令　和　五　年　度　　修　士　論　文

分散型積和演算を用いたテイラー展開省面積演算回路の設計

指導教員　　田中　勇樹　准教授

群馬大学大学院理工学府 理工学専攻
電子情報・数理教育プログラム

Hemthavy　Xaybandith

目次

第 1 章　はじめに　1
- 1.1　研究背景 1
- 1.2　研究内容 1

第 2 章　分散型積和演算法 (DA 法)　3
- 2.1　DA 法の手順 3
- 2.2　回路に適用する際のアルゴリズム 4
- 2.3　回路内の動作例 5
- 2.4　メリット 6

第 3 章　項分割分散型積和演算法（項分割 DA 法）　7
- 3.1　項分割 DA 法の場合の変更点と実行例 7
- 3.2　メリット 8

第 4 章　回路設計　9
- 4.1　通常演算法 9
- 4.2　DA 法 10
- 4.3　項分割 DA 法 11

第 5 章　評価　13
- 5.1　シミュレーション結果と誤差 13
- 5.2　回路面積と遅延時間 16

第 6 章　まとめと今後の課題　18

参考文献　19

謝辞 21

付録 A ソースコードリスト 22

A.1 通常演算法 26

A.2 DA 法 33

図目次

4.1 通常演算法のブロック図 9
4.2 通常演算法のタイミングチャート 10
4.3 DA 法のブロック図 11
4.4 DA 法のタイミングチャート 12
4.5 項分割 DA 法のブロック図 12

5.1 通常演算法回路のシミュレーション結果（前半） 13
5.2 通常演算法回路のシミュレーション結果（後半） 14
5.3 DA 法回路のシミュレーション結果（前半） 14
5.4 DA 法回路のシミュレーション結果（後半） 15
5.5 回路演算結果と各式との誤差 15
5.6 項数と誤差の相関図 16

表目次

2.1 $a_0 = 1, a_1 = 3, a_2 = 6$ のすべての X_i のパターンにおける和 4

5.1 通常演算法回路と DA 法回路の回路評価 16

第 1 章

はじめに

1.1　研究背景

集積回路技術の発展により，電子計算機や携帯電子機器など多岐にわたり高速・高精度・省面積であることなどが求められる．それとともに積和演算処理が多く使われるが，乗算器を用いる個数が多いほど回路規模が大きくなり，ハードウェアコストの増加の一因となる．したがって積和演算回路を省面積で設計することが全体のハードウェアコストの削減に貢献するものと考える．ここで分散型積和演算法（Distributed Arithmetic 法，以降 DA 法）を用いることで乗算器の個数を削減し，省面積回路を設計することを目的とした [1]-[4]．DA 法とは乗算器なしで積和演算が行えるために LMS(Least Mean Square) 適応フィルタ等を設計する際などに省面積，低電力を実現するために使われる [5]-[14]．

1.2　研究内容

本研究は積和演算回路において分散型積和演算法を用いることで乗算器の使用個数を削減し，省面積化を図ることを目的とする．自然現象（アナログ）の表現や様々な関数の計算をデジタル信号処理で行う際によく使われるテイラー展開式 [10]-[14] をモデルに DA 法を適用したときを考える．ここでは特に e^x の 0 周りでのテイラー展開を x^5 の項まで行い，

$$f(x) = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \frac{1}{5!}x^5 \tag{1.1}$$

(1.1) 式について通常演算法と DA 法のそれぞれで演算回路をデジタル回路設計用のハードウェア記述言語である VHDL を用いて回路設計し，評価する．また，これまでの研究で DA 法についても，提案する項分割 DA 法で演算を行うことで省面積になることが期

待されるが今回の設計時には考案する構造において欠陥が見つかったためアルゴリズムの説明のみに止める.

本研究に係る対外発表

1. X. Hemthavy, J. Wei, S. Katayama, A. Kuwana, H. Kobayashi, K. Kubo, ”Efficient Hardware Architecture for Taylor-Series Expansion Calculation Using Distributed Arithmetic with Term Division,” The 24th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2022), Aomori, Japan, (Oct. 2022)
2. X. Hemthavy, J. Wei, S. Katayama, A. Kuwana, H. Kobayashi, K. Kubo, ”DA for Hardware Architecture of Taylor Series Expansion Calculation,” Taiwan and Japan Conference on Circuits and Systems (TJCAS2022), online, (Nov. 2022)
3. ヘムタビーサイバンディット，田中勇樹，小林春夫,「分散型積和演算を用いたテイラー展開省面積演算回路の設計」，第 29 回 電子情報通信学会東京支部学生会研究発表会（2024 年 3 月発表予定）

第 2 章

分散型積和演算法 (DA 法)

この章では，DA 法を使った計算手順を解説する．

2.1 DA 法の手順

整数部 2 進数 k 桁，小数部 2 進数 l 桁の正の実数 X_i を以下の様に表す．

$$X_i = x_{i,k-1}, x_{i,k-1}, \ldots, x_{i,0}, x_{i,-1}, \ldots, x_{i,-l} \tag{2.1}$$

ここで，$x_{i,j} \in \{0,1\}(-l \leq j \leq k-1)$ であり，a_i を定数としたとき，n 個の項の重み付き和 Z は

$$Z = \sum_{i=0}^{n-1} a_i X_i \tag{2.2}$$

のように表せる．このとき，各 X_i に着目すると，

$$\begin{aligned} Z &= \sum_{j=-l}^{k-1} \sum_{i=0}^{n-1} 2^j a_i x_{i,j} \\ &= 2^{k-1} \sum_{i=0}^{n-1} a_i x_{i,k-1} + 2^{k-2} \sum_{i=0}^{n-1} a_i x_{i,k-2} + \cdots + 2^{-l} \sum_{i=0}^{n-1} a_i x_{i,-l} \end{aligned} \tag{2.3}$$

と書ける．ここで，各 a_i は定数であり，$x_{i,j}$ は 0 または 1 であるので，a_0 から a_{n-1} までの 2^n 種類ある全ての組み合わせに対して定数の和を計算して記録しておくことができる．これによって式 (2.3) の回路上での計算において 2^j はビットシフト，$\sum_{i=0}^{n-1} a_i x_{i,j}$ は記録から読みだした値によって与えられる．したがって，DA 法では事前準備を除けばビットシフトと加算のみで積和演算を実現できる．

例として，式 (2.2) において以下の条件で計算する場合を考える．

$$
\begin{aligned}
a_0 = 1, a_1 = 3, a_2 &= 6 \\
X_0 &= (100.01101)_2 \\
X_1 &= (010.11100)_2 \\
X_2 &= (111.00100)_2
\end{aligned}
$$

このとき a_i の和の記録は以下の表のようになる．

表 2.1　$a_0 = 1, a_1 = 3, a_2 = 6$ のすべての X_i のパターンにおける和

X_0	X_1	X_2	和
0	0	0	0
0	0	1	6
0	1	0	3
0	1	1	9
1	0	0	1
1	0	1	7
1	1	0	4
1	1	1	10

したがって Z を算出する式は以下のようになる．

$$
\begin{aligned}
Z &= a_0X_0 + a_1X_1 + a_2X_2 \\
&= 2^2 \times 7 + 2^1 \times 9 + 2^0 \times 6 + 2^{-1} \times 3 + 2^{-2} \times 10 + 2^{-3} \times 10 + 2^{-4} \times 0 + 2^{-5} \times 1
\end{aligned}
$$

2.2　回路に適用する際のアルゴリズム

まずは上記の様に全ての X_i を求める．式 (1.1) を考えた場合，x のべき乗を計算して式 (2.4) のように列を揃えてレジスタに格納する．

$$
\begin{pmatrix} x^0 \\ x^1 \\ x^2 \\ x^3 \\ x^4 \\ x^5 \end{pmatrix} = \begin{pmatrix} \square & \square & \square. & \square & \square & \square & \square \\ \square & \square & \square. & \square & \square & \square & \square \\ \square & \square & \square. & \square & \square & \square & \square \\ \square & \square & \square. & \square & \square & \square & \square \\ \square & \square & \square. & \square & \square & \square & \square \\ \square & \square & \square. & \square & \square & \square & \square \end{pmatrix} \tag{2.4}
$$

次に表 2.1 のように全ての x のべき乗のパターンに対応する和を求めて LUT を作成する．レジスタに格納された x のべき乗を列ごとに得たものをアドレスとして，和の LUT から和の値を読みだして加算する．列ごとの和に 2 進数の桁ごとの重みを付けながらすべての列の和を求めることで (1.1) 式の答えが得られる．最上位桁から最下位桁に向けて加算する場合は列ごとに 2 の重みを，最下位桁から最上位桁に向けて加算する場合には列ごとに $\frac{1}{2}$ の重みをかけながら加算する．最後に小数点の位置を補正するために $\frac{1}{2}$ を小数部分の桁数だけかける，または 2 を整数部分の桁数 -1 回分かける．

以下の式は最上位桁から最下位桁に向けて加算した場合である．なお，このときの n は小数部分の桁数である．

$$f(x) = \left(2 \times \ldots \left(2 \times \left(\begin{array}{c} 2 \times (2 \times (\text{最上位桁})) \\ +(\text{最上位桁} - 1) \end{array} \right) \begin{array}{c} \\ \\ +(\text{最上位桁} - 2) \\ \ldots + (\text{最下位桁}) \end{array} \right) \right) \times \frac{1}{2^n} \tag{2.5}$$

2.3 回路内の動作例

(1.1) 式において 10 進で $x = 1.5$ とおくと，以下の (2.6) 式のようになる．

$$f(1.5) = 1 + 1.5 + \frac{1}{2} \times 1.5^2 + \frac{1}{6} \times 1.5^3 + \frac{1}{24} \times 1.5^4 + \frac{1}{120} \times 1.5^5 \tag{2.6}$$

ここで 2 進数でべき乗を計算し，(2.4) 式のような行列を作ると次の (2.7) のようになる．

$$\begin{pmatrix} 1.1^0 \\ 1.1^1 \\ 1.1^2 \\ 1.1^3 \\ 1.1^4 \\ 1.1^5 \end{pmatrix}_{(2)} = \begin{pmatrix} 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1. & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0. & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1. & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1. & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1. & 1 & 0 & 0 & 1 & 1 \end{pmatrix}_{(2)} \tag{2.7}$$

