Vol.2016-ARC-219 No.34
Vol.2016-SLDM-175 No.34
Vol.2016-EMB-40 No.34
2016/3/25

Approximate Computing を用いた乗算器の実装および検証

後藤 敏宏[1,a)]　山下 茂[1]

概要：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，莫大な演算処理が行われるが，それらすべての演算処理を必ずしも正確に行う必要はない．そこで近年，計算結果の多少の誤差を許容することによって，計算時間の短縮や低面積化，低消費電力化を目的とした “Approximate Computing (AC)” と呼ばれる手法が注目されている．本研究では，乗算器に対する AC の適用を考えた．既存研究として，乗算器の構成要素である全加算器を，単純に一般的な AC 全加算器に置き換えることで AC 乗算器を実現させる研究が存在する．しかし，一般的な AC 全加算器は，乗算器のために設計された AC 全加算器ではないため，乗算器に対して有用な AC 全加算器であるとは限らない．そこで，提案手法では，乗算処理の一つである部分積の加算に注目し，その部分積の加算で使用される全加算器に適した AC 全加算器を提案した．提案手法の AC 全加算器を使用することで，既存研究よりも，低面積で高速に計算可能な AC 乗算器を実現することができることが確認できた．

1. はじめに

動画像処理やデータマイング，機械学習といったアプリケーションでは，計算において多少のエラーを許容できる場合がある．そのため近年，計算結果の多少の誤差を許容することによって消費電力の改善や計算時間の短縮，回路規模の縮小を目指す “Approximate Computing (以降，AC)” と呼ばれる手法が注目されている [3].

動画像処理やデータマイング，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ため，乗算器に対して AC を用いる研究も行われている [1], [9]. 以降，AC を用いた加算器を AC 加算器と呼ぶ.

AC 乗算器の研究には，様々な手法がある．その手法の一つが，乗算器の構成を変更することで，AC 乗算器を実現する手法である [5], [8]．この手法では，乗算器を構成する加算器などの回路のゲート数を削減するなどして，回路規模や計算時間が小さくなるように乗算器の構成を変更する．他にも，乗算器の入力となる乗数や被乗数から特定のビットを抜き取り，乗数や被乗数のビット数を減らすことで AC 乗算器を実現する手法 [4], [6] や，乗算器のハードウェア設計を変更するのではなく，VOS (Voltage Scaling) を用い，電圧を変化させることで AC 乗算器を実現する手法 [2], [7] が存在する.

本研究では，まず乗算の計算過程で行われる部分積の加算に適した AC 加算器を提案する．そして，その AC 加算器を用いることで，計算時間や回路規模などの観点から，既存研究の AC 乗算器よりも優れた AC 乗算器を実現する.

提案手法の AC 加算器を乗算器に適用すると，乗算結果に対する補正を行わない場合，既存研究よりも，乗算結果の精度が高くなるという結果が得られた．また，計算時間や回路規模も小さくなることが確認できた．さらに，画像の平滑化フィルタ処理を用いてソフトウェア検証を行い，提案手法の有用性を示した.

[1] 立命館大学
[a)] bowsan@ngc.is.ritsumei.ac.jp

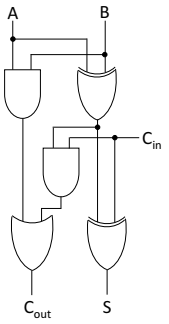

図 1　全加算器

図 2　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

2. AC を用いた加算器

正確な結果が出力される一般的な全加算器 (Full Adder: FA) を図 1 に，その全加算器の真理値表を表 2 にそれぞれ示す．次に，一般的な AC を用いた全加算器を図 3 に，その全加算器の真理値表を表 4 にそれぞれ示す．以降，正確な結果が出力される一般的な全加算器を全加算器，AC を用いた全加算器を AC 全加算器とそれぞれ呼ぶ.

