Approximate Computing を利用した配列型乗算器の遅延故障への対処法

渡邊 結希乃 †　山下 茂 ‡

1. はじめに

近年，演算結果の多少の誤差を許容することによって，演算時間の削減や消費電力の削減を目的とした“Approximate Computing”(以降，AC) と呼ばれる手法が注目されている [1]．ただし出力に多少の誤差が生じるため，正確さが必要とされるアプリケーションでは AC を用いることはできない．しかし，莫大な演算処理を含み演算処理のすべてを正確に行う必要のないアプリケーションでは，演算結果の正確さよりも，消費電力や演算時間，回路規模の観点で優れた AC を用いることが有効である．

加算器は，乗算や除算のような複雑な算術演算を実現するための重要な演算器である．そのため，加算器に対して AC を用いる研究が盛んに行われている [2–5]．また，これらの加算器を使って，AC を用いた乗算器の研究も行われている [6,7]．以降，AC を用いた加算器および乗算器をそれぞれ AC 加算器，AC 乗算器と呼ぶ．AC に関する研究の多くは，消費電力や回路規模削減に AC を利用 [8–10] しており，それ以外への AC の利用方法はほぼ考えられていない．

演算器は経年劣化により回路のどこかに不具合が生じる場合がある．不具合の一つに遅延故障がある．乗算器において遅延故障が発生した場合，上位ビットの計算が遅れる．それにより，上位ビットに誤差が発生する．

本研究では，AC の考え方を利用した，配列型乗算器に遅延故障が発生した際の対処法について提案する．提案手法では，遅延故障が発生した配列型乗算器の入力の下位ビットを削減する．そして，各ビットの値をそれぞれ削減した分だけ下位ビットにずらして入力し，削減した分だけ上位ビットの値を 0 に変更する．かつ，乗算器の出力の上位ビットを削減し，削減した分だけ下位に 0 を追加することで，遅延故障による誤差を削減する．

8 ビットの乗算に対して提案手法を用いた場合，平均誤差と最大誤差ともに遅延故障が発生した配列型乗算器を用いた場合の 100 分の 1 程度に誤差を抑えられることを示す．また，平滑化フィルタ処理を用いてソフトウェア検証を行った結果，PSNR の値の増減の割合は，入力を 1 ビットずつ削減した場合は平均 −1.10%，2 ビットずつ削減した場合は平均 −6.51% であった．さらに，上位ビットを確認する場合は 3 ビットずつ削減した場合でも PSNR の値の増減の割合は平均 −4.61% であった．

† 立命館大学大学院 情報理工学研究科
‡ 立命館大学 情報理工学部

本論文は 5 章で構成されている．以下，第 2 章で AC および乗算器における遅延故障の影響について述べる．第 3 章では，本論文の提案手法について述べる．第 4 章では提案手法の評価方法とその結果について述べる．最後に第 5 章でまとめと今後の課題を述べる．

2. 準備

本章では，2.1 節で AC について，2.2 節で遅延故障による影響についてそれぞれ説明する．

2.1 Aproximate Computing

Aproximate Computing とは演算結果の多少の誤差を許容することによって，演算時間の削減や消費電力の削減を目的とした手法である．動画像処理やデータマイニング，機械学習といったアプリケーションでは，演算において多少のエラーを許容できる場合がある．また，これらのアプリケーションでは莫大な演算処理を行うことが多いため，演算時間が長くなったり消費電力が大きくなったりしてしまう．そのため，AC を用いることが有効とされている．

AC を用いる場合，出力結果の精度と演算時間および回路規模はトレードオフの関係になる．そのため，AC を用いるアプリケーションに応じて，どの程度までの出力の誤差を許容するか検討する必要がある．

以下 2.1.1 節および 2.1.2 節にて多くの複雑な算術演算やアプリケーションにおいて必要となる加算器，乗算器に対する AC について説明する．

2.1.1 AC を用いた加算器

加算器は乗算や除算のような複雑な算術演算を実現する際に必要となる重要な演算器である．そのため，加算器に対して AC を用いる研究が盛んに行われている．

正確な結果が出力される全加算器を図 1 に，その全加算器の真理値表を表 1 に示す．また，AC を用いた全加算器を図 2 に，その全加算器の真理値表を表 2 に示す．以降，正確な結果が出力される全加算器を全加算器，AC を用いた全加算器を AC 全加算器と呼ぶ．

全加算器と AC 全加算器を比較すると，図 1 と図 2 からわかるように，AC 加算器の方が AND ゲートと XOR ゲートが一つずつ少ない．そのため，全加算器と比べて AC 全加算器の方が回路規模が小さく，演算時間も短くなる．ただし表 1 および表 2 からわかるように，AC

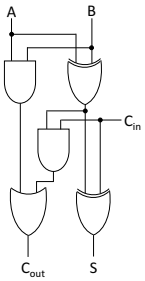


図 1：全加算器

表 1：図 1 の真理値表

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

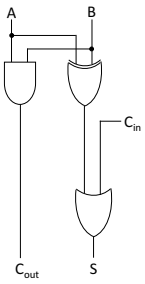


図 2：AC 全加算器

表 2：図 2 の真理値表

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

加算器で加算を行った場合に特定の入力で正確な結果を得ることができない．入力 (A, B, C_{in}) の組み合わせが $(0, 1, 1)$ または $(1, 0, 1)$ のとき，出力 (C_{out}, S) の組み合わせは全加算器では $(1, 0)$ となるところが，AC 全加算器では $(0, 1)$ となり誤った値を出力する．

2.1.2 AC を用いた乗算器

動画像処理やデータマイニング，機械学習といったアプリケーションにおいて，乗算は必要不可欠な演算である．そのため，乗算器に対して AC を用いる研究も行われている．

乗算の処理は大きく分けて次の二つの処理に分けることができる．

1. 部分積の生成
2. 部分積の加算

部分積の生成は乗算器の 2 つの入力のその部分積に対応するそれぞれのビットの AND を取ることで実現できる．例えば，図 4 の P00 を生成しようとするならば A0 と B0 の AND を取ればよく，P22 を生成するならば A2 と B2 の AND を取ればよい．そのため，部分積の生成の部分はすべて AND ゲートで構成される．

部分積の加算の一般的な回路を図 4 に示す．この回路は，生成された部分積を順次加算することで実現している．部分積の加算を行う回路は生成の部分と比べ，回路が大きく複雑になる．そのため，部分積の加算で使用される加算器に注目し，その加算器を AC 加算器に置き換えることで AC 乗算器を実現する手法 [8] などが考えられている．本論文は図 4 のような一般的な部分積の加算の回路で構成される 8 ビット配列型乗算器を想定している．

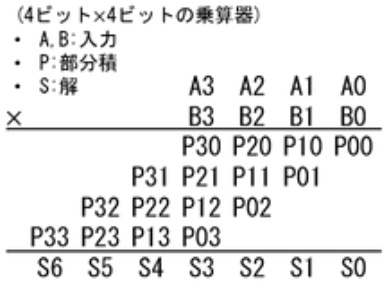


図 3：部分積の生成

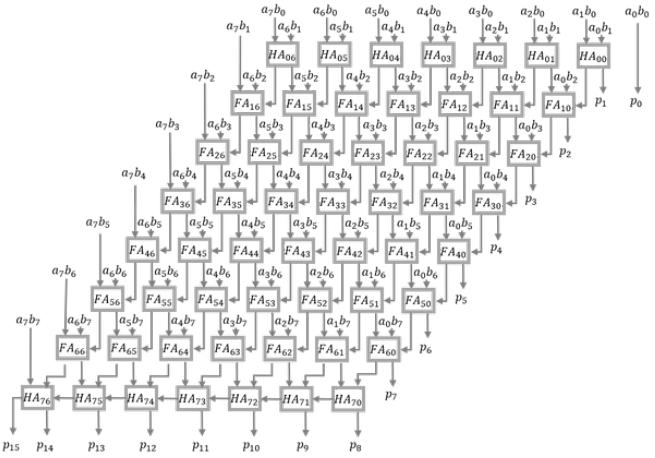


図 4：配列型乗算器

2.2 遅延故障による影響

乗算器において遅延故障が発生した場合，上位ビットの値の確定が遅れる．もし，図 4 における FA_{23} に遅延故障が発生した場合，FA_{23} の演算結果を必要とするビットすべての値の確定が遅れる．そのなかでも，上位ビットは入力によって値の確定がクロックサイクル内に終わらない場合がありえる．乗算器の最長パスの計算時間が全加算器 1 つ分増えた場合を 2 倍遅延，2 つ分増えた場合を 3 倍遅延が発生しているということにする．2 倍遅延が発生している場合だと p_{15}，p_{14} の値の確定がクロックサイクル内に終わらない場合がありえる．3 倍遅延の場合だと p_{13}，4 倍遅延の場合だと p_{12} より上位ビットの値の確定がクロックサイクル内に終わらない場合がありえる．p_{15} と p_{14} の値に誤差が発生した場合，演算結果の誤差が最大で $2^{15} + 2^{14}$ にも及ぶ．

3. 提案手法

本研究では，AC の考え方を利用した，配列型乗算器に遅延故障が発生した際の対処法について提案する．3.1 節で提案手法の概要，3.2 節で遅延故障に対処する手法について，3.3 節で上位ビットを確認し，より精度を上げる場合について説明する．

3.1 概要

2.2 節でも説明したとおり，2 倍遅延が発生している場合だと p_{15}，p_{14} の値の確定がクロックサイクル内に終わらない場合がありえる．3 倍遅延の場合だと p_{13}，4 倍遅延の場合だと p_{12} より上位ビットの値の確定がクロックサイクル内に終わらない場合がありえる．上位ビットの値の確定が遅れることにより演算がクロックサイクル内に終わらない場合，値が正しい値とならず誤差が発生することがある．また，上位ビットの値に誤差が発生するため，演算結果に対する誤差の割合が多くなり，問題になることが多いと考えられる．しかし，上位ビットの演算結果を利用しいのならば大きな誤差を防げる可能性がある．

そこで，入力の下位ビットを削減する．そして，各ビットの値をそれぞれ削減した分だけ下位ビットにずらして入力し，削減した分だけ上位ビットの値を 0 に変更する．かつ，乗算器の出力の上位ビットを削減し，削減した分だけ下位に 0 を追加する手法を提案する．そうすることにより，誤差のある上位ビットの演算結果を利用しないですむ．しかし，入力を下位ビットにずらすことは下位ビットの値を切り捨てることと同義のため，誤差が発生する場合がある．だが，この時発生する誤差は，上位ビットの値の確定がクロックサイクル内に終わらないことにより発生する誤差と比較して少なくなることが期待できる．

3.2 遅延故障に対処する手法

提案手法の具体的な手順を図 5 に示す．まずステップ 1 で入力の下位ビットを削減する．図 5 は，入力をそれぞれ 1 ビットずつ削減する場合の例である．その後，各ビットの値を下位ビットにずらして入力し，これにより空いた上位ビットには 0 を入力する（入力の削減するビット数をそれぞれ a,b とすると，入力をそれぞれ 2^{-a} 倍，2^{-b} 倍することを意味する）．次に，ステップ 2 で遅延故障が発生した配列型乗算器で演算する．そしてステップ 3 で，出力の最上位ビットから，ステップ 1 で削減したビット数分だけ値を 0 に変更する．最後にステップ 4 で，ステップ 1 で削減したビット数分だけ出力の下位ビットに 0 を追加する（出力を 2^{a+b} 倍することを意味する）．

図 5：提案手法の手順

ステップ 1 で入力の下位ビットを削減するが，このときに削減するビット量により何倍遅延まで対応できるかが決定する．これは，入力それぞれの下位ビットを削減したビット数の合計分までの遅延に対応できるためである（合計 2 ビット削減した場合 2 倍遅延，3 ビット削減した場合は 3 倍遅延まで対応できる）．このとき，2 入力それぞれ 1 ビットずつ削減する場合でも，片方の入力だけを 2 ビット削減する場合，どちらでも 2 倍遅延に対応できるようになる．しかし，削減する量を増やすほど大きな遅延に対応できるが，より多くの下位ビットを切り捨てることになるため，誤差が大きくなる．

3.3 上位ビットを確認する場合

演算の入力の値が小さいほど，値に対して下位ビットの値が占める割合が大きくなる．それにより提案手法を利用すると，入力の値が小さいほど，ステップ 1 で入力の下位ビットを削減することにより発生する誤差が大きくなる．また，削減するビット数を n としたときに入力の値が 2^n 以下の場合は，下位ビットを削減することにより入力の値が 0 となるため，誤差が最大になる．

値が小さいものは，上位ビットの値が 0 である．乗数の最上位ビットから n ビット 0 が続き，被乗数の最上位ビットから m ビット 0 が続く場合，演算結果の上位ビットには $n+m$ ビット分 0 が続く．例えば 8 ビット同士の乗算で乗数と被乗数がそれぞれ，00101101 と 01101101 の場合，演算結果は 0001001100101001 となり，乗数と被乗数それぞれの最上位ビットから連続して 0 がある分（例の場合は 3）だけ演算結果の最上位ビットに 0 が並んでいる．そのため，上位ビットに 0 が並ぶ入力の場合，出力の上位ビットの値を強制的に 0 に変更することで遅延故障による誤差を減らすことができる．

上位ビットを確認する場合の具体的な手順を図 6 に示す．まずステップ 1 で入力の下位ビットを削る．図 6 は，InputA の最上位ビットが 0 かつ，入力それぞれを 1 ビットずつ削減する場合の例である．下位ビットを削

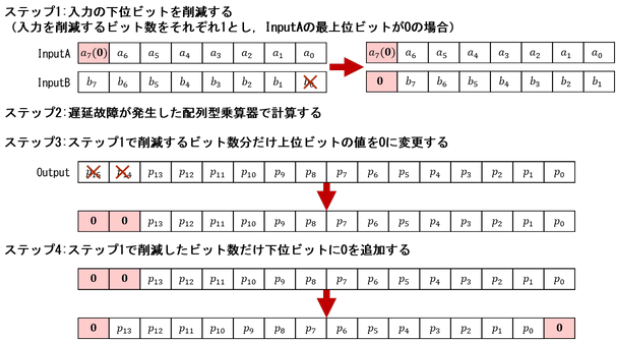


図 6：上位ビットを確認する場合の手順

り，各ビットの値を下位ビットにずらして入力し，これにより空いた上位ビットには 0 を入力する（入力の削減するビット数をそれぞれ a, b とすると，入力をそれぞれ 2^{-a} 倍，2^{-b} 倍することを意味する）．このとき，最上位ビットから連続して 0 がある場合は，0 の数だけ削減するビット数を減らす．図 6 の場合では，InputA の最上位ビットが 0 のため InputA は下位ビットを削減しない．次に，ステップ 2 で遅延故障が発生した配列型乗算器で演算する．そしてステップ 3 で，出力の最上位ビットから，ステップ 1 で削減するビット数だけ値を 0 に変更する．最後にステップ 4 で，実際に入力を削減したビット数だけ出力の下位ビットに 0 を追加する（出力を 2^{a+b} 倍することを意味する）．図 6 の場合では，実際に削減したビット数は InputB の 1 ビットだけのため，0 を 1 つ追加することになる．

4. 検証結果と考察

本章では 4.1 節で，提案手法を用いた場合に発生する演算結果の誤差と遅延故障が発生した配列型乗算器を用いた場合に発生する演算結果の誤差を比較する．4.2 節で，画像処理の 1 つである平滑化フィルタ処理によるソフトウェア検証を行い，提案手法により発生する誤差の影響を調べる．

4.1 誤差の比較

本節では，提案手法を用いた場合に発生する演算結果の誤差と遅延故障が発生した配列型乗算器を用いた場合に発生する演算結果の誤差を比較する．入力が 8 ビットの場合，乗数と被乗数ともに 10 進数で 0 から 255 の値となるため，入力の組み合わせは 65536 通りとなる．このすべての入力の組み合わせで，提案手法を用いた場合と遅延故障が発生した配列型乗算器を用いた場合の演算結果の誤差を算出した．また，このときの提案手法のステップ 1 の入力それぞれの削減ビット数を (InputA, InputB) と表すとすると，$(1,1)$，$(2,0)$，$(0,2)$ の場合で誤差の比較を行った．

表 3：演算結果の誤差

	(1, 1)	(2, 0)	(0, 2)	遅延
平均誤差	127.25	191.25	191.25	14992
最大誤差	509	765	765	49152

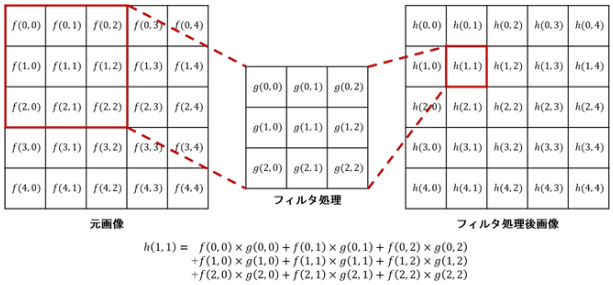


図 7：フィルタ処理

表 4.1 に提案手法を用いた場合と遅延故障が発生した配列型乗算器を用いた場合の演算結果の平均誤差と最大誤差を示す．表 4.1 より提案手法を用いた場合，平均誤差，最大誤差ともに遅延故障が発生した場合の誤差の 100 分の 1 程度に抑えることができることがわかる．

4.2 平滑化フィルタ処理による検証

本節では提案手法を用いた場合に発生する演算結果の誤差が，画像の平滑化フィルタ処理にどれだけ影響するかを検証し，その検証結果について考察する．

4.2.1 平滑化フィルタ処理

フィルタ処理とは，ある画像などに含まれる特定成分を取り除くことで，目的とする情報を抽出する処理のことをいう．例えば，ある画像に対してフィルタ処理をすることでより明暗のはっきりした画像などを得ることができる．フィルタ処理は図 7 に示すように行われる．

図 7 は画像が 5×5 ピクセル，フィルタが 3×3 ピクセルの時のフィルタ処理を表す．また $f(x,y)$ は，元画像の座標 (x,y) に位置するピクセルの輝度値を表している．例えば，$f(1,2)$ は，元画像の座標 $(1,2)$ に位置するピクセルの輝度値を表す．同様に $h(x,y)$ は，フィルタ処理後の画像の座標 (x,y) に位置するピクセルの輝度値を表している．また，$g(x,y)$ はフィルタの座標 (x,y) に位置する係数（以降，レート）のことを表す．

フィルタ処理は演算対象のピクセル（以下，注目ピクセル）の値とそのピクセルの周りにあるピクセルの値を用いて行われる．図 7 の例は，$(1,1)$ を注目ピクセルとした場合である．$(1,1)$ ピクセルのフィルタ処理を行う場合，$(0,0)$, $(0,1)$, $(0,2)$, $(1,0)$, $(1,1)$, $(1,2)$, $(2,0)$, $(2,1)$, $(2,2)$ の 9 ピクセルを用いる．そして，元画像とフィル

(3×3の場合)

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

(5×5の場合)

$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$
$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$	$\frac{1}{25}$

図 8：移動平均フィルタの例

(3×3の場合)

$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

(5×5の場合)

$\frac{1}{256}$	$\frac{4}{256}$	$\frac{6}{256}$	$\frac{4}{256}$	$\frac{1}{256}$
$\frac{4}{256}$	$\frac{16}{256}$	$\frac{24}{256}$	$\frac{16}{256}$	$\frac{4}{256}$
$\frac{6}{256}$	$\frac{24}{256}$	$\frac{36}{256}$	$\frac{24}{256}$	$\frac{6}{256}$
$\frac{4}{256}$	$\frac{16}{256}$	$\frac{24}{256}$	$\frac{16}{256}$	$\frac{4}{256}$
$\frac{1}{256}$	$\frac{4}{256}$	$\frac{6}{256}$	$\frac{4}{256}$	$\frac{1}{256}$

図 9：ガウシアンフィルタの例

タの対応する座標のピクセルの輝度値とレートを用いることで，フィルタ処理後における座標のピクセルの輝度値を演算する．例えば，$h(1,1)$ を求めるのであれば図 7 の下部に示す数式のとおりに演算する．そして，画像内のすべてのピクセルが注目ピクセルとして処理されると，フィルタ処理が完了したことになる．

次に，平滑化フィルタ処理について説明する．平滑化フィルタ処理とは，画像の輝度値を平らに滑らかにするための手法であり，画像中のノイズを除去するために用いられる．平滑化フィルタを用いてフィルタ処理を行うことで，より見やすい画像を得ることができる．

平滑化フィルタには移動平均フィルタとガウシアンフィルタがある．移動平均フィルタとガウシアンフィルタを図 8 と図 9 にそれぞれ示す．図 8 と図 9 では，それぞれのフィルタの例としてフィルタのサイズが 3×3 のものと 5×5 のものを示す．

移動平均フィルタは，注目ピクセルとその周辺のピクセルの輝度値を用いて，輝度値を平均し，処理後の画像の輝度値とする手法である．注目ピクセルとその周辺のピクセルの輝度値に図 8 に示すレートを掛け合わせて輝度値を演算する．図 8 のように，移動平均フィルタではフィルタ処理に関わる全てのピクセルに対して同じレートを掛け合わせる．

移動平均フィルタでは注目ピクセルとその周辺のピクセルの輝度値を単に平均しているが，一般的な画像では，注目ピクセルから離れるほど注目ピクセルの輝度値との差が大きくなる場合が多い．ガウシアンフィルタではそれを考慮し，注目ピクセルに近いほど平均値を演算するときの重みを大きくし，離れるほど重みが小さくなるようにガウス分布の関数 (式 1) を用いて演算している．そのため，ガウシアンフィルタは図 9 に示すようなフィルタとなる．また，移動平均フィルタとガウシアンフィルタともに，フィルタ内のすべてのレートを足し合わせると必ず 1 になる．

$$f(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \qquad (1)$$

4.2.2　平滑化フィルタ処理で行われる乗算の誤差

4.2.1 節で説明したとおり，平滑化フィルタ処理には移動平均フィルタとガウシアンフィルタがあるが，本検証ではガウシアンフィルタを用いた．以降，平滑化フィルタ処理とは，ガウシアンフィルタを用いるフィルタ処理を意味する．また，本検証では図 9 に示すフィルタのサイズが 5×5 のフィルタを平滑化フィルタとして用いる．

平滑化フィルタ処理では，加算及び乗算が繰り返し行われる．また，フィルタのレートが分数で表されるために除算も行われる．本検証では，乗算器に提案する遅延故障への対処法を適用した場合の評価が目的であるため，加算や除算には正確な結果が得られる演算器を用いた．輝度値は 2 進数で 8 ビットを用いて表現されるため，10 進数で 0 から 255 の値となる．そのため，フィルタのレートの分子を乗数，輝度値を被乗数とすると，乗数は 1, 4, 6, 16, 24, 36 の 6 通り，被乗数は 0 から 255 の 256 通りとなる．したがって，乗数と被乗数の組み合わせは，1536 $(= 6 \times 256)$ 通りとなる．この全ての入力の組み合わせで提案する遅延故障への対処法を用いて乗算を行った．なお，ステップ 1 にて下位ビットを削減するビット数は，乗数と被乗数ともに 1, 2, 3 ビットの場合をそれぞれ行った．また，3.3 節で説明している上位ビットを確認する場合の手法でも同様に，ステップ 1 にて下位ビットを削減するビット数を設定している．この手法の場合では上位ビットの値によって削減するビット数を減らしているが，入力の上位ビットに削減するビット数と同じだけ上位ビット 0 が並んでいた場合のみ削減するビット数を減らすようにして行った．

4.2.3　平滑化フィルタ処理による検証結果

本検証では，図 9 に示すフィルタのサイズが 5×5 のフィルタのガウシアンフィルタを用いて平滑化フィルタ

表 4 : PSNR 測定結果の比較 (1 ビット)

	乗算器	提案		提案（確認あり）	
	PSNR [dB]	PSNR [dB]	増減の割合 (%)	PSNR [dB]	増減の割合 (%)
Airplane	27.12	26.67	(-1.67)	27.08	(-0.17)
BARBARA	24.05	23.95	(-0.42)	24.04	(-0.06)
BORT	29.58	29.02	(-1.90)	29.50	(-0.25)
BRIDGE	23.33	23.23	(-0.46)	23.32	(-0.07)
Building	27.85	27.48	(-1.34)	27.80	(-0.18)
Cameraman	25.86	25.65	(-0.78)	25.83	(-0.12)
LAX	23.00	22.93	(-0.27)	22.99	(-0.04)
LENNA	29.09	28.63	(-1.59)	29.04	(-0.18)
Lighthouse	23.96	23.81	(-0.65)	23.94	(-0.09)
Text	25.35	25.15	(-0.77)	25.30	(-0.17)
WOMAN	29.24	28.73	(-1.76)	29.18	(-0.19)
girl	32.12	31.62	(-1.56)	32.08	(-0.11)
平均	26.71	26.41	(-1.10)	26.68	(-0.14)

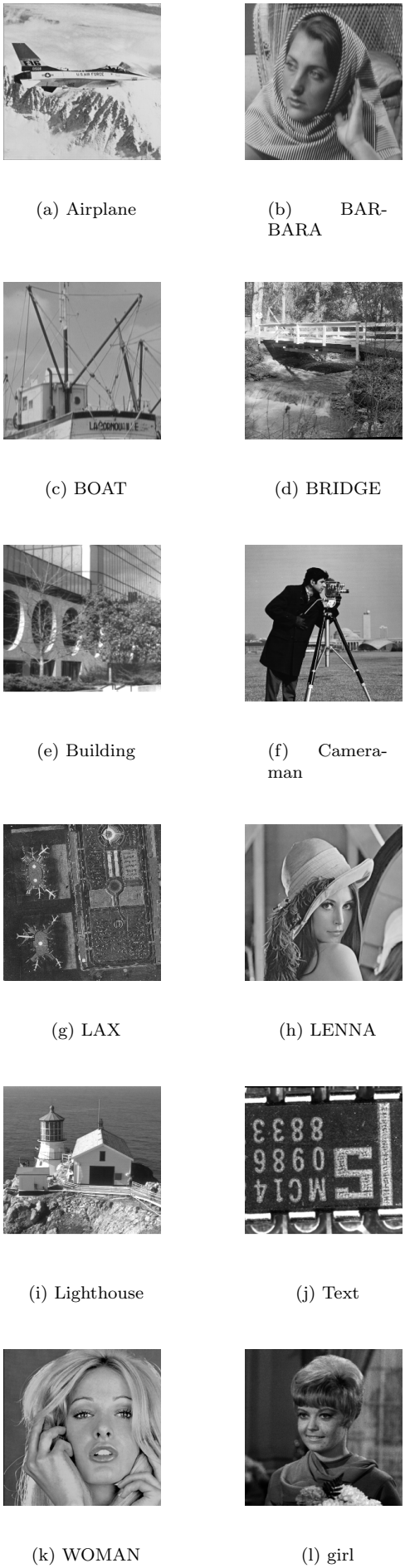
処理を行った．また．以下の手順で行う．

1. フィルタ処理を行うための画像 (以降，元画像) を用意する．
2. 元画像に対して適当なノイズを付加する．
3. 手順 2 にてノイズを付加した画像に対してフィルタ処理を行う．
4. 手順 3 にて生成された画像 (以降，フィルタ処理後の画像) と元画像を用いて評価を行う．

本検証では，図 10 に示す 12 枚の元画像を用いて平滑化フィルタ処理を行う．以降の説明で画像を使用する場合は，LENNA (図 10 (h)) の画像を例として用いる．図 10 に示す元画像に対して適当にノイズを付加し，平滑化フィルタ処理を行った後の画像を図 11 に示す．ただし，図 11 には，フィルタ処理を行う前のノイズを付加した画像 (図 11 (a)) も示す．

検証の評価には，Peak Signal to Noise Ratio (以降，PSNR) を用いる．PSNR は，画像が含むノイズの比率であり，元画像とフィルタ処理後の画像を比較することで計測する．PSNR の値が大きくなればなるほど，高画質であることを意味する．元画像とフィルタ処理後の画像を用いて検証の評価を行った結果 (以降，検証結果) を表 4，表 5 および表 6 に示す．表 4 は削減するビット数を 1 ビットとする場合の検証結果，表 5 は 2 ビットとする場合の検証結果，表 6 は 3 ビットとする場合の検証結果である．ただし，表 4，表 5 および表 6 内の値は小数第三位を四捨五入した値である．また表の括弧内の数字は，提案する遅延故障への対処法を用いた乗算器や，上位ビットを確認する場合の対処法を用いた乗算器の PSNR の値と正確な結果が得られる乗算器の PSNR の値との増減の割合を意味する．

提案手法の PSNR の値は，正確な乗算器の場合の PSNR の値と比較して，1 ビットずつ削減の場合は平均



(a) Airplane　(b) BARBARA

(c) BOAT　(d) BRIDGE

(e) Building　(f) Cameraman

(g) LAX　(h) LENNA

(i) Lighthouse　(j) Text

(k) WOMAN　(l) girl

図 10 : 元画像

(a) ノイズ付加　(b) 乗算器

(c) 提案 (1 ビット)　(d) 提案 (2 ビット)

(e) 提案 (3 ビット)　(f) 提案 (確認あり 1 ビット)

(g) 提案 (確認あり 2 ビット)　(h) 提案 (確認あり 3 ビット)

図 11：フィルタ処理後の画像

表 5：PSNR 測定結果の比較 (2 ビット)

	乗算器	提案		提案（確認あり）	
	PSNR [dB]	PSNR [dB]	増減の割合 (%)	PSNR [dB]	増減の割合 (%)
Airplane	27.12	24.44	(-9.91)	26.98	(-0.53)
BARBARA	24.05	23.25	(-3.36)	23.98	(-0.29)
BORT	29.58	26.44	(-10.62)	29.31	(-0.89)
BRIDGE	23.33	22.56	(-3.31)	23.27	(-0.28)
Building	27.85	25.63	(-7.98)	27.65	(-0.71)
Cameraman	25.86	24.48	(-5.33)	25.74	(-0.45)
LAX	23.00	22.55	(-1.93)	22.95	(-0.22)
LENNA	29.09	26.57	(-8.67)	28.86	(-0.80)
Lighthouse	23.96	22.87	(-4.54)	23.88	(-0.35)
Text	25.35	24.12	(-4.85)	25.21	(-0.54)
WOMAN	29.24	26.44	(-9.58)	29.01	(-0.79)
girl	32.12	29.52	(-8.10)	31.82	(-0.92)
平均	26.71	24.91	(-6.51)	26.56	(-0.56)

表 6：PSNR 測定結果の比較 (3 ビット)

	乗算器	提案		提案（確認あり）	
	PSNR [dB]	PSNR [dB]	増減の割合 (%)	PSNR [dB]	増減の割合 (%)
Airplane	27.12	14.25	(-47.45)	25.64	(-5.48)
BARBARA	24.05	16.90	(-29.76)	23.39	(-2.77)
BORT	29.58	16.10	(-45.57)	27.59	(-6.72)
BRIDGE	23.33	16.59	(-28.91)	22.72	(-2.61)
Building	27.85	16.44	(-40.99)	26.31	(-5.55)
Cameraman	25.86	16.61	(-35.75)	24.91	(-3.67)
LAX	23.00	18.49	(-19.58)	22.50	(-2.15)
LENNA	29.09	16.93	(-41.81)	27.38	(-5.90)
Lighthouse	23.96	15.80	(-34.05)	23.18	(-3.27)
Text	25.35	17.21	(-32.12)	24.34	(-3.97)
WOMAN	29.24	16.41	(-43.89)	27.41	(-6.28)
girl	32.12	20.42	(-36.43)	29.88	(-6.95)
平均	26.71	16.85	(-36.36)	25.44	(-4.61)

−1.10%，2 ビットずつ削減の場合は −6.51%，3 ビットずつ削減の場合は，−36.36% 減少した．3 ビットずつ削減の場合の誤差は多く，人間の目でも違いがわかるほどになっている．しかし，2 ビットずつ削減であれば，平均 −6.51% の誤差で抑えられ，4 倍遅延まで対処できる．

上位ビットを確認する場合の手法の PSNR の値は，正確な乗算器の場合の PSNR の値と比較して，1 ビットずつ削減の場合は平均 −0.14%，2 ビットずつ削減の場合は −0.56%，3 ビットずつ削減の場合は，−4.61% 減少した．上位ビットを確認する場合は，3 ビットずつ削減する場合でも，平均 −4.61% の誤差で抑えられており，有用であると考えられる．

5. おわりに

本研究では，AC の考え方を利用した，配列型乗算器に遅延故障が発生した際の対処法について提案した．提案手法では，遅延故障が発生した配列型乗算器の入力の下位ビットを削減する．そして，各ビットの値をそれぞれ削減した分だけ下位ビットにずらして入力し，削減した分だけ上位ビットの値を 0 に変更する．かつ，乗算器の出力の上位ビットを削減し，削減した分だけ下位に 0 を追加する．そうすることで，遅延故障が発生した配列

型乗算器を利用することで発生する誤差が，上位ビットではなく下位ビットになり誤差を削減できる．

8 ビットの乗算に対して提案手法を用いた場合，平均誤差と最大誤差ともに遅延故障が発生した配列型乗算器を用いた場合の 100 分の 1 程度に誤差を抑えられた．また，平滑化フィルタ処理を用いてソフトウェア検証を行った結果，PSNR の値の増減の割合は 2 倍遅延への対応で平均-1.10%，6 倍遅延への対応の場合も平均-6.51%であった．さらに，上位ビットを確認する場合は 6 倍遅延に対応をする場合でも，PSNR の値の増減の割合は平均-4.61%であった．

入力と出力を削減し，値をずらすことで遅延故障に対処しているため，実際の回路を想定すると値をずらす分の回路規模と演算時間が増えてしまうという問題がある．そのため，回路規模と演算時間について検討することが今後の課題である．

参考文献

[1] Jie Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*, pp. 1–6, May 2013.

[2] Ajay K. Verma, Philip Brisk, and Paolo Ienne. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pp. 1250–1255, New York, NY, USA, 2008. ACM.

[3] Ning Zhu, Wang Ling Goh, Weija Zhang, Kiat Seng Yeo, and Zhi Hui Kong. Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol. 18, No. 8, pp. 1225–1229, Aug 2010.

[4] Doochul Shin and S.K. Gupta. A re-design technique for datapath modules in error tolerant applications. In *Asian Test Symposium, 2008. ATS '08. 17th*, pp. 431–437, Nov 2008.

[5] Junjun Hu and Weikang Qian. A new approximate adder with low relative error and correct sign calculation. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pp. 1449–1454, San Jose, CA, USA, 2015. EDA Consortium.

[6] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. *Trans. Cir. Sys. Part I*, Vol. 57, No. 4, pp. 850–862, April 2010.

[7] Z. Babić, A. Avramović, and P. Bulić. An iterative logarithmic multiplier. *Microprocess. Microsyst.*, Vol. 35, No. 1, pp. 23–33, February 2011.

[8] Cong Liu, Jie Han, and Fabrizio Lombardi. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pp. 95:1–95:4, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[9] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pp. 346–351, Jan 2011.

[10] 後藤敏宏, 山下茂. Approximate computing を用いた乗算器の実装および検証 (コンピュータシステム) – (組込み技術とネットワークに関するワークショップ etnet2016). 電子情報通信学会技術研究報告 = IEICE technical report : 信学技報, Vol. 115, No. 518, pp. 199–204, mar 2016.