分散型積和演算を適用すると (2.8) 式が得られる．

$$f(1.5) =$$

$$
\left(2\times\left(2\times\left(2\times\left(2\times\left(2\times\left(2\times\left(2\times\left(\frac{1}{24}+\frac{1}{120}\right)+\left(\frac{1}{2}+\frac{1}{6}+\frac{1}{120}\right)\right)+\left(1+1+\frac{1}{6}+\frac{1}{24}+\frac{1}{120}\right)\right)+\left(1+\frac{1}{120}\right)\right)+\left(\frac{1}{2}+\frac{1}{6}\right)\right)+\left(\frac{1}{6}\right)\right)+\left(\frac{1}{24}+\frac{1}{120}\right)\right)+\left(\frac{1}{120}\right)\right)\times\frac{1}{2^5} \tag{2.8}
$$

この式に基づいて計算すると $f(1.5) = 4.461719$ が得られ，元の $e^{1.5}$ の計算結果，4.481689 と近い値が得られることが確認できた．

2.4　メリット

上記で示した計算法で回路を設計する際には 2 や $\frac{1}{2}$ の重みをかける際にはビットシフトで実現できるため，実際に乗算が必要な個所ははじめの x のべき乗を計算する部分のみである．これにより乗算が必要な計算箇所が減るため，回路設計時に省面積になることが期待される．

また，例として序論で引用した先行研究ではこの DA 法をテイラー展開式に適用することで，e^x，$\log x$，$\sqrt{x}$ や $\frac{1}{\sqrt{x}}$ などを少ない回数の乗算処理で計算している [1]-[4].

第 3 章

項分割分散型積和演算法（項分割 DA 法）

この章では DA 法で計算するにあたって作成する LUT のサイズを，さらに小さくする方法として考案した項分割 DA 法の計算手順について解説する．

3.1　項分割 DA 法の場合の変更点と実行例

DA 法は (1.1) 式について x のべき乗を全て計算するものだったの対し，項分割 DA 法とは以下の (3.1) 式のように項を大きく 2 分割した式に対して行うものである．

$$f(x) = \left(1 + \frac{1}{2}x^2 + \frac{1}{24}x^4\right) + x\left(1 + \frac{1}{6}x^2 + \frac{1}{120}x^4\right) \tag{3.1}$$

手順については DA 法とおおむね同様である．DA 法と同様に $x = 1.5$ で考えたとき，はじめに必要な x のべき乗を 2 進数で計算すると以下の (3.2) 式のようになる．

$$\begin{pmatrix} 1.1^0 \\ 1.1^2 \\ 1.1^4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1. & 0 & 0 & 0 & 0 \\ 0 & 1 & 0. & 0 & 1 & 0 & 0 \\ 1 & 0 & 1. & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

次に左右の括弧について DA 法と同様に計算をする．左の括弧について示したものが

(3.3)，右の括弧について示したものが (3.4) である．

$$\left(1+\frac{1}{2}x^2+\frac{1}{24}x^4\right)=\left(2\times\left(2\times\left(2\times\left(2\times\left(2\times\left(\tfrac{1}{24}\right)+\left(\tfrac{1}{2}\right)\right)+\left(1+\tfrac{1}{24}\right)\right)+(0)\right)+\left(\tfrac{1}{2}\right)\right)+(0)\right)+\left(\tfrac{1}{24}\right)\right)\times\frac{1}{2^5}$$
$$=2.335938 \tag{3.3}$$

$$\left(1+\frac{1}{6}x^2+\frac{1}{120}x^4\right)=\left(2\times\left(2\times\left(2\times\left(2\times\left(2\times\left(\tfrac{1}{120}\right)+\left(\tfrac{1}{6}\right)\right)+\left(1+\tfrac{1}{120}\right)\right)+(0)\right)+\left(\tfrac{1}{6}\right)\right)+(0)\right)+\left(\tfrac{1}{120}\right)\right)\times\frac{1}{2^5}$$
$$=1.417188 \tag{3.4}$$

最後に式通り右括弧を 1.5 倍して和を求めると (3.5) のように答えが求まる．

$$\begin{aligned} f(x) &= (2.335938)+1.5\times(1.417188) \\ &= 4.461719 \end{aligned} \tag{3.5}$$

3.2　メリット

DA 法では x のべき乗を 6 つ計算するのに対し，項分割 DA 法では共通した x のべき乗を作るため 3 つ計算するだけでよい．計算する項の多さに合わせて 2 分割だけでなく 3 分割，4 分割することで計算する x のべき乗を削減することができる．

これは回路設計時に行列の列をアドレスとして LUT で和を出力する構成にする場合，DA 法では 6 ビットの全てのアドレスに対して LUT を作成するため $2^6=64$ パターンのアドレスを用意する必要がある．項分割 DA 法では上記の様に 2 分割した場合，3 ビットの全てのアドレスに対して LUT を 2 個作成すればよいため $2^3=8$ パターンの LUT を 2 個作成することになるので LUT の面積縮小が期待できる．

第 4 章

回路設計

この章では通常演算法回路，DA 法回路，分割 DA 法回路の設計概要について解説する．

4.1　通常演算法

通常演算法では (1.1) 式のとおり左から順に計算していく．この時の回路のブロック図を以下の図 4.1 に示す．回路内では主に浮動小数点演算を行う．入力は (1.1) 式の x であ

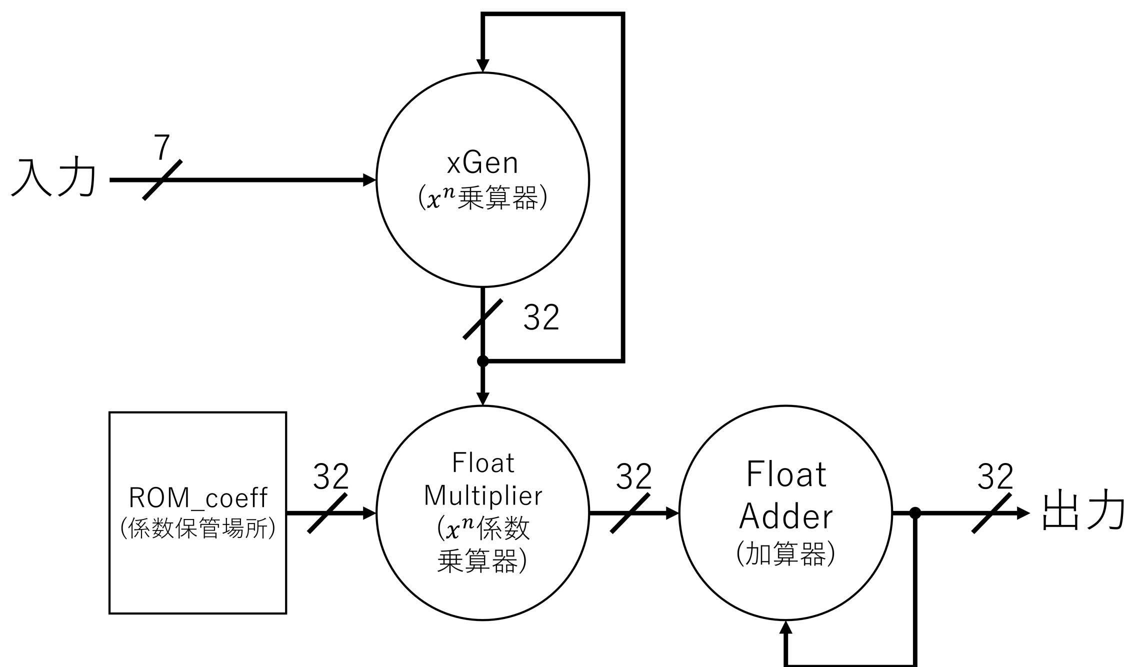

図 4.1　通常演算法のブロック図

り，10 進数において 1 以上 2 未満となる数を扱う．したがって浮動小数点表現における仮数部のように入力を表せるので 2 進数の 7 ビットを入力とし，回路内で以下の (4.1) 式のような x で (1.1) 式を通常演算法で計算する．

$$1 \leq x = 1.\square\square\square\square\square\square\square < 2 \tag{4.1}$$

これを xGen で繰り返し乗算を行い，x^5 まで計算する．ROM_coeff で保管されていた各項の係数と x^0 から x^5 までを Float Multiplier でクロックごとに積を計算し，Float Adder で順次加算することで計算結果を得る．このときのタイミングチャートを以下の図 4.2 に示す．

4.2 DA 法

DA 法のブロック図を以下の図 4.3 に示す. 通常演算法と同様の入力をもとに Multiplier で x のべき乗を算出し，上位 22 ビットのみを Address Reg に格納する．そうして x^0 から x^5 までを (2.4) 式のように格納し終わったのちに，最下位ビットから最上位ビットに向けて列ごとに切り取り，6 ビットのアドレスとして LUT_when に渡す．アドレスに対応した係数の和を Float Adder で順次加算しながら $\frac{1}{2}$ の重みをかけていく．この $\frac{1}{2}$ の重みをかける処理は指数部を 1 減らすことで実現できる．最上位ビットまでを加算し終わっ

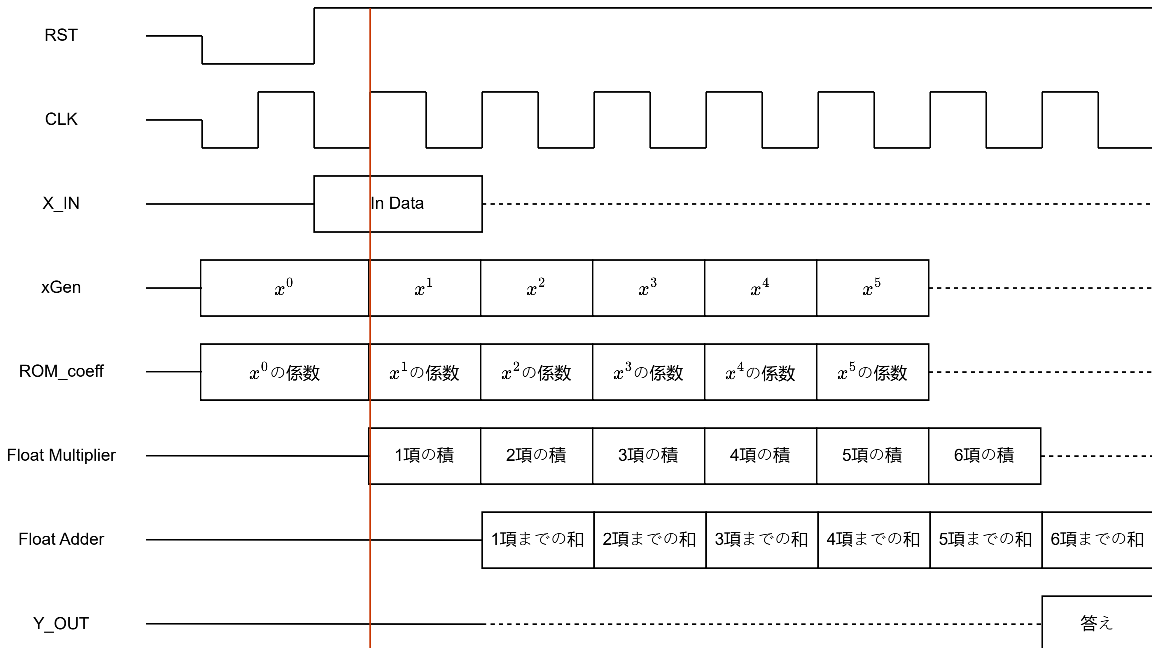

図 4.2　通常演算法のタイミングチャート

たタイミングで Point Adjustment にデータを渡し，小数点位置補正をするために 2 の重みをかけることで計算結果を得る．2 章では整数桁数 −1 回分重みをかけると記述したが，回路の設計都合上今回は整数桁数分だけ重みをかけている．この時のタイミングチャートを以下の図 4.4 に示す．

22 ビットのみをアドレスとして格納する理由は浮動小数点表現の加算におけるビットシフトによる下位ビットの消滅である．ここでは単精度浮動小数点を扱っているため，加算を行う際には指数部を揃えたうえで仮数部の加算を行う．このとき指数部の大きいほうに指数部を揃えるため，指数部の小さいほうの仮数部はビット右シフトにより下位ビットが消滅する．仮数部は 23 ビット用意されているが，省略されている整数部 1 ビットを加えたうえで加算処理を行うため，最下位ビットが消滅し，22 ビットまでしか結果に反映されない．そのため 23 ビット以上格納すると，列ごとに加算を行う際に 23 回以上のビットシフトを行う事になり，指数部が小さいほうの仮数部は全て消滅してしまうため加算結果に影響しなくなる．21 ビット以下の列ごとの加算では逆に近似不足により精度が落ちてしまう．したがって，単精度浮動小数点表現では近似精度とビット長において最も都合が良い 22 ビットをアドレス用に格納している．

4.3　項分割 DA 法

項分割 DA 法のブロック図を以下の 4.5 に示す．ブロック図の設計段階で DA 法の回路に追加でさらに 2 入力浮動小数点乗算器と 2 入力浮動小数点加算器が必要であることが

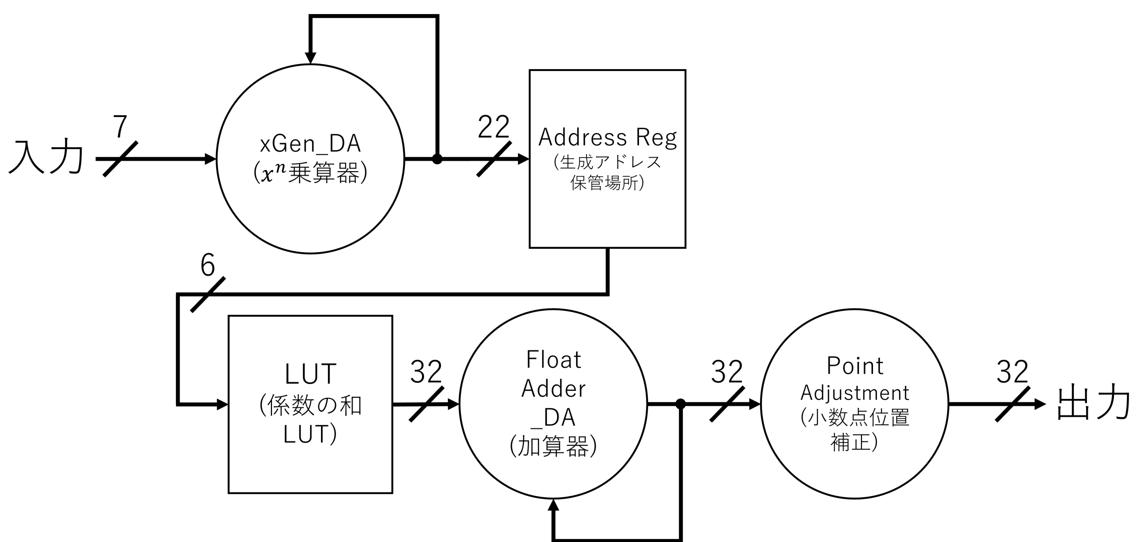

図 4.3　DA 法のブロック図

分かった．メモリの面積縮小化は期待できるが，全体の面積がかえって大きくなってしまうことが予想されたため，VHDL での設計は行わなかった．

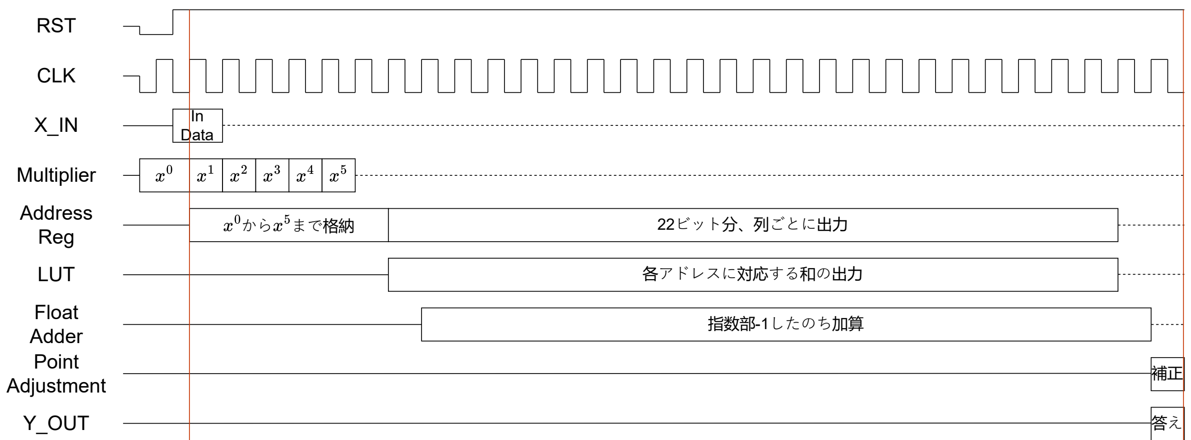

図 4.4　DA 法のタイミングチャート

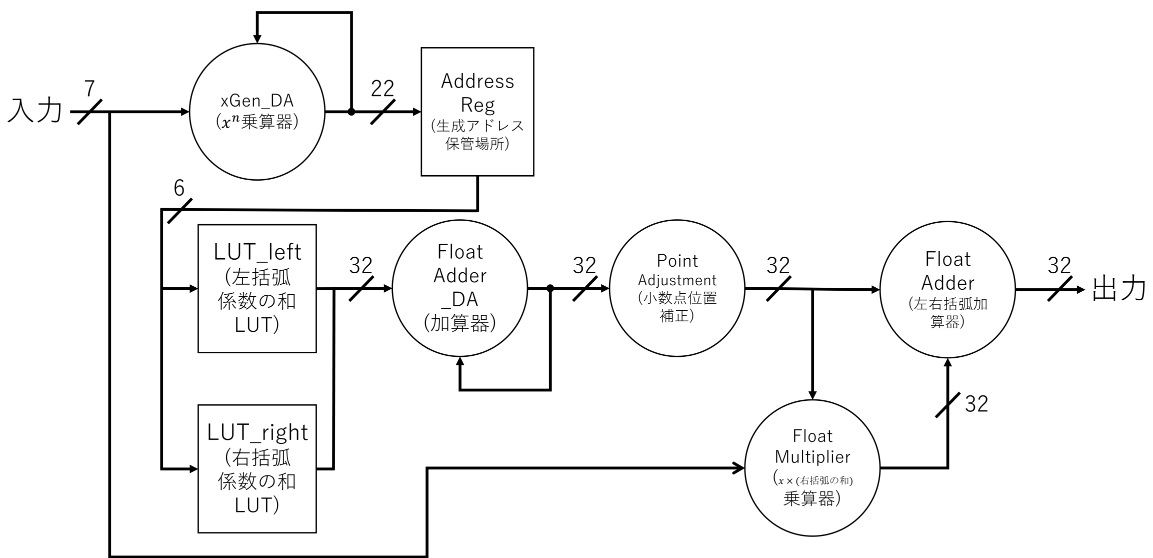

図 4.5　項分割 DA 法のブロック図

第 5 章

評価

この章では設計した回路の面積や遅延時間の違いとその原因について考察する．各回路の回路合成には Synopsys 社製の Design Compiler を用いた．また，0.18μmCMOS ゲートアレイ設計ライブラリを用いて各回路の性能評価を行った．

5.1　シミュレーション結果と誤差

設計した通常演算法回路と DA 法回路において，入力 7 ビットのすべてのパターンでシミュレーションを行った．以下の図 5.1 から 5.4 までにそれぞれの入力が 10 進数の $x = 1.5$ のシミュレーション結果を示す．それぞれの図での入力は上から 3 行目，出力は一番下の行である．