全加算器 (図 1) と AC 全加算器 (図 3) を比較すると，AC 全加算器のほうが AND ゲートと XOR ゲートが一つ

Vol.2016-ARC-219 No.34
Vol.2016-SLDM-175 No.34
Vol.2016-EMB-40 No.34
2016/3/25

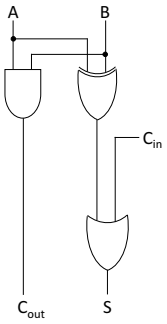

図 4 図 3 の真理値表

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

図 3 AC 全加算器

ずつ少ない．それにより，全加算器のクリティカルパスがゲート 3 段分であるのに対し，AC 全加算器のクリティカルパスはゲート 2 段分となり短くなる．したがって，全加算器と比較して AC 全加算器は回路規模は小さく，計算時間は短くなる．ただし，表 2 および表 4 からわかるように，AC 全加算器で加算を行った場合，特定の入力の組み合わせにおいて，正確な結果を得ることはできなくなってしまう．全加算器では，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，表 2 に示すように $(1, 0)$ となる．しかし，AC 全加算器では，同様の入力の組み合わせのとき，出力 (C_{out}, S) の組み合わせは表 4 に示すように $(0, 1)$ となり，誤った値を出力する．

3. AC 乗算器に関する既存研究

乗算器の構成を変更することで AC 乗算器を実現する手法の既存研究として，Liu らの手法が挙げられる [8]．本節では，Liu らの手法について説明する．

3.1 乗算器の構成

乗算処理は，“部分積の生成”と “部分積の加算” の二つに分けられる．部分積の生成は，乗算器における 2 入力の対応するビットの AND を取ることで実現できる．そのため，部分積を生成するための回路は AND ゲートのみで構成される．また，部分積を生成するための回路は AND ゲートのみの単純な回路であるため，Liu らの手法では部分積の生成に対して AC を適用していない．Liu らの手法では，部分積の加算に対して，AC が適用される．

3.2 AC 全加算器の乗算器への適用

Liu らの手法で使用される AC 全加算器を図 5 に，その全加算器の真理値表を表 6 にそれぞれ示す．以降，図 5 に示す Liu らの手法で使用される AC 全加算器を，“Liu らの AC 全加算器” と呼ぶ．

Liu らの AC 全加算器では，図 5 に示すように，A_{i-1} と B_{i-1} の AND を取ったもの $(A_{i-1} \cdot B_{i-1})$ を下位ビットからのキャリーとして扱う．E_i はエラー信号をあらわしており，$E_i = 1$ ならば S_i にエラーが発生していることを示す．ただし，Liu らの AC 全加算器では，エラーが発生した場合そのエラーを補正することを考慮しており，S_i と E_i を

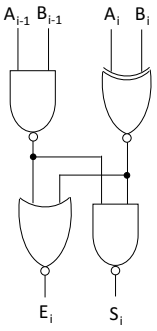

図 6 図 5 の真理値表

A_i	B_i	$A_{i-1} \cdot B_{i-1}$	S_i	E_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	0

図 5 AC 全加算器 (Liu らの手法)

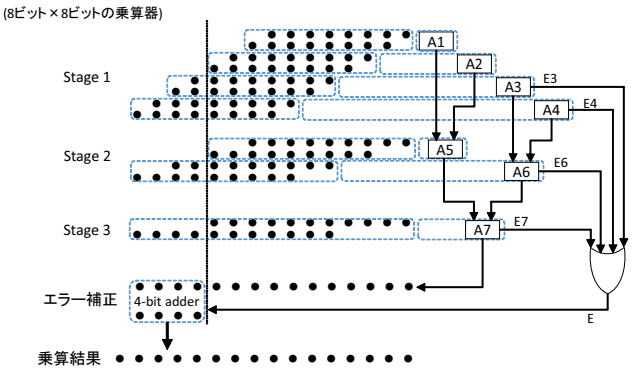

図 7 AC 乗算器の構造 (Liu らの手法)

加算することで正確な S_i を計算することができる．なお，S_i と E_i を加算をすることでキャリーが発生した場合，このエラー補正では，そのキャリーが正しく伝搬されるものとしている．

Lui らの AC 全加算器は, 前述したように $A_{i-1} \cdot B_{i-1}$ を下位ビットからのキャリーとして扱う．そのため, $A_{i-1} \cdot B_{i-1}$ は，2 節で紹介した一般的な AC 全加算器 (図 3) の C_{in} と同じであるといえる．したがって，Liu らの AC 全加算器の真理値表 (表 6) と一般的な AC 全加算器の真理値表 (表 4) を比較することでわかるように，エラー信号を生成すること以外は，Liu らの AC 全加算器と一般的な AC 全加算器はまったく同じ構造であるといえる．

次に，Liu らの AC 全加算器を使用した AC 乗算器の全体の構造を図 7 に示す．以降，Liu らの AC 全加算器を使用した AC 乗算器のことを，“Liu らの AC 乗算器” と呼ぶ．図 7 において，A1, A2, A3, A4, A5, A6, A7 は Liu らの AC 全加算器を使用した多ビットの AC 加算器を意味し，E3, E4, E6, E7 は対応する AC 加算器から出力されるエラー信号を意味する．図 7 に示す Liu らの AC 乗算器では，部分積の加算が Stage 1, Stage 2, Stage 3 の順に行われ，その後にエラー補正が行われる．エラー補正に使用される加算器には，AC 加算器ではなく，正確な値が出力される加算器が使用される．図 7 では，上位 4 ビットに対してエラー補正を行っている．

Liu らの AC 乗算器におけるエラー補正は，Liu らの AC 全加算器から出力されるエラー信号 (E_i) を使用して行う．

Vol.2016-ARC-219 No.34
Vol.2016-SLDM-175 No.34
Vol.2016-EMB-40 No.34
2016/3/25

表 1 Liu らの AC 乗算器の出力の誤差

補正ビット数	絶対差の平均	絶対差の最大値	相対差の平均	相対差の最大値	誤出力率
補正なし	3789.49	35024	18.48%	61.36%	82.64%
上位 1 ビット	2756.49	18704	16.40%	61.36%	82.41%
上位 2 ビット	1933.49	12960	13.60%	61.36%	82.03%
上位 3 ビット	1205.49	8864	10.32%	61.36%	81.06%
上位 4 ビット	777.49	8864	7.53%	60.69%	79.54%
上位 8 ビット	153.56	7072	1.16%	57.14%	51.42%
上位 12 ビット	127.50	6992	0.52%	44.44%	19.56%

上位４ビットのエラー補正を行うならば，上位４ビットの加算にかかわる Liu らの AC 全加算器のエラー信号を用いる．図 7 では， A3, A4, A6, A7 が最終的な乗算結果の上位４ビット (15 から 12 ビット目)，またはその一部のビットの加算を含んでいる．そのため，エラーの補正には A3, A4, A6, A7 のエラー信号 (E3, E4, E6, E7) の上位 4 ビットを用いる．そして，それらのエラー信号の対応するビットごとに OR を取ることで，補正するためのエラー信号を計算する．ただし，この補正するために計算したエラー信号は，単純に複数のエラー信号の OR を取っているだけである．そのため，同じビットに複数のエラーが発生した場合，その複数のエラーのうちの一つのエラーしか補正することができない．したがって，この計算したエラー信号を用いて，乗算結果に対して補正を行ってもすべてのエラーを補正できるわけではない．

次に，Liu らの AC 乗算器の出力の誤差について説明する．Liu らの AC 乗算器を用いて，8 ビット× 8 ビットの乗算を行った場合の出力の誤差について表 1 にまとめる．表 1 は，8 ビット× 8 ビットの乗算におけるすべての入力の組み合わせ ($2^8 \times 2^8 = 2^{16} = 65536$ 通り) に対する出力結果を用いて作成したものである．

表 1 に示すように，補正するビット数を増やせば出力の誤差は小さくなる．しかし，補正するビット数を増やした分だけ，計算時間は長く，回路規模は大きくなってしまう．計算時間や回路規模に関しては，5 節で述べる．

4. 提案手法

4.1 提案する AC 全加算器

本研究では，Liu らの手法と同様に，乗算における部分積の加算で使用される加算器に注目する．そして，本研究ではエラー補正を行わない場合に，Liu らの AC 乗算器よりも，乗算結果の誤差が小さくなるような AC 乗算器の実現を目指す．そのような AC 乗算器を実現するために，Liu らの AC 全加算器よりも乗算器に適した AC 全加算器を提案する．提案する AC 全加算器を図 8 に，その全加算器の真理値表を表 9 にそれぞれ示す．

図 8 に示すように，提案する AC 全加算器は，2 入力の OR をとったもの ($A_i + B_i$) を S_i とする．また，Liu らの AC 全加算器と同様に，E_i はエラー信号をあらわしており，$E_i = 1$ ならば S_i にエラーが発生していることを示す．また、提案する AC 乗算器におけるエラー補正は，Liu らの AC 全加算器のエラー補正と同様に行うものとする．

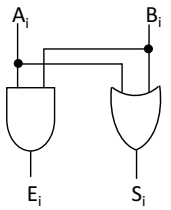

図 8 提案する AC 全加算器

図 9 図 8 の真理値表

A_i	B_i	S_i	E_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

表 2 Liu らの AC 全加算器および提案する AC 全加算器の真理値表

			Liu			提案		
A_i	B_i	$A_{i-1} \cdot B_{i-1}$	S_i	E_i	E_{i-1}	S_i	E_i	E_{i-1}
0	0	0	0	0	$*$	0	0	0
0	0	1	1	0	0	0	0	1
0	1	0	1	0	$*$	1	0	0
0	1	1	1	1	0	1	0	1
1	0	0	1	0	$*$	1	0	0
1	0	1	1	1	0	1	0	1
1	1	0	0	0	$*$	1	1	0
1	1	1	1	0	0	1	1	1

4.2 提案する AC 全加算器における出力の誤差

本節では，提案する AC 全加算器の出力の誤差と Liu らの AC 全加算器の出力の誤差を比較する．そのために，それぞれの全加算器の真理値表 (表 6 および表 9) を表 2 にまとめる．表 2 では，一桁下位のエラー信号 (E_{i-1}) についても示している．また，表 2 内の $*$ は，i ビット目と $i-1$ ビット目の情報だけでは，0 か 1 か決定できないことを意味し，$A_{i-2} \cdot B_{i-2} = 1$ かつ $A_{i-1} \bigoplus B_{i-1} = 1$ ならば 1 となり，そうでなければ 0 となる．

乗算における部分積の加算を行う場合，加算は 0 を補填して行われる．Stage 1 では上位 1 ビット (9 ビット目) および下位 1 ビット (1 ビット目)，Stage 2 では上位 2 ビット (10, 11 ビット目) および下位 2 ビット (1, 2 ビット目)，Stage 3 では上位 4 ビット (12 から 15 ビット目) および下位 4 ビット (1 から 4 ビット目) が 0 で補填され，部分積の加算が行われる．

したがって，部分積の加算では，0 が補填されたビットの入力 (A_i, B_i) は必ず $(0,0)$, $(0,1)$ または $(1,0)$ になる．そのため，提案する AC 全加算器では，0 が補填されたビットの E_i は必ず 0 となり，0 が補填されたビットではエラーは必ず発生しない．一方，Liu らの AC 全加算器では，0 が補填されたビットの入力の組み合わせ $(A_i, B_i, A_{i-1} \cdot B_{i-1})$ が $(0,1,1)$ または $(1,0,1)$ の場合，E_i が 1 となるため，0 が補填されたビットでエラーが発生することがある．したがって，0 が補填されたビットに関しては，提案する AC 全加算器のほうが Liu らの AC 全加算器よりも，出力に対するエラーが発生しにくいといえる．

Stage 1 で行われる部分積の加算は図 10 に示す 4 通りである．図 10 に示す 4 通りの Stage 1 で行われる部分積の加算では，入力の組み合わせ $(A_i, B_i, A_{i-1} \cdot B_{i-1})$ が $(0,0,1)$ になることはない．そのため，Liu らの AC 全加算器を使用することで発生する出力の誤差よりも，提案する AC 全加算器を使用することで発生する出力の誤差のほうが大きくなる入力の組み合わせ $(A_i, B_i, A_{i-1} \cdot B_{i-1})$ は，表 2

Vol.2016-ARC-219 No.34
Vol.2016-SLDM-175 No.34
Vol.2016-EMB-40 No.34
2016/3/25

図 10 Stage 1 で行われる部分積の加算の例

表 3 提案する AC 乗算器の出力の誤差

補正ビット数	絶対差の平均	絶対差の最大値	相対差の平均	相対差の最大値	誤出力率
補正なし	3166.49	32258	15.37%	49.61%	87.57%
上位 1 ビット	2654.49	24066	14.43%	49.41%	87.51%
上位 2 ビット	2014.49	19970	13.40%	49.22%	87.29%
上位 3 ビット	1478.49	17922	10.12%	48.82%	86.82%
上位 4 ビット	1102.49	16898	7.92%	48.43%	85.93%
上位 8 ビット	605.23	15938	2.97%	43.16%	68.63%
上位 12 ビット	581.73	15878	2.43%	24.42%	52.39%

より，$(1,1,0)$ および $(1,1,1)$ である．しかし，前述したように Stage 1 では最上位ビットが 0 で補填されるため，最上位ビットで入力の組み合わせ $(A_i, B_i, A_{i-1} \cdot B_{i-1})$ が $(1,1,0)$ および $(1,1,1)$ になることはない．したがって，最上位ビットに関しては，Liu らの AC 全加算器を使用することで発生する出力の誤差のほうが，提案する AC 全加算器を使用することで発生する出力の誤差よりも必ず大きくなる．以上のことから，Stage 1 で行われる部分積の加算では，提案する AC 全加算器を使用したほうが，Liu らの AC 全加算器を使用するよりも，出力の誤差が必ず小さくなるといえる．

Stage 2 および Stage 3 で行われる部分積の加算では，Stage 2 の上位 2 ビット，Stage 3 の上位 4 ビットが 0 で補填されるため，Stage 1 と同様の理由から，上位ビットでは提案する AC 全加算器を使用したほうが，Liu らの AC 全加算器を使用するよりも，出力の誤差は小さくなると考えられる．

4.3 提案する AC 全加算器の乗算器への適用

提案する AC 全加算器を使用した AC 乗算器の全体の構造は，Liu らの AC 乗算器の構造と同様で，図 7 に示すとおりである．また，エラーの補正の方法や考え方も Liu らの手法と同様である．提案する AC 全加算器を使用した AC 乗算器を用いて，8 ビット× 8 ビットの乗算を行った場合の出力の誤差について表 3 にまとめる．以降，提案する AC 全加算器を使用した AC 乗算器のことを “提案する AC 乗算器” と呼ぶ．表 3 は，8 ビット× 8 ビットの乗算におけるすべての入力の組み合わせ ($2^8 \times 2^8 = 2^{16} = 65536$ 通り) に対する出力結果を用いて作成したものである．

提案する AC 乗算器の出力の誤差 (表 3) と Liu らの AC 乗算器の出力の誤差 (表 1) を比較すると，補正なしの場合，提案する AC 乗算器のほうが誤差が小さいことがわかる．しかし，補正するビット数を増やしていくと，Liu らの AC 乗算器のほうが誤差が小さくなる．提案する AC 乗算器では，Liu らの AC 乗算器と同様のエラー補正を行っている．そのため，同じビットに複数のエラーが発生した場合，その複数のエラーのうちの一つのエラーしか補正することができない．前述したように，提案する AC 乗算器は Liu らの AC 乗算器よりも上位ビットでのエラーは発生しにくい．しかし，下位ビットでは，同一ビットで複数のエラーが発生することがある．したがって，提案する AC 乗算器では，Liu らの AC 乗算器と比較して，補正しきれないエラーが多いため，補正するビット数を増やすと，Liu らの AC 乗算器の出力の誤差のほうが小さくなったと考えられる．

表 4 回路面積および処理時間

	補正するビット数	回路面積 [μm^2]	処理時間 [ns]
乗算器	–	4844.85	8.01
提案する AC 乗算器	補正なし	883.81	1.06
	上位 4 ビット	1228.95	2.99
	上位 8 ビット	2038.58	5.26
	上位 12 ビット	2757.89	7.33
Liu らの AC 乗算器	補正なし	3119.16	3.31
	上位 4 ビット	3548.16	5.05
	上位 8 ビット	4386.82	7.78
	上位 12 ビット	5025.48	9.03

5. 検証結果と考察

5.1 Design Complier による検証

回路規模および計算時間を調べるために，VLSI Design and Education Center (VDEC) が提供する作業環境を利用して検証を行った．論理合成は Synopsys 社製の Design Compiler で行い，ROHM 0.18μm CMOS プロセスを利用した．また，ハードウェア記述言語には VerilogHDL を使用した．また，本節において，処理時間とは，回路における最悪実行時間のことを意味する．

8 ビット× 8 ビットの正確な結果が得られる乗算器，提案する AC 乗算器および Liu らの AC 乗算器それぞれの回路面積および処理時間を表 4 に示す．表 4 における “乗算器” とは，正確な結果が得られる乗算器のことを意味する．表 4 に示すように，補正するビット数を増やすほど，回路規模は大きくなり処理時間は長くなる．

前述したように，エラー補正を行わない場合，Liu らの AC 乗算器よりも提案する AC 乗算器のほうが出力の誤差が小さい．また，表 4 からわかるように，エラー補正を行わない場合，正確な結果が得られる乗算器と比較して，Liu らの AC 乗算器では回路規模が 35.6%，処理時間が 58.7%削減されたのに対し，提案する AC 乗算器では回路規模が 81.8%，処理時間が 86.8%削減された．したがって，エラー補正を行わない場合は，精度 (出力の誤差) だけでなく，回路規模や処理時間でも提案する AC 乗算器のほうが優れているといえる．

5.2 平滑化フィルタ処理による検証

本節では，提案する AC 乗算器や Liu らの AC 乗算器を特定のアプリケーションで使用した場合に，それらの AC 乗算器を使用することで発生する出力の誤差が，そのアプ

リケーションの最終的な結果にどれだけ影響するかについて述べる．本研究では，画像処理の一つである画像の平滑化フィルタ処理を用い，ソフトウェア検証を行った．

本検証は以下の手順で行う．

(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 (以降，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，12 枚の元画像に対して平滑化フィルタ処理を行う．平滑化フィルタ処理は，正確な結果が得られる乗算器，提案する AC 乗算器，Liu らの AC 乗算器でそれぞれ行う．ただし，AC 乗算器に関しては，エラー補正なし，上位 4 ビットを補正，上位 8 ビットを補正，の 3 通りでそれぞれ行う．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 (以降，フィルタ処理後の画像) を図 11 に示す．ただし，図 11 には，比較のために元画像 (図 11 (a)) も示す．また，図 11 において，“提案” とは提案する AC 乗算器を意味し，“Liu” とは Liu らの AC 乗算器を意味する．

検証の評価には，Peak Signