Yes	5	NA	4
PPA	4	7	float	8	Yes	8	fixed	5
Sasao [12]	4	9	float	8	Yes	7	fixed	4
Proposed	9	9	float	8	Yes	6~8	float/ fixed	3~5

fun types: supported function types; impl. effort: implementation effort;
number system: number type of x, $f(x)$; cordic_s/p: serial/pipelined Cordic;
PPA: traditional piecewise polynomial approximation

and low hardware cost (90% smaller). Moreover, our FPU synthesis tool provides a friendly interface, through which users can easily specify the functions to be implemented.

Table 2 compares the popular arithmetic unit synthesis methods with ours on different aspects. For aspects that can be numerically rated, 9 means best and 1 means worst.

3. Overview of Proposed FPU Synthesis Method

Figure 2 shows the flow of the proposed synthesis tool. The 1st step is *function profiling*, in which user gives C descriptions of the function $f(x)$ to be synthesized. Meanwhile, the domain X of $f(x)$ as well as the acceptable absolute error $e(x)$ defined over X is provided. Then, in the 2nd step, X is partitioned into segments. Different from previous partition methods, ours first groups the floating-point numbers of X to *zones*; then, performs *zone*-based, *aligned* partition (details are explained in Section 4). The novel partition method greatly simplifies the indexing of excessive segments, making the synthesis of functions of wide domain possible. In the 3rd step, FPU is synthesized. Multipliers and adders of different

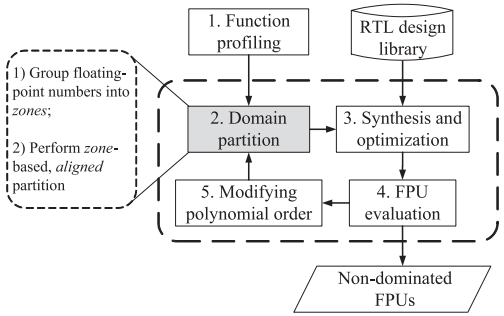

Fig. 2 Flow of proposed FPU synthesis method by aligned partition.

versions in the *RTL design library* are tried and various optimization techniques are applied to further refine the FPU. The 4th step evaluates the synthesized FPU (described in Verilog) by a logic synthesizer in terms of area, frequency, and delay. Non-dominated FPUs (FPU that is superior to any previous FPU on at least one aspect) are preserved in a cache, where users can choose the desired one. In the 5th step, the polynomial order is changed in order to explore different hardware architecture.

4. Zone-Based Aligned Partition

Partition of domain X into segments is critical in piecewise polynomial approximation. Two aspects need to be considered: 1) the approximation error; 2) hardware implementation. Traditional partitions [4)–8)] generate segments of fixed width (uniform partition), which are simple but produce unnecessary segments. The recent non-uniform partitions [9)–12)] support variable segment-width and reduce unnecessary segments in different ways. Fewer segments mean a smaller coefficient table. However, over-reducing segments will greatly complicate the hardware. In this study we for the first time systematically analyzed the domain partition and its impact on implementation. The resultant *zone*-based, *aligned* partition guarantees a high-speed, fully pipelined implementation that handles excessive floating-point segment boundaries at low hardware cost.

4.1 Segment Boundary Placement and Its Impact on Hardware

Definition 4.1 (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, we have $x_{j-1}' < x_j$, where x_j and x_{j-1}' are two adjacent numbers in X.*

```
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. 3 RTL implementation of the segment indexer.

Definition 4.2 (segment indexer). *The indexer is an integer function of the input $x \in X$ and the partition $PT = \{[x_0, x_0'], [x_1, x_1'], ..., [x_{k-1}, x_{k-1}']\}$. $indexer(x, PT) = \sum_{j=0}^{k-1} j \cdot T_j(x)$, 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 4.2 shows, the indexer strongly depends on partition PT. Hence, for any domain partition method, the impact of segment boundary placement on the indexer should be analyzed, which however has not been studied in any paper.

An indexer of k segments can be implemented by the Verilog code of **Fig. 3** in pure combinational circuit, where $t = \lceil \log_2 k \rceil$ is the bit-width of the segment index. To meet the requirement of pipelined processing, Fig. 3 adopts a fast, fully paralleled architecture that calculates index j for given x within one clock cycle. The implementation cost can be roughly estimated by Eq. (2).

$$\begin{aligned} \text{size(indexer)} < & \sum_{\text{j=0}}^{\text{k-1}} [\text{size}(T_j(x)) + \text{wired signal of } j] \\ & + (\text{k} \cdot \log_2 k)\,\text{OR gates} + (\text{k} \cdot \log_2 k)\text{AND gates} \end{aligned} \tag{2}$$

Usually, the segment number k is not extremely large (within 2 k as the experiment shows). 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 4.3 (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*

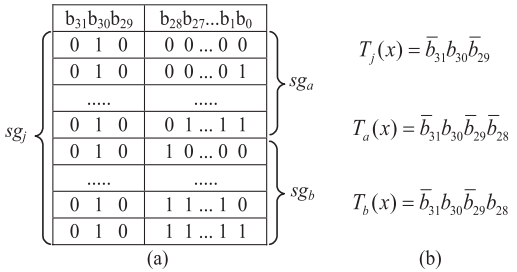

Fig. 4 Example of aligned segments. (a) sg_j is 29-bit aligned, sg_a and sg_b are 28-bit aligned; (b) corresponding $T_j(x)$, $T_a(x)$, and $T_b(x)$.

significant L bits change from 00...0 (L-bit wide) to 11...1 (L-bit wide) and the higher bits remain constant.

Theorem 4.1: *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.*

Proof: The Boolean function $T_j(x)$, as defined in Definition 4.2, is virtually a *truth table* [19]. Every $x \in sg_j$, with each bit regarded as a Boolean variable, is a *minterm* [19] for which $T_j(x) = 1$. If sg_j is L-bit aligned, the least significant L bits of all the minterms in the truth table will be eliminated after logic optimization. Thus, $T_j(x)$ becomes a single *product term* [19], which can be implemented by just a few AND gates. If sg_j is not L-bits aligned, the least significant L bits cannot be completely eliminated. Hence, $T_j(x)$ contains at least two longer product terms, which have more complex implementation.

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 4.1 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. 4**, 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 product term '$\bar{b}_{31}$ AND b_{30} AND $\bar{b}_{29}$', which requires only two AND gates without any comparator.

Theorem 4.2: *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.*

Figure 4 (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 $T_b(x)$.

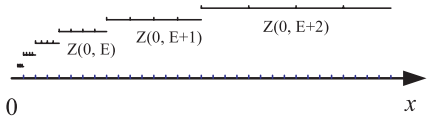

Fig. 5 Grouping floating-point numbers into zones.

4.2 Grouping Floating-Point Numbers to 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* floating-point 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 4.4 (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 ($b_{30}...b_{23} = E$).*

Figure 5 shows the grouping of all positive floating-point numbers into 254 zones. Each horizontal line represents a zone. The dots within a zone 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.

Apparently, $Z(S, E)$ is a 23-bit aligned segment. This means $T_j(x)$ 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.

4.3 Zone-Based, Aligned Non-Uniform Partition

As Fig. 5 shows, positive single-precision floating-point numbers can be grouped into 254 zones. Similarly, the entire X-axis, $(-2^{128}, 2^{128})$, can be divided 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 $T_j(x)$ 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 4.2.

Algorithm 1 shows the zone-based, aligned partition algorithm. The approximating polynomial $P(x)$ is obtained by the 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.

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;

5. FPU Optimizations

Extending the traditional piecewise polynomial approximation to functions of wide, irreducible domain is challenging. Efforts at all levels are required to deal with a potentially large number of segments. This section presents some of the efforts other than the aligned partition presented in Section 4.

5.1 Optimization of Evaluating Polynomial

The polynomial $P(x)$ of Fig. 1 plays an important role in the final FPU. Raising the order of $P(x)$ greatly reduces the segment number, which in turn decreases memory usage. However, higher order prolongs the evaluation delay and requires more multipliers, adders, and registers. Hence, a proper tradeoff should be made. In the proposed FPU synthesis method, the polynomial order is automatically modified, so that different hardware architectures can be explored in the design space, as Fig. 2 shows.

To minimize the delay of the polynomial evaluation and maximize the FPU throughput, we exploit the parallelism in $P(x)$ by optimizing the scheduling of multiplications and additions. Two techniques are proposed to optimize the pipelined scheduling:

(1) prioritizing multiplications involving x;
(2) registering segment index j instead of polynomial coefficients;

For explanation convenience, the following discussion is based on the 2nd-order approximation: $P(x) = c_0 + c_1 x + c_2 x^2$. However, the techniques can be used to polynomials of other order.

Figure 6 shows the pipelined scheduling of the 2nd-order polynomial. For a given x, the *indexer* decides the segment j that x falls in. Then, j is used as memory address to retrieve the polynomial coefficients. Terms of x^2 and $c_1 x$ are calculated first. Giving multiplications that involve x a higher priority saves registers. For term $c_2 x^2$, if the order is changed to $(c_2 x)x$, additional shift registers are needed to preserve the value of x for later multiplication. Thus, Fig. 6 saves (32-bit $\times$ *MUL_delay*) D registers, where *MUL_delay* is the delay of the multiplier (typically 6 clock cycles).

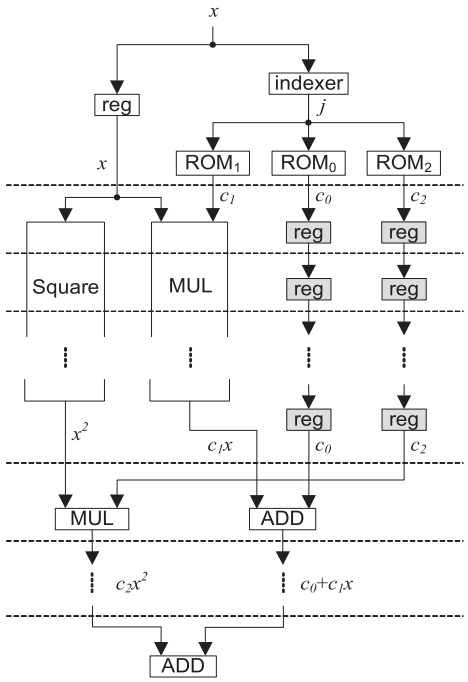

Fig. 6 Scheduling of 2nd-order $P(x)$ with x^2 and c_1x prioritized.

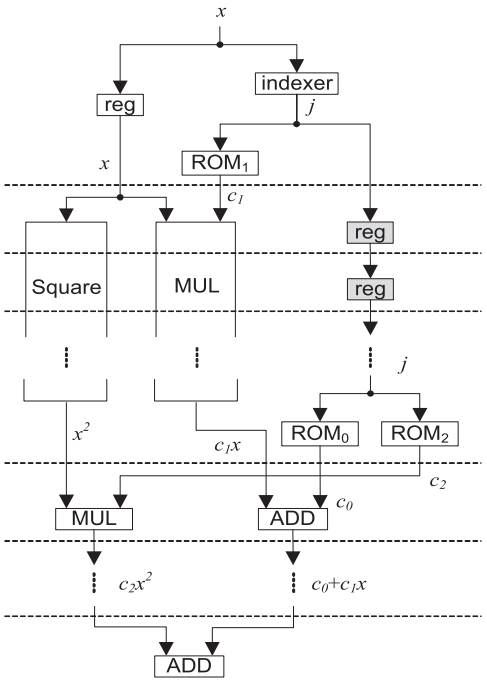

Fig. 7 Registering segment index j instead of c_0 and c_2.

The pipeline schedule of Fig. 6 eliminates registers to preserve x, however, at the cost of delaying operations involving c_0 and c_2. 64-bit wide shift registers are therefore needed to preserve c_0 and c_2. This problem can be solved by preserving index j instead of c_0 and c_2, as **Fig. 7** shows. The index j has a bit-width of $t = \lceil \log_2 k \rceil$, where k is the segment number. Since k is usually no more than 2000, t is typically within $6 \sim 11$. Consequently, another $(64 - t) \times MUL\_delay$ D registers can be saved.

5.2 Optimization of Segment Indexer

The term, {t{T_0(x)}} & t'd0 |, in the Verilog code of Fig. 3 can be eliminated, because $(y \ \& \ 0)$ is always 0 for any Boolean variable y. This means $T_0(x)$ of segment 0 need not be implemented by hardware. Consequently, the indexer can be optimized by finding the segment sg_j that has the most complex $T_j(x)$ among the k segments and assigning its index to 0. The optimization details are given in Algorithm 2, which has time complexity of O(k).

Algorithm 2: *Simplification of the indexer*

1. Evaluate the original indexer by compiling the automatically generated Verilog code;
2. Store the Verilog code in a cache;
3. for ($j = 1; j < k; j$++)
4. Change the segment indices of sg_j and sg_0 of the original partition to 0 and j respectively;
5. Generate the Verilog code of the indexer based on the new segment sequence;
6. Compile the new code. If it outperforms the one in the cache, update the cache with current code;

6. Experiment

For implementation convenience, we carried out the experiment on the FPGA platform. The automatically synthesized FPUs (in Verilog) are evaluated by Xilinx ISE 9.1i [15] on a Spartan 3 xc3s4000 device. The default optimization settings, "optimization goal = speed" and "optimization effort = normal", are used.

6.1 Aligned Partition vs. Least-Segment Partition

Experiment 1 compares the proposed aligned partition with Sasao's least-segment partition [12]. **Table 3** lists the functions to be examined. The *'Reduced domain X'* is the actual domain to be partitioned. Apparently, $tanh(x)$, $sech(x)$, and $sigmoid(x)$ that cannot have their original domain reduced to a narrow range cannot be implemented by the traditional piecewise polynomial approximation [4)–12)]. Both partition methods are based on the 2nd-order Chebyshev approximation [16),17)]. Absolute error e_a is set at 2^{-22} over X.

As expected, the least-segment partition [12] generates slightly fewer segments, because it does not consider the alignment constraint. The aligned partition requires 20% more memory bits on average. For ASIC design, these 20% extra memory bits (about 1.1 kb) are not a serious problem for the modern manufacturing technology. On the FPGA platform, both methods virtually use the same amount of memories, since the memory depth is $2^{\lceil \log_2 k \rceil}$ not the segment number k. Therefore, all the FPUs require three block RAMs to respectively store the 0th, 1st, and 2nd order coefficients of c_0, c_1, and c_2. Table 3 verifies that the aligned partition is almost as good as the least-segment partition on minimizing memory usage. The constraint imposed on the placement of segment boundaries is weak, which produces acceptable number of segments of nearly arbitrary width.

Table 4 compares the indexers under the two partition methods. On average, the aligned partition achieves 50% faster and 90% smaller (by 4.4 k gates) indexers. These indexers are generally faster than the final FPUs, as Table 6 shows. Therefore they are not on the critical path. The least-segment partition, on the other hand, generates some indexers as slow as around 50 MHz, which become the system bottleneck and seriously deteriorate the FPU performance. This result confirms the importance of aligning segment boundaries during the domain partitioning, which makes $T_j(x)$ independent of the least significant bits of x. The result is a simpler and faster indexer.

For the evaluating polynomial, as Fig. 1 shows, both methods use identical $P(x)$. Thus, the total gate count increase of the proposed method can be roughly estimated as (1.1 k bits × 6 gates/bit - 4.4 k gates) = 2.2 k gates, which is well justified by the 50% speedup of the indexer. These 2.2 k extra gates are actually insignificant, compared with the evaluating polynomial $P(x)$ that dominates the total FPU size. Any floating-point adder (about 10 k gates) or multiplier (about 20 k gates) in $P(x)$ is much larger. Based on these data, we conclude that the aligned partition outperforms the least-segment partition.

Table 3 Memory usage under two partition methods ($e_a \leq 2^{-22}$).

f(x)	Original domain	Reduced domain X	Aligned			Least-segment [12]		
			Seg #	Mem bits	Mem depth	Seg #	Mem bits	Mem depth
$1/x$	$(-\infty, -2^{-126}] \cup [2^{-126}, \infty)$	[1, 2)	99	6336	128	82	5248	128
$\sqrt{x}$	$[2^{-126}, \infty)$	[1, 2)	50	3200	64	43	2752	64
log2(x)	$[2^{-126}, \infty)$	[1, 2)	49	3136	64	38	2432	64
sin(x)	$[-\pi/2, \pi/2]$	$[0, \pi/2]$	47	3008	64	40	2560	64
tanh(x)	$(-\infty, \infty)$	$[0, \infty)$	113	7232	128	92	5888	128
sech(x)	$(-\infty, \infty)$	$[0, \infty)$	142	9088	256	132	8448	256
sigmoid(x)	$(-\infty, \infty)$	$(-\infty, \infty)$	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 10s.
2. ∞ denotes the largest number (around 3.4×10^{38}) the single-precision floating-point number can represent.

Table 4 Indexers under two partition methods ($e_a \leq 2^{-22}$).

f(x)	Aligned		Least-segment [12]	
	Gate #	Freq. MHz	Gate #	Freq. MHz
$1/x$	564	131	1242	76
$\sqrt{x}$	132	185	123	207
$log_2(x)$	135	217	132	196
sin(x)	168	166	3423	76
tanh(x)	462	136	8847	53
sech(x)	870	118	12186	51
sigmoid(x)	912	116	8313	72
Average	463	152	4895	104

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

6.2 Comparison with Manual Design on sigmoid(x)

Hyperbolic functions, like $tanh(x)$ and $sigmoid(x)$, have important application in neural networks. However, due to the wide, irreducible domain, they cannot be implemented by the traditional piecewise polynomial approximation [4)–12)]. In Ref. 14), Savich et al. manually implemented $sigmoid(x)$ in various number formats on FPGA. Hence, Ref. 14) serves as a reliable example of the latest manual implementation. Experiment 2 compares our automatic synthesis with the manual design [14)].

$$sigmoid(x) \approx \begin{cases} 0, & \text{if } x \le -8 \\ (8 - |x|)/64, & \text{if } -8 < x \le -1.6 \\ x/4 + 0.5, & \text{if } |x| < 1.6 \\ 1 - (8 - |x|)/64, & \text{if } 1.6 \le x < 8 \\ 1, & \text{if } x \ge 8 \end{cases} \quad (3)$$

Reference 14) uses five-segment linear approximation, as Eq. (3) shows. The 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 a method is representative in RTL designing: first find the approximating function experimentally; then apply some '*optimizations*'. Implementation like this has three problems: 1) *high hardware cost*. The segment boundaries (± 1.6) are not aligned. Expensive comparators are needed to implement the indexer; 2) *low accuracy*. The restriction on the 1st-order coefficients increases the error, making the approximation useless to high precision applications; 3) *poor systematic tradeoff*. Eliminating the multiplier seems to simplify the design. Actually, any segment that has a different 1st-order coefficient will need specific bit shift or exponent addition, making the hardware cost proportional to the segment number.

In **Table 5**, *FXD* and *FLT* denotes the fixed and floating point implementation of $sigmoid(x)$ in Ref. 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*' shows the correct way of eliminating the multiplier. It only consists of an indexer and the 0th-order coefficient table. '*a0_8*' has an absolute error no more than 0.0039. That is about 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*', which is a simple look-up table (0th-order approx.), to outperform the linear (1st-order) approximation of Ref. 14).

Table 5 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

'*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 20 k times smaller than that of *FLT*.

Experiment 2 proved the ability of the aligned partition algorithm to handle excessive segments at low cost and high speed. The support of excessive segments greatly expands the domain X. Consequently, the proposed synthesis method is able to support more general functions with wide, irreducible domain.

It should be emphasized that our FPU synthesis tool is highly automatic. FPUs are generated and verified in minutes. The implementations of Ref. 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 in modern integration technology. Since Spartan3 xc3s4000 provides abundant (96) block RAMs, we store the coefficients in block RAMs.

Figure 8 compares the approximation error of the five-segment method [14)] with '*a1_18*'. Obviously, '*a1_18*' is far more accurate and uses only half the flip-flops and LUTs.

6.3 Comparison with Commercial Tools

As Fig. 1 shows, the synthesized FPU has fixed architecture: indexer, coefficient

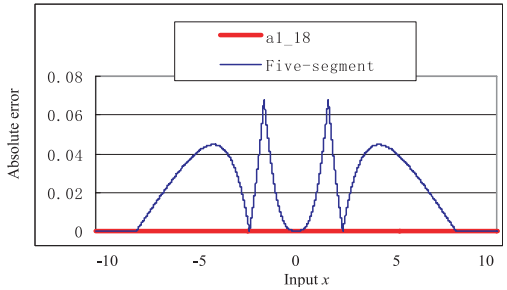

Fig. 8 Absolute error of different $sigmoid(x)$ implementations.

table, and the evaluating polynomial $P(x)$. Since the aligned partition greatly simplifies the indexer, the FPU size is mainly decided by $P(x)$. To cope with a wide domain X, floating-point adders and multipliers are used in this study to evaluate $P(x)$. It should be emphasized that though the floating-point multipliers and adders are larger than their fixed-point counterparts, they are not a problem in modern integration technology. Moreover, these units are efficiently utilized, as they are shared by all the segments. The number of multipliers and adders can be greatly reduced by lowering $P(x)$ order. However, lower order will increase segments (memory usage). Thus, it is the user who selects the most suitable FPU from the automatically synthesized ones.

Table 6 shows the FPUs synthesized by the proposed method under the 1st and 2nd order approximation. Some FPUs have nearly 2,000 segments (all segment boundaries are floating-point, not fixed-point as in previous methods [4)–12)]) but still achieve high speed. Also, FPUs under the same order use similar resources. These observations mean that the aligned partition effectively handles excessive segments. The optimized indexer is fast and small, so that the multipliers and adders of $P(x)$ dominate the FPU size. It should be emphasized that all the synthesized units are pipelined (process a new x every clock cycle) and none use more than 8% of the FPGA resources. Thus, a single FPGA chip can include multiple independent FPUs that perform pipelined calculation in parallel, which can accelerate the computation-intensive embedded applications greatly.

Table 7 shows the arithmetic units synthesized by Xilinx Core Generator [15)] using CORDIC algorithm. As is shown, non-pipelined units are compact but very slow. Pipelined units, though being fixed-point, are much larger and use even more LUTs and flip-flops than our floating-point implementations in Table 6.

Table 6 FPUs synthesized by proposed method ($e_a \leq 2^{-22}$).

f(x)	1st-order, delay = 19 cycles					2nd-order, delay = 32 cycles				
	Seg #	Flip flop	LUT	Block RAM	Clk (MHz)	Seg #	Flip flop	LUT	Block RAM	Clk (MHz)
1/x	962	885 (1%)	1415 (2%)	4/96	122	99	2201 (3%)	1681 (3%)	3/96	97
$\sqrt{x}$	624	885 (1%)	1101 (1%)	4/96	119	50	2190 (3%)	1607 (2%)	3/96	124
$log_2(x)$	626	885 (1%)	1000 (1%)	4/96	117	49	2189 (3%)	1606 (2%)	3/96	101
sin(x)	880	894 (1%)	1298 (2%)	4/96	124	47	2213 (4%)	1663 (3%)	3/96	144
tanh(x)	1197	904 (1%)	1268 (2%)	6/96	115	113	2219 (4%)	1715 (3%)	3/96	141
sech(x)	1599	894 (1%)	1967 (3%)	8/96	126	142	2219 (4%)	1772 (3%)	3/96	111
sigmoid(x)	1896	894 (1%)	1690 (3%)	8/96	127	181	2219 (4%)	1781 (3%)	3/96	107

1. Each unit can be automatically synthesized in less than one minute.
2. Pipelined floating-point adders and multipliers are used to implement $P(x)$, which have respectively 13 and 6 pipeline stages.
3. For functions of reducible domain (e.g. $1/x$, $sin(x)$, etc.), smaller and faster fixed-point multipliers and adders can be used. In such cases, our method is reduced to traditional piecewise polynomial approximation.

Table 7 Arithmetic units synthesized by Xilinx Core Generator.

f(x)	Pipeline	Delay (cycles)	Flip flops	LUTs	Block RAM	Clk (MHz)
sinh(x)/cosh(x)_p	Yes	27	2,255	2,294	0/96	121
sinh(x)/cosh(x)_s	No	29	165	576	0/96	78
sin(x)/cos(x)_p	Yes	25	2,155	2,234	0/96	123
sin(x)/cos(x)_s	No	27	352	857	0/96	68

* The input/output bit-width is set at 24, which has the same precision as the single-precision floating-point mantissa.

7. Conclusion

Floating-point arithmetic units (FPUs) are important to accelerate computation-intensive embedded applications. In this paper, we present an automatic method to synthesize general FPUs by aligned partition. Based on the novel domain partition, our method is able to synthesize functions of wide, irre-

ducible domain that previous automatic methods cannot. The synthesized FPU features compact size, high frequency, and low error. Experimental results show that our method achieves an indexer on average 90% smaller and 50% faster than the conventional automatic method. On synthesizing functions of wide domain, like $sigmoid(x)$, our method achieves 20 K times smaller error rate and 50% use of LUTs and flop-flops than the manual design. Future works will focus on the rounding of polynomial coefficients and the reduction of memory usage.

References

1) FdLibM: C math library for machines that support IEEE 754 floating-point, Available: http://www.netlib.org/fdlibm/
2) Andraka, R.: A survey of CORDIC algorithms for FPGA based computers, *Int. Symp. on FPGA*, pp.191–200 (1998).
3) Hu, X., Harber, R.G. and Bass, S.C.: Expanding the range of convergence of the CORDIC algorithm, *IEEE Trans. on Computers*, Vol.40, No.1, pp.13–21 (1991).
4) Pineiro, J.A., Oberman, S.F., Muller, J.M. and Bruguera, J.D.: High-speed function approximation using minimax quadratic interpolator, *IEEE Trans. Computers*, Vol.54, No.3, pp.304–318 (2005).
5) Pineiro, J.A., Bruguera, J.D. and Muller, J.M.: Faithful powering computation using table look-up and a fused accumulation tree, *IEEE Symp. on Computer Arithmetic*, pp.40–47 (2001).
6) Jain, V.K. and Lin, L.: High-speed double precision computation of nonlinear functions, *IEEE Symp. on Computer Arithmetic*, pp.107–114 (1995).
7) Schulte, M.J. and Stine, J.E.: Symmetric bipartite tables for accurate function approximation, *IEEE Symp. on Computer Arithmetic*, pp.175–183 (1997).
8) Wong, W.F. and Goto, E.: Fast evaluation of the elementary functions in single precision, *IEEE Trans. Computers*, Vol.44, No.3, pp.453–457 (1995).
9) Sasao, T., Nagayama, S. and Butler, J.T.: Numerical function generators using LUT cascades, *IEEE Trans. Computers*, Vol.56, No.6, pp.826–838 (2007).
10) Lee, D., Luk, W., Villasenor, J. and Cheung, P.Y.K.: Non-uniform segmentation for hardware function evaluation, *Int. Conf. on Field Programmable Logic and Applications*, pp.796–807 (2003).
11) Nagayama, S., Sasao, T. and Butler, J.T.: Numerical function generators using edge-valued binary decision diagrams, *Asia and South Pacific Design Automation Conference*, pp.535–540 (2007).
12) Nagayama, S., Sasao, T. and Butler, J.T.: Programmable numerical function generators based on quadratic approximation: architecture and synthesis method, *Asia and South Pacific Design Automation Conference*, pp.378–383 (2006).
13) Brisebarre, N., Defour, D., Kornerup, P., Muller, J.M. and Revol, N.: A new range-reduction algorithm, *IEEE Trans. Computers*, Vol.54, No.3, pp.331–339 (2005).
14) Savich, A.W., Moussa, M. and Areibi, S.: The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study, *IEEE Trans. Neural Networks*, Vol.18, No.1, pp.240–252 (2007).
15) Core Generator of Xilinx ISE 9.1i, Xilinx corp., Available: http://www.xilinx.com/
16) Li, R.: Near optimality of Chebyshev interpolation for elementary function computations, *IEEE Trans. Computers*, Vol.53, No.6, pp.678–687 (2004).
17) Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P.: *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
19) Holdsworth, B. and Woods, C.: *Digital logic design*. 4th ed., Newnes, 2002, ch. 2.

(Received December 25, 2007)
(Revised March 24, 2008)
(Accepted May 1, 2008)
(Released August 27, 2008)

(Recommended by Associate Editor: *Tsuyoshi Isshiki*)

Liangwei Ge received his B.E. from Xi'an Jiaotong University, China, in 2000 and M.E. from Tsinghua University, China, in 2003. He is currently a Ph.D. candidate in the Graduate School of Information, Production and System, Waseda University, Japan. His research interests include high-level synthesis, computer arithmetic, and VLSI design automation.

Song Chen received the B.S. degree in computer science from Xi'an Jiaotong University, China, in 2000 and Ph.D. in computer science from Tsinghua University, Peking, China, in 2005. From August 2005, he has been a post-doctor in the Graduate School of Information, Production and System, Waseda University, Japan. His research interests include high-level/physical synthesis of VLSI circuits, especially floorplanning/placement for 2D/3D ICs, integration of high-level synthesis and physical synthesis, etc.

Yuichi Nakamura received B.E. in information engineering and a M.E. in electrical engineering from Tokyo Institute of Technology in 1986 and 1988. He received Dr. Eng. degree from Graduate School of Information, Production, and Systems, Waseda University. In 1988, he joined NEC Corporation, where he is currently a principal researcher of the System IP Core Research Laboratories. His research interests include the design and verification of high-speed and complex VLSIs.

Takeshi Yoshimura received B.E., M.E., and Dr. Eng. degrees from Osaka University, Osaka, Japan, in 1972, 1974, and 1997. He joined the NEC Corporation, Kawasaki, Japan, in 1974, where he has been engaged in research and development efforts devoted to computer application systems for communication network design, hydraulic network design, and VLSI CAD. From 1979 to 1980 he was on leave at the Electronics Research Laboratory, University of California, Berkeley, where he worked on very large scale integration computer-aided design layout. He received Best Paper Awards from the Institute of Electronics, Information and Communication Engineers of Japan (IEICE) and the IEEE CAS Society. Dr. Yoshimura is a Member of the IEICE, IPSJ (the Information Processing Society of Japan), and IEEE.