また，全パターンの入力の回路による演算シミュレーション結果と，JavaScript による

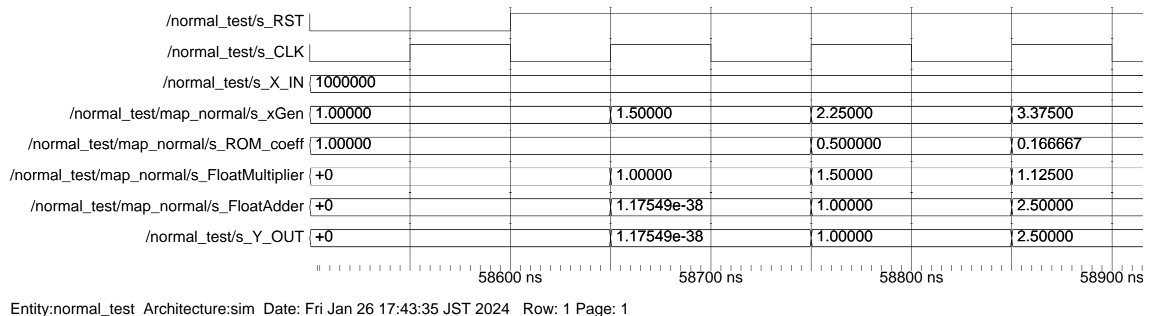

図 5.1　通常演算法回路のシミュレーション結果（前半）

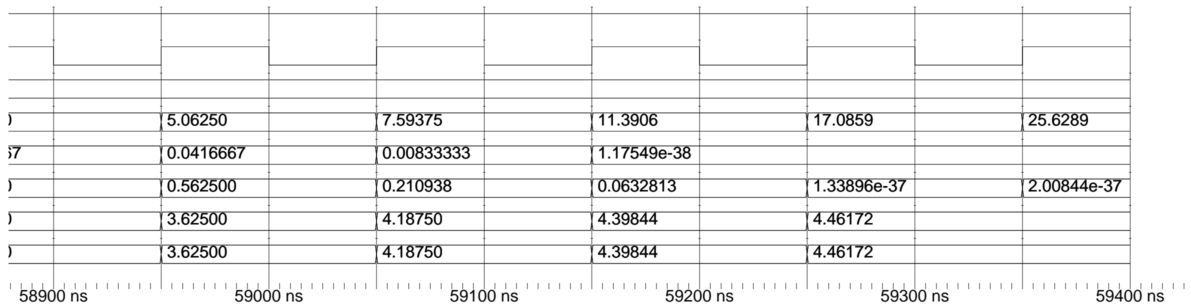

図 5.2　通常演算法回路のシミュレーション結果（後半）

単精度浮動小数演算での元式 e^x，同様にそのテイラー展開式である $f(x)$ との誤差の比較を図 5.5 に示す．

展開式との最大誤差は -0.00007% であり，元式との最大誤差は約 -1.6% であった．図 5.5 からもわかる通り，全ての入力に対して展開式との誤差はおよそ 0% 付近であるが，元式との誤差は入力が大きくなるほど大きくなっている．これはテイラー展開式が 0 を中心とした展開であることために入力が 0 から離れるほど誤差が大きくなることが要因の一つとして考えられる．もう一つの要因はテイラー展開の項数が少ないことによる近似不足である．図 5.6 ではテイラー展開式の項数を 10 項まで 1 項ずつ増やして e^x において $x=2$ を計算したときの元式との誤差を示している．ここでのテイラー展開の項数とは x

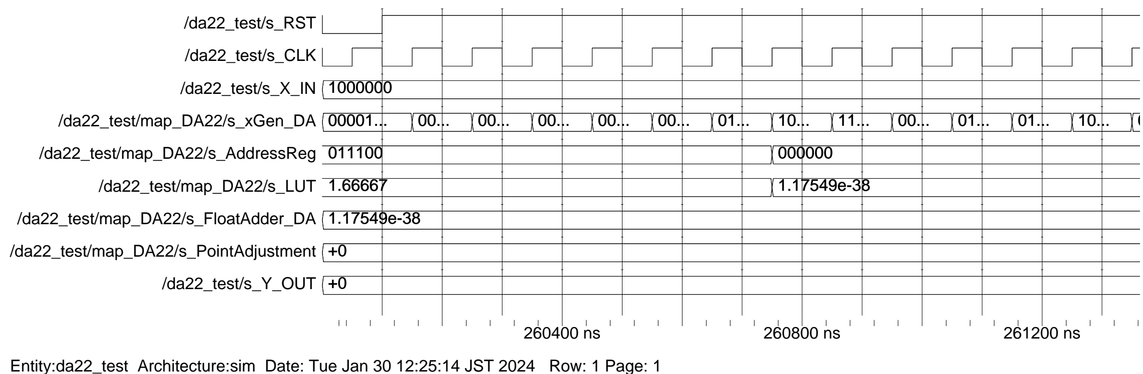

図 5.3　DA 法回路のシミュレーション結果（前半）

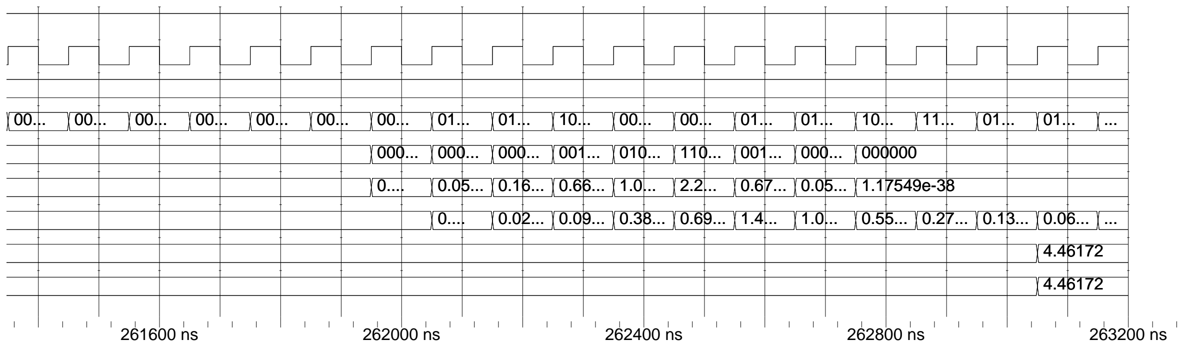

図 5.4　DA 法回路のシミュレーション結果（後半）

が含まれている項を指す．5 項までの展開では誤差が約 -1.66% であるのに対して 6 項では -0.45%，7 項では -0.11% と誤差が小さくなっている．したがって，回路の演算との誤差を小さくするためには計算する項数を増やすことなどが解決案となる．

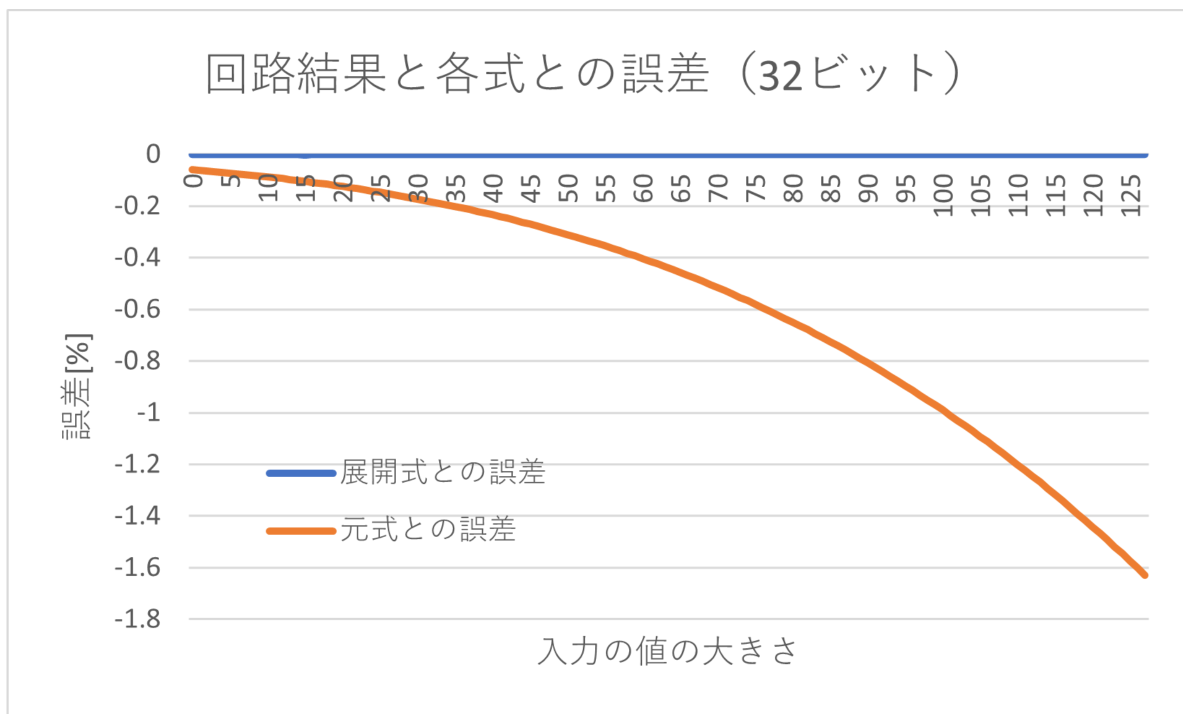

図 5.5　回路演算結果と各式との誤差

5.2 回路面積と遅延時間

通常演算法回路と DA 法回路それぞれの回路面積と遅延時間を以下の表 5.1 に示す.

表 5.1　通常演算法回路と DA 法回路の回路評価

	通常演算法回路	DA 法回路
面積 [μm^2]	94478.33	52583.73
遅延時間 [ns]	30.13	27.38
クロック数 [CLK]	7	30
計算時間 [ns]	210.91	821.40

面積を比較すると DA 法回路は通常演算法回路よりも 44.34% 削減できていることが分かる．遅延時間とは 1CLK あたりに処理に最も時間がかかる回路上の経路を信号が通る際に要する時間である．クロック数は全ての計算過程が終わるまでに要する処理回数である．したがって，遅延時間 × クロック数が計算時間となる．DA 法回路は通常演算法回路

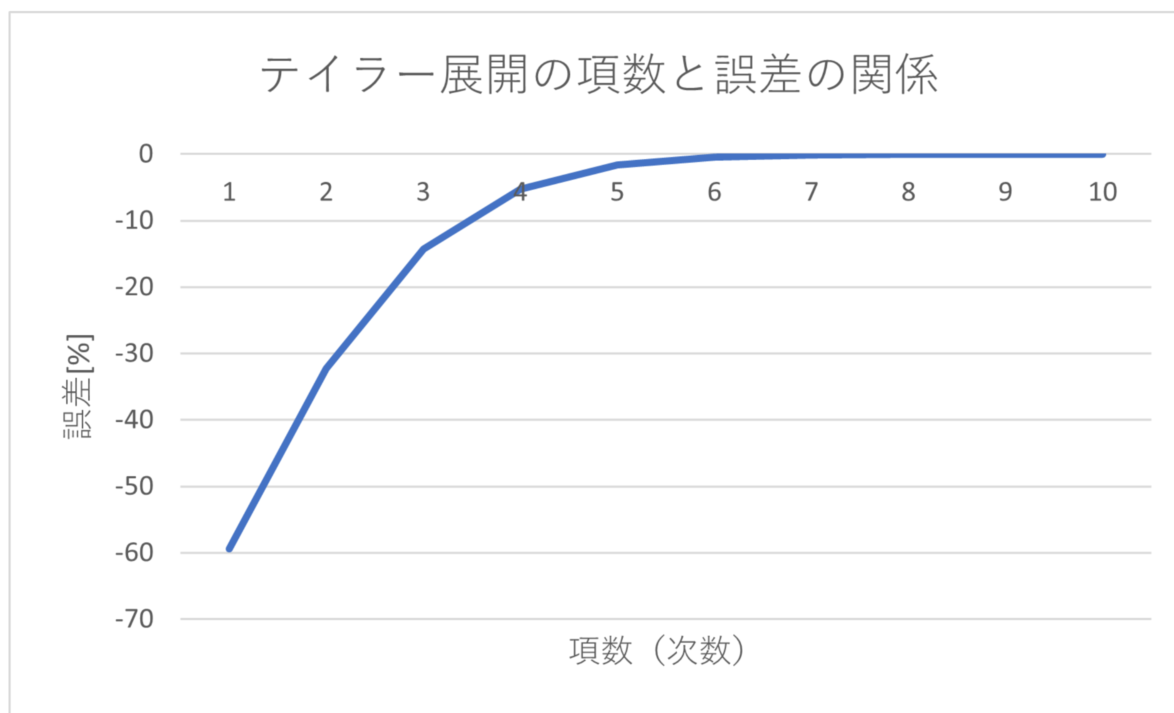

図 5.6　項数と誤差の相関図

に比べて計算時間が約 4 倍となっている．

DA 法回路は乗算器が減ったことにより省面積化を達成できたが，その分加算回数が増えたために計算時間が長くなった．

第 6 章

まとめと今後の課題

本研究では，特に e^x のテイラー展開を計算する積和演算回路において，DA 法を適用することで省面積化，さらにその DA 法を応用した項分割 DA 法を適用することでメモリ部の省面積化を図った．結果として DA 法を適用した積和演算回路では面積が 44.34% 削減できたが，その分計算時間が約 4 倍になった．項分割 DA 法ではメモリ部の省面積化が期待できるが，回路内で実装しようとすると追加で乗算器が必要になる事が想定されたため，かえって面積が大きくなってしまうことが予想された．回路内で制御用回路を実装して一つの乗算器で計算可能にするなどの改良を施すことで計算する式の項数が多い際の省面積化は期待できる．

今回は e^x を x^5 までテイラー展開した際の専用演算回路を設計したため，今後の課題としては特定の式を計算するだけでなく積和演算回路としての汎用性を高める改良があげられる．また，項分割 DA 法の計算アルゴリズムを改良して省面積回路の設計を行うことも課題となる．

参考文献

[1] J. Wei, A. Kuwana, H. Kobayashi, K. Kubo, ”Revisit to floating-point division algorithm based on Taylor-series expansion,” 16th IEEE Asia Pacific Conference on Circuits and Systems, Ha Long Bay, Vietnam, (Dec. 2020).

[2] J. Wei, A. Kuwana, H. Kobayashi, K. Kubo, Y. Tanaka, “Floating-point inverse square root algorithm based on Taylor-series expansion,” IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 68, No. 7, pp. 2640–2644 (Jul. 2021).

[3] J. Wei, A. Kuwana, H. Kobayashi, K. Kubo, “Divide and conquer: floating-point exponential calculation based on Taylor-series expansion,” IEEE 14th International Conference on ASIC, Kunming, China (Oct. 2021).

[4] J. Wei, A. Kuwana, H. Kobayashi, K. Kubo, “IEEE754 binary32 floating-point logarithmic algorithms based on Taylor-series expansion with mantissa region conversion and division,” IEICE Trans. Fundamentals, Vol. E105-A, No. 7, pp. 1020–1027 (Jul. 2022).

[5] M. T. Khan, R. A. Shaik, “High-performance VLSI architecture of DLMS adaptive filter for fast-convergence and low-MSE,” IEEE Transactions on Circuits and Systems Ⅱ: Express Briefs, Vol. 69, No. 4, pp. 2106–2110 (Apr. 2022).

[6] D. J. Allred, H. Yoo, V. Krishnan, W. Huang, D. V. Anderson, “LMS adaptive filters using distributed arithmetic for high throughput,” IEEE Transactions on Circuits and Systems I: Regular Papers, Vol. 52, No. 7, pp. 1327–1337 (Jul. 2005)

[7] R. Guo, L. S. DeBrunner, “Two high-performance adaptive filter implementation schemes using distributed arithmetic,” IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 58, No. 9, pp. 600–604 (Sept. 2011).

[8] M. S. Prakash, R. A. Shaik, “Low-area and high-throughput architecture for an adaptive filter using distributed arithmetic,” IEEE Transactions on Circuits and

Systems II: Express Briefs, Vol. 60, No. 11, pp. 781–785 (Nov. 2013).

[9] S. Y. Park, P. K. Meher, “Low-power, high-throughput, and low-area adaptive FIR filter based on distributed arithmetic,” IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 60, No. 6, pp. 346–350 (Jun. 013).

[10] M. T. Khan, R. A. Shaik, S. P. Matcha, “Improved convergent distributed arithmetic based low complexity pipelined least-mean-square filter,” IET Circuits, Devices & Systems, Vol. 12, No. 6, pp. 792–801 (May. 2018).

[11] M. T. Khan, R. A. Shaik, “Optimal complexity architectures for pipelined distributed arithmetic-based LMS adaptive filter,” IEEE Transactions on Circuits and Systems I: Regular Papers, Vol. 66, No. 2, pp. 630–642 (Feb. 2019).

[12] S. Ahmad, S. G. Khawaja, N. Amjad, M. Usman, “A novel multiplier-less LMS adaptive filter design based on offset binary coded distributed arithmetic,” IEEE Access, Vol. 9, pp. 78138–78152 (May. 2021).

[13] R. Bala, S. Aktar, “Fast Fourier transformation realization with distributed arithmetic,” International Journal of Computer Applications, Vol. 102, No. 15, pp. 22–25 (Sept. 2014).

[14] E. Özalevli, W. Huang, P. E. Hasler, D. V. Anderson, “A reconfigurable mixed-signal VLSI implementation of distributed arithmetic used for finite-impulse response filtering,” IEEE Trans. Circuits and Systems - I: Regular Papers, Vol. 55, No. 2, pp. 510–521 (Mar. 2008).

謝辞

本研究の遂行にあたり，指導教員として日頃から多大な指導をしていただいた，群馬大学名誉教授教授の小林春夫先生，並びに群馬大学大学院理工学府電子情報・数理教育プログラム准教授の田中勇樹先生に厚くお礼申し上げます．

付録 A

ソースコードリスト

本研究に用いた通常演算法と DA 法それぞれの回路作成に必要なソースコードを付録する.

VHDL ソースコード		
ファイル名	ファイル内で宣言されている エンティティ名	generic 宣言の 変数名と意味
	コンポーネントとして 読み込むエンティティ	エンティティの 入出力ポート
	エンティティの役割	

VHDL ソースコード		
Normal.vhdl	Normal	
	xGen ROM_coeff FloatMultiplier FloatAdder	入力: CLK_TOP (1bit) RST_TOP (1bit) X_IN_TOP (7bit) 出力: s_FloatAdder (32bit)
	TOP エンティティ	
xGen.vhdl	xGen	
		入力: CLK (1bit) RST (1bit) X_IN (7bit) 出力: Y_OUT (32bit)
	x のべき乗の乗算回路	
ROM_coeff.vhdl	ROM_coeff	
		入力: CLK (1bit) RST (1bit) 出力: DATA (32bit)
	係数を保管・出力する ROM	
FloatMultiplier.vhdl	FloatMultiplier	
		入力: CLK (1bit) RST (1bit) X_IN1 (32bit) X_IN2 (32bit) 出力: Y_OUT (32bit)
	x のべき乗と係数との乗算回路	
FloatAdder.vhdl	FloatAdder	
		入力: CLK (1bit) RST (1bit) X_IN (32bit) 出力: Y_OUT (32bit)
	全ての項の加算回路	

VHDL ソースコード		
DA22.vhdl	DA22	
	AddressReg LUT FloatAdder_DA PointAdjustment	入力: CLK_TOP (1bit) RST_TOP (1bit) X_IN_TOP (7bit) 出力: s_PointAdjustment (32bit)
	TOP エンティティ	
xGen_DA.vhdl	xGen_DA	
		入力: CLK (1bit) RST (1bit) X_IN (7bit) 出力: Y_OUT (22bit)
	x のべき乗の乗算回路	
AddressReg.vhdl	AddressReg	
		入力: CLK (1bit) RST (1bit) inDATA (22bit) 出力: EN (1bit) Address (6bit)
	x のべき乗をアドレスに変換出力	
LUT.vhdl	LUT	
		入力: Address (6bit) 出力: Ingredients (32bit)
	アドレスに応じて係数の和を出力	

VHDL ソースコード		
FloatAdder_DA.vhdl	FloatAdder_DA	
	xGen_DA	入力: CLK_TOP (1bit) RST_TOP (1bit) X_IN (32bit) EN_IN (1bit) 出力: Y_OUT (32bit) EN_OUT (1bit)
	加算回路	
PointAdjustment.vhdl	PointAdjustment	
		入力: CLK (1bit) RST (1bit) X_IN (32bit) EN_IN (1bit) 出力: Y_OUT (32bit)
	小数点の位置補正回路	

A.1 通常演算法

Listing A.1 通常演算法回路のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------

entity Normal is
    port(
        CLK_TOP             :   in  std_logic;
        RST_TOP             :   in  std_logic;
        X_IN_TOP            :   in  std_logic_vector(6 downto 0);
        s_FloatAdder            :   out std_logic_vector(31 downto 0)
    );
end Normal;


-----アーキテクチャ定義-----

architecture RTL of Normal is

    component xGen is
        port(
            CLK             :   in  std_logic;
            RST             :   in  std_logic;
            X_IN            :   in  std_logic_vector(6 downto 0);
            Y_OUT           :   out std_logic_vector(31 downto 0)
        );
    end component;

    component ROM_coeff is
        port(
            CLK     : in  std_logic;
            RST     : in  std_logic;
            DATA    : out std_logic_vector(31 downto 0)
        );
    end component;

    component FloatMultiplier is
        port(
            CLK             :   in  std_logic;
            RST             :   in  std_logic;
            X_IN1           :   in  std_logic_vector(31 downto 0);
            X_IN2           :   in  std_logic_vector(31 downto 0);
            Y_OUT           :   out std_logic_vector(31 downto 0)
        );
    end component;

    component FloatAdder is
        port(
```

```
50            CLK             :   in  std_logic;
51            RST             :   in  std_logic;
52            X_IN            :   in  std_logic_vector(31 downto 0);
53            Y_OUT           :   out std_logic_vector(31 downto 0)
54        );
55    end component;
56
57
58    signal  s_xGen     :   std_logic_vector(31 downto 0);
59    signal  s_ROM_coeff      :   std_logic_vector(31 downto 0);
60    signal  s_FloatMultiplier         :   std_logic_vector(31 downto 0);
61
62 begin
63
64    map_xGen     :   xGen port map(
65        CLK     =>      CLK_TOP,
66        RST     =>      RST_TOP,
67        X_IN    =>      X_IN_TOP,
68        Y_OUT   =>      s_xGen
69    );
70
71    map_ROM_coeff :   ROM_coeff port map(
72        CLK     =>      CLK_TOP,
73        RST     =>      RST_TOP,
74        DATA    =>      s_ROM_coeff
75    );
76
77    map_FloatMultiplier   :   FloatMultiplier port map(
78        CLK     =>      CLK_TOP,
79        RST     =>      RST_TOP,
80        X_IN1   =>      s_xGen,
81        X_IN2   =>      s_ROM_coeff,
82        Y_OUT   =>      s_FloatMultiplier
83    );
84
85    map_FloatAdder    :   FloatAdder port map(
86        CLK     =>      CLK_TOP,
87        RST     =>      RST_TOP,
88        X_IN    =>      s_FloatMultiplier,
89        Y_OUT   =>      s_FloatAdder
90    );
91
92
93 end RTL;
```

Listing A.2　xGen のソースコード

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4
5 --エンティティ宣言
6
7 entity xGen is
8     port(
```

```
 9          CLK             :   in  std_logic;
10          RST             :   in  std_logic;
11          X_IN            :   in  std_logic_vector(6 downto 0);
12          Y_OUT           :   out std_logic_vector(31 downto 0)
13      );
14  end xGen;
15
16
17  --アーキテクチャ定義
18
19  architecture RTL of xGen is
20      signal Q1                   : std_logic_vector(22 downto 0);
21      signal Q2                   : std_logic_vector(7 downto 0);
22      signal Q1_SIG               : std_logic_vector(23 downto 0);
23      signal X_SIG                : std_logic_vector(7 downto 0);
24      signal Y_SIG_IN             : std_logic_vector(31 downto 0);
25      signal Y_SIG_OUT            : std_logic_vector(22 downto 0);
26      signal X_EXP                : std_logic_vector(7 downto 0);
27
28  begin
29      mux : process (Y_SIG_IN)                                   --マルチプレクサで判定
30      begin
31          if (Y_SIG_IN(31) = '1') then
32              Y_SIG_OUT  <=  Y_SIG_IN(30 downto 8);
33          else
34              Y_SIG_OUT  <=  Y_SIG_IN(29 downto 7);
35          end if;
36      end process;
37
38
39      gen_man_reg : process (RST, CLK)              --仮数レジスタ
40      begin
41          if (RST = '0') then                                          --リセット信号にノットがついている想定
42              Q1  <=  "00000000000000000000000";                                  --初期値、最下位ビット１それ以外０
43          elsif (CLK'event and CLK = '1') then
44              Q1  <=  Y_SIG_OUT;
45          end if;
46      end process;
47
48      gen_exp_reg : process (RST, CLK)              --指数レジスタ
49      begin
50          if (RST = '0')then
51              Q2  <=  "01111111";            --初期値126
52          elsif (CLK'event and CLK = '1') then
53              Q2  <=  X_EXP;
54          end if;
55      end process;
56
57      X_SIG   <=  '1' & X_IN;
58      Q1_SIG  <=  '1' & Q1;
59      Y_SIG_IN   <=  X_SIG * Q1_SIG;
60      X_EXP   <=  "0000000"&Y_SIG_IN(31) + Q2;
61      Y_OUT   <=  '0' & Q2 & Q1;
62
63  end RTL;
```

Listing A.3　ROM_coeff のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------
entity ROM_coeff is
    port
    (
        CLK     : in  std_logic;
        RST     : in  std_logic;
        DATA    : out std_logic_vector(31 downto 0)
    );
end ROM_coeff;

-----アーキテクチャ定義-----
architecture RTL of ROM_Coeff is

    type ROM_type is array (7 downto 0) of std_logic_vector(31 downto 0);

    constant Coeff  :   ROM_type:=
    (
        0       => "00111111100000000000000000000000",
        1       => "00111111000000000000000000000000",
        2       => "00111110001010101010101010101011",
        3       => "00111101001010101010101010101011",
        4       => "00111100000010001000100010001001",
        5       => "00000000100000000000000000000000",
        6       => "00000000100000000000000000000000",
        7       => "00000000100000000000000000000000"
    );

    begin
        process(RST, CLK)
        variable ADDR : std_logic_vector(2 downto 0):="000";
        begin
            if (RST = '0') then
                ADDR := "000";
                DATA <= "00111111100000000000000000000000";
            elsif (CLK'event and CLK = '1') then
                DATA <= Coeff(conv_integer(ADDR));
                ADDR := ADDR + "001";
            end if;
        end process;

    end RTL;
```

Listing A.4　Float Multiplier のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------

```

```
7   entity FloatMultiplier is
8       port(
9           CLK             :   in  std_logic;
10          RST             :   in  std_logic;
11          X_IN1           :   in  std_logic_vector(31 downto 0);
12          X_IN2           :   in  std_logic_vector(31 downto 0);
13          Y_OUT           :   out std_logic_vector(31 downto 0)
14      );
15  end FloatMultiplier;
16
17
18  -----アーキテクチャ定義-----
19
20  architecture RTL of FloatMultiplier is
21      signal Q_exp                    :   std_logic_vector(8 downto 0);   --指数
22
23      signal Q_man                    :   std_logic_vector(47 downto 0);  --仮数
24
25      signal Q_sf                     :   std_logic_vector(31 downto 0);  --出力
26
27      constant sig : std_logic_vector(8 downto 0) := "001111111";
28
29      begin
30
31          mux : process (RST, CLK)                                    --マルチプレクサで判定
32          begin
33              if (RST = '0') then
34                  Q_sf <= (others => '0');
35              elsif (CLK'event and CLK = '1') then
36                  if (Q_man(47) = '1') then
37                      Q_sf  <=  '0' & Q_exp(7 downto 0) & Q_man(46 downto 24);
38                  else
39                      Q_sf  <=  '0' & Q_exp(7 downto 0) & Q_man(45 downto 23);
40                  end if;
41              end if;
42          end process;
43  -------------------------------------------------------------
44      Q_exp <= ('0'&X_IN1(30 downto 23)) + ('0'&X_IN2(30 downto 23)) - sig + ("00000000"&Q_man(47));
45  -------------------------------------------------------------
46      Q_man <= ('1'&X_IN1(22 downto 0)) * ('1'&X_IN2(22 downto 0));
47      Y_OUT <= Q_sf;
48
49  end RTL;
```

Listing A.5　Float Adder のソースコード

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.std_logic_unsigned.all;
4
5   -----エンティティ宣言------
6
7   entity FloatAdder is
8       port(
9           CLK             :   in  std_logic;
```

```
10          RST             :   in  std_logic;
11          X_IN            :   in  std_logic_vector(31 downto 0);
12          Y_OUT           :   out std_logic_vector(31 downto 0)
13      );
14  end FloatAdder;
15
16  -----アーキテクチャ定義-----
17
18  architecture RTL of FloatAdder is
19      signal      SUM         :   std_logic_vector(31 downto 0);
20      signal      BIG         :   std_logic_vector(31 downto 0);
21      signal      SMALL       :   std_logic_vector(31 downto 0);
22
23      signal      dEXP        :   std_logic_vector( 7 downto 0);
24      signal      EXP1        :   std_logic_vector( 7 downto 0);
25      signal      BIG_M       :   std_logic_vector(22 downto 0);
26      signal      SMALL_M     :   std_logic_vector(22 downto 0);
27
28      signal      EXP2        :   std_logic_vector( 7 downto 0);
29      signal      BIG_Ms      :   std_logic_vector(24 downto 0);
30      signal      SMALL_Ms    :   std_logic_vector(24 downto 0);
31
32      signal      SUM_M       :   std_logic_vector(24 downto 0);
33      signal      SUM_E       :   std_logic_vector( 7 downto 0);
34
35      function barrel_shift (
36          v   : std_logic_vector(23 downto 0) := (others => '0');
37          num : std_logic_vector( 7 downto 0) := (others => '0')
38              ) return std_logic_vector is
39          variable tmp : std_logic_vector(23 downto 0);
40      begin
41          case(num) is
42
43              when X"00"  =>  tmp   :=  v;
44              when X"01"  =>  tmp   :=  ('0'&v(23 downto 1));
45              when X"02"  =>  tmp   :=  ("00"&v(23 downto 2));
46              when X"03"  =>  tmp   :=  ("000"&v(23 downto 3));
47              when X"04"  =>  tmp   :=  ("0000"&v(23 downto 4));
48              when X"05"  =>  tmp   :=  ("00000"&v(23 downto 5));
49              when X"06"  =>  tmp   :=  ("000000"&v(23 downto 6));
50              when X"07"  =>  tmp   :=  ("0000000"&v(23 downto 7));
51              when X"08"  =>  tmp   :=  ("00000000"&v(23 downto 8));
52              when X"09"  =>  tmp   :=  ("000000000"&v(23 downto 9));
53              when X"0A"  =>  tmp   :=  ("0000000000"&v(23 downto 10));
54              when X"0B"  =>  tmp   :=  ("00000000000"&v(23 downto 11));
55              when X"0C"  =>  tmp   :=  ("000000000000"&v(23 downto 12));
56              when X"0D"  =>  tmp   :=  ("0000000000000"&v(23 downto 13));
57              when X"0E"  =>  tmp   :=  ("00000000000000"&v(23 downto 14));
58              when X"0F"  =>  tmp   :=  ("000000000000000"&v(23 downto 15));
59              when X"10"  =>  tmp   :=  ("0000000000000000"&v(23 downto 16));
60              when X"11"  =>  tmp   :=  ("00000000000000000"&v(23 downto 17));
61              when X"12"  =>  tmp   :=  ("000000000000000000"&v(23 downto 18));
62              when X"13"  =>  tmp   :=  ("0000000000000000000"&v(23 downto 19));
63              when X"14"  =>  tmp   :=  ("00000000000000000000"&v(23 downto 20));
64              when X"15"  =>  tmp   :=  ("000000000000000000000"&v(23 downto 21));
```

```
65             when X"16"  =>  tmp   :=  ("0000000000000000000000"&v(23 downto 22));
66             when X"17"  =>  tmp   :=  ("00000000000000000000000"&v(23));
67             when others =>  tmp   :=  (others => '0');
68         end case ;
69         return tmp;
70     end function;
71 
72     begin
73 
74     Comparation : process(X_IN, SUM)
75     begin
76                 if (X_IN(30 downto 23) > SUM(30 downto 23)) then
77                     BIG     <=  X_IN;
78                     SMALL   <=  SUM;
79                 else
80                     BIG     <=  SUM;
81                     SMALL   <=  X_IN;
82                 end if;
83     end process;
84 
85     --Preparation
86                 dEXP        <=  BIG(30 downto 23) - SMALL(30 downto 23);
87                 EXP1        <=  BIG(30 downto 23);
88                 BIG_M       <=  BIG(22 downto 0);
89                 SMALL_M     <=  SMALL(22 downto 0);
90 
91 
92     --Judge
93                 EXP2        <=  EXP1;
94                 BIG_Ms      <=  ("01" & BIG_M);
95                 SMALL_Ms    <=  ('0'  & barrel_shift(('1'&SMALL_M), dEXP));
96 
97 
98     gen_SUM : process(RST, CLK)
99     begin
100         if (RST = '0') then
101             SUM     <=  (others => '0');
102         elsif (CLK'event and CLK = '1') then
103             if (SUM_M(24) = '1') then
104                 SUM  <=  '0' & SUM_E & SUM_M(23 downto 1);
105             else
106                 SUM  <=  '0' & SUM_E & SUM_M(22 downto 0);
107             end if;
108         end if;
109     end process;
110 
111 
112 
113     SUM_M       <=  BIG_Ms + SMALL_Ms;
114     SUM_E       <=  EXP2 + SUM_M(24);
115     Y_OUT       <=  SUM;
116 
117 end RTL;
```

A.2　DA 法

Listing A.6　DA 法回路のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------

entity DA22 is
    port(
        CLK_TOP             :   in  std_logic;
        RST_TOP             :   in  std_logic;
        X_IN_TOP            :   in  std_logic_vector(6 downto 0);
        s_PointAdjustment           :   out std_logic_vector(31 downto 0)
    );
end DA22;


-----アーキテクチャ定義-----

architecture RTL of DA22 is

    component xGen_DA is
        port(
            CLK             :   in  std_logic;
            RST             :   in  std_logic;
            X_IN            :   in  std_logic_vector(6 downto 0);
            Y_OUT           :   out std_logic_vector(21 downto 0)
        );
    end component;

    component AddressReg is
        port(
            RST         : in  std_logic;
            CLK         : in  std_logic;
            inDATA      : in  std_logic_vector(21 downto 0);
            EN          : out std_logic;
            Address     : out std_logic_vector(5 downto 0)
        );
    end component;

    component LUT is
        port(
            Address     : in  std_logic_vector(5 downto 0);
            Ingredients : out std_logic_vector(31 downto 0)
        );
    end component;

    component FloatAdder_DA is
        port(
            CLK             :   in   std_logic;
```

```
50            RST             :   in   std_logic;
51            EN_IN           :   in   std_logic;
52            EN_OUT          :   out  std_logic;
53            X_IN            :   in   std_logic_vector(31 downto 0);
54            Y_OUT           :   out  std_logic_vector(31 downto 0)
55        );
56    end component;
57
58    component PointAdjustment is
59        port(
60            CLK                :   in  std_logic;
61            RST                :   in  std_logic;
62            EN_IN              :   in  std_logic;
63            X_IN               :   in  std_logic_vector(31 downto 0);
64            Y_OUT              :   out std_logic_vector(31 downto 0)
65        );
66    end component;
67
68
69    signal  s_xGen_DA  :std_logic_vector(21 downto 0);
70    signal  s_AddressReg   :std_logic_vector( 5 downto 0);
71    signal  EN1             :std_logic;
72    signal  s_LUT           :std_logic_vector(31 downto 0);
73    signal  EN2             :std_logic;
74    signal  s_FloatAdder_DA             :std_logic_vector(31 downto 0);
75
76 begin
77
78    map_xGen_DA :   xGen_DA port map(
79        CLK         =>  CLK_TOP,
80        RST         =>  RST_TOP,
81        X_IN        =>  X_IN_TOP,
82        Y_OUT       =>  s_xGen_DA
83    );
84
85    map_AddressReg :   AddressReg port map(
86        CLK         =>  CLK_TOP,
87        RST         =>  RST_TOP,
88        inDATA      =>  s_xGen_DA,
89        EN          =>  EN1,
90        Address     =>  s_AddressReg
91    );
92
93    map_LUT   :   LUT port map(
94        Address     =>  s_AddressReg,
95        Ingredients =>  s_LUT
96    );
97
98    map_FloatAdder_DA  :   FloatAdder_DA port map(
99        CLK         =>  CLK_TOP,
100       RST         =>  RST_TOP,
101       EN_IN       =>  EN1,
102       EN_OUT      =>  EN2,
103       X_IN        =>  s_LUT,
104       Y_OUT       =>  s_FloatAdder_DA
```

```
105     );
106
107     map_PointAdjustment   :   PointAdjustment port map(
108         CLK         =>  CLK_TOP,
109         RST         =>  RST_TOP,
110         EN_IN       =>  EN2,
111         X_IN        =>  s_FloatAdder_DA,
112         Y_OUT       =>  s_PointAdjustment
113     );
114
115
116 end RTL;
```

Listing A.7　xGen_DA のソースコード

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4
5  --エンティティ宣言
6
7  entity xGen_DA is
8      port(
9          CLK             :   in  std_logic;
10         RST             :   in  std_logic;
11         X_IN            :   in  std_logic_vector(6 downto 0);
12         Y_OUT           :   out std_logic_vector(21 downto 0)
13     );
14 end xGen_DA;
15
16
17 --アーキテクチャ定義
18
19 architecture RTL of xGen_DA is
20     signal Q1           : std_logic_vector(21 downto 0);
21     signal Y_sig        : std_logic_vector(29 downto 0);
22
23 begin
24     gen_reg : process (RST, CLK)
25     begin
26         if (RST = '0') then                         --リセット信号にノットがついている想定
27             Q1  <= (17 => '1', others => '0');        --初期値、最下位ビット１それ以外０
28         elsif (CLK'event and CLK = '1') then
29             Q1  <= Y_sig(28 downto 7);
30         end if;
31     end process;
32
33     Y_sig <= ('1' & X_IN) * Q1;
34     Y_OUT <= Q1;
35
36 end RTL;
```

Listing A.8　AddressReg のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------
entity AddressReg is
    port
    (
        RST         : in  std_logic;
        CLK         : in  std_logic;
        inDATA      : in  std_logic_vector(21 downto 0);
        EN          : out std_logic;
        Address     : out std_logic_vector(5 downto 0)
    );
end AddressReg;

-----アーキテクチャ定義-----
architecture RTL of AddressReg is
    signal  EN_o    :   std_logic;

    type mem_type is array (5 downto 0) of std_logic_vector(21 downto 0);
    signal mem  : mem_type;

begin

    process (RST, CLK)
    variable i      : integer range 0 to 21 := 0;
    variable addr   : integer range 0 to 6  := 0;
    begin
        if (RST = '0') then
            i       := 0;
            addr    := 0;
            EN_o    <= '0';
        elsif (CLK'event and CLK = '1') then
            if (addr /= 6) then
                mem(addr) <= inDATA;
                addr := addr + 1;
            else
                EN_o    <=  '1';
                Address <= mem(0)(i) & mem(1)(i) & mem(2)(i) & mem(3)(i) & mem(4)(i) & mem(5)(i);
                if (i /= 21) then
                    i := i + 1;
                else
                    i := 0;
                end if;
            end if;
        end if;
    end process;

    EN  <=  EN_o;

end RTL;
```

Listing A.9　LUT のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------
entity LUT is
    port
    (
        Address     : in  std_logic_vector(5 downto 0);
        Ingredients : out std_logic_vector(31 downto 0)
    );
end LUT;

-----アーキテクチャ定義-----
architecture RTL of LUT is
begin
    with Address select
        Ingredients <=  X"3c088889"     when    "000001",
                        X"3d2aaaab"     when    "000010",
                        X"3d4ccccd"     when    "000011",
                        X"3e2aaaab"     when    "000100",
                        X"3e333333"     when    "000101",
                        X"3e555555"     when    "000110",
                        X"3e5dddde"     when    "000111",
                        X"3f000000"     when    "001000",
                        X"3f022222"     when    "001001",
                        X"3f0aaaab"     when    "001010",
                        X"3f0ccccd"     when    "001011",
                        X"3f2aaaab"     when    "001100",
                        X"3f2ccccd"     when    "001101",
                        X"3f355555"     when    "001110",
                        X"3f377777"     when    "001111",
                        X"3f800000"     when    "010000",
                        X"3f811111"     when    "010001",
                        X"3f855555"     when    "010010",
                        X"3f866666"     when    "010011",
                        X"3f955555"     when    "010100",
                        X"3f966666"     when    "010101",
                        X"3f9aaaab"     when    "010110",
                        X"3f9bbbbc"     when    "010111",
                        X"3fc00000"     when    "011000",
                        X"3fc11111"     when    "011001",
                        X"3fc55555"     when    "011010",
                        X"3fc66666"     when    "011011",
                        X"3fd55555"     when    "011100",
                        X"3fd66666"     when    "011101",
                        X"3fdaaaab"     when    "011110",
                        X"3fdbbbbc"     when    "011111",
                        X"3f800000"     when    "100000",
                        X"3f811111"     when    "100001",
                        X"3f855555"     when    "100010",
                        X"3f866666"     when    "100011",
                        X"3f955555"     when    "100100",
                        X"3f966666"     when    "100101",
                        X"3f9aaaab"     when    "100110",
```

```
56                          X"3f9bbbbc"     when    "100111",
57                          X"3fc00000"     when    "101000",
58                          X"3fc11111"     when    "101001",
59                          X"3fc55555"     when    "101010",
60                          X"3fc66666"     when    "101011",
61                          X"3fd55555"     when    "101100",
62                          X"3fd66666"     when    "101101",
63                          X"3fdaaaab"     when    "101110",
64                          X"3fdbbbbc"     when    "101111",
65                          X"40000000"     when    "110000",
66                          X"40008889"     when    "110001",
67                          X"4002aaab"     when    "110010",
68                          X"40033333"     when    "110011",
69                          X"400aaaab"     when    "110100",
70                          X"400b3333"     when    "110101",
71                          X"400d5555"     when    "110110",
72                          X"400dddde"     when    "110111",
73                          X"40200000"     when    "111000",
74                          X"40208889"     when    "111001",
75                          X"4022aaab"     when    "111010",
76                          X"40233333"     when    "111011",
77                          X"402aaaab"     when    "111100",
78                          X"402b3333"     when    "111101",
79                          X"402d5555"     when    "111110",
80                          X"402dddde"     when    "111111",
81                          "00000000100000000000000000000000"     when    others;
82
83      end RTL;
```

Listing A.10 FloatAdder_DA のソースコード

```
 1  library IEEE;
 2  use IEEE.std_logic_1164.all;
 3  use IEEE.std_logic_unsigned.all;
 4
 5  -----エンティティ宣言------
 6
 7  entity FloatAdder_DA is
 8      port(
 9          CLK             :   in   std_logic;
10          RST             :   in   std_logic;
11          EN_IN           :   in   std_logic;
12          EN_OUT          :   out  std_logic;
13          X_IN            :   in   std_logic_vector(31 downto 0);
14          Y_OUT           :   out  std_logic_vector(31 downto 0)
15      );
16  end FloatAdder_DA;
17
18  -----アーキテクチャ定義-----
19
20  architecture RTL of FloatAdder_DA is
21      signal      SUM         :   std_logic_vector(31 downto 0);
22      signal      BIG         :   std_logic_vector(31 downto 0);
23      signal      SMALL       :   std_logic_vector(31 downto 0);
24
```

```
25  signal      dEXP        :   std_logic_vector( 7 downto 0);
26  signal      EXP1        :   std_logic_vector( 7 downto 0);
27  signal      BIG_M       :   std_logic_vector(22 downto 0);
28  signal      SMALL_M     :   std_logic_vector(22 downto 0);
29
30  signal      EXP2        :   std_logic_vector( 7 downto 0);
31  signal      BIG_Ms      :   std_logic_vector(24 downto 0);
32  signal      SMALL_Ms    :   std_logic_vector(24 downto 0);
33
34  signal      SUM_M       :   std_logic_vector(24 downto 0);
35  signal      SUM_E       :   std_logic_vector( 7 downto 0);
36
37  signal      EN_O        :   std_logic;              --変更点
38
39  function barrel_shift (
40      v   : std_logic_vector(23 downto 0) := (others => '0');
41      num : std_logic_vector( 7 downto 0) := (others => '0')
42          ) return std_logic_vector is
43      variable tmp : std_logic_vector(23 downto 0);
44  begin
45      case(num) is
46
47          when X"00"  =>  tmp   :=  v;
48          when X"01"  =>  tmp   :=  ('0'&v(23 downto 1));
49          when X"02"  =>  tmp   :=  ("00"&v(23 downto 2));
50          when X"03"  =>  tmp   :=  ("000"&v(23 downto 3));
51          when X"04"  =>  tmp   :=  ("0000"&v(23 downto 4));
52          when X"05"  =>  tmp   :=  ("00000"&v(23 downto 5));
53          when X"06"  =>  tmp   :=  ("000000"&v(23 downto 6));
54          when X"07"  =>  tmp   :=  ("0000000"&v(23 downto 7));
55          when X"08"  =>  tmp   :=  ("00000000"&v(23 downto 8));
56          when X"09"  =>  tmp   :=  ("000000000"&v(23 downto 9));
57          when X"0A"  =>  tmp   :=  ("0000000000"&v(23 downto 10));
58          when X"0B"  =>  tmp   :=  ("00000000000"&v(23 downto 11));
59          when X"0C"  =>  tmp   :=  ("000000000000"&v(23 downto 12));
60          when X"0D"  =>  tmp   :=  ("0000000000000"&v(23 downto 13));
61          when X"0E"  =>  tmp   :=  ("00000000000000"&v(23 downto 14));
62          when X"0F"  =>  tmp   :=  ("000000000000000"&v(23 downto 15));
63          when X"10"  =>  tmp   :=  ("0000000000000000"&v(23 downto 16));
64          when X"11"  =>  tmp   :=  ("00000000000000000"&v(23 downto 17));
65          when X"12"  =>  tmp   :=  ("000000000000000000"&v(23 downto 18));
66          when X"13"  =>  tmp   :=  ("0000000000000000000"&v(23 downto 19));
67          when X"14"  =>  tmp   :=  ("00000000000000000000"&v(23 downto 20));
68          when X"15"  =>  tmp   :=  ("000000000000000000000"&v(23 downto 21));
69          when X"16"  =>  tmp   :=  ("0000000000000000000000"&v(23 downto 22));
70          when X"17"  =>  tmp   :=  ("00000000000000000000000"&v(23));
71          when others =>  tmp   :=  (others => '0');
72      end case ;
73      return tmp;
74  end function;
75
76  begin
77
78  Comparation : process(X_IN, SUM)
79  begin
```

```
80          if (X_IN(30 downto 23) > SUM(30 downto 23)) then
81              BIG     <=  X_IN;
82              SMALL   <=  SUM;
83          else
84              BIG     <=  SUM;
85              SMALL   <=  X_IN;
86          end if;
87      end process;
88
89      --Preparation
90          dEXP        <=  BIG(30 downto 23) - SMALL(30 downto 23);
91          EXP1        <=  BIG(30 downto 23);
92          BIG_M       <=  BIG(22 downto 0);
93          SMALL_M     <=  SMALL(22 downto 0);
94
95
96      --Judge
97          EXP2        <=  EXP1;
98          BIG_Ms      <=  ("01" & BIG_M);
99          SMALL_Ms    <=  ('0'  & barrel_shift(('1'&SMALL_M), dEXP));
100
101
102     gen_SUM : process(RST, CLK)
103     variable SUM_E2     : std_logic_vector(7 downto 0) := "00000000";     --変更点
104     variable counter    : std_logic_vector(4 downto 0) := "00000";       --カウンター宣言
105     begin
106         if (RST = '0') then
107             SUM     <=  (23 => '1', others => '0');
108             EN_O    <=  '0';
109         elsif (CLK'event and CLK = '1') then
110             if (EN_IN = '1') then
111                 SUM_E2  :=  SUM_E - "00000001";               ----変更点(+ではなく-)
112                 counter :=  counter + "00001";               --ENが入ってからカウント
113                 if (counter = "10110") then                  --counterが任意の値の時だけ出力
114                     EN_O  <= '1';
115                 else
116                     EN_O  <= '0';
117                 end if;
118                 -------------------------------------------------
119                 if (SUM_M(24) = '1') then
120                     SUM     <=  '0' & SUM_E2 & SUM_M(23 downto 1);
121                 else
122                     SUM     <=  '0' & SUM_E2 & SUM_M(22 downto 0);
123                 end if;
124             end if;
125         end if;
126     end process;
127
128     SUM_M       <=  BIG_Ms + SMALL_Ms;
129     SUM_E       <=  EXP2 + SUM_M(24);
130     Y_OUT       <=  SUM;
131     EN_OUT      <=  EN_O;
132
133 end RTL;
```

Listing A.11 PointAdjustment のソースコード

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-----エンティティ宣言------

entity PointAdjustment is
    port(
        CLK                 :   in  std_logic;
        RST                 :   in  std_logic;
        EN_IN               :   in  std_logic;
        X_IN                :   in  std_logic_vector(31 downto 0);
        Y_OUT               :   out std_logic_vector(31 downto 0)
    );
end PointAdjustment;

-----アーキテクチャ定義-----

architecture RTL of PointAdjustment is
    signal  S   :   std_logic;
    signal  E   :   std_logic_vector( 7 downto 0);
    signal  M   :   std_logic_vector(22 downto 0);

    signal  Y   :   std_logic_vector(31 downto 0);

begin
    S   <=  X_IN(31);
    E   <=  X_IN(30 downto 23) + "00000101";
    M   <=  X_IN(22 downto  0);

    Output:process(RST, CLK)
    begin
        if (RST = '0') then
            Y   <= (others => '0');
        elsif (CLK'event and CLK = '1') then
            if (EN_IN = '1') then
                Y   <= S & E & M;
            end if;
        end if;
    end process;

    Y_OUT   <=  Y;

end RTL;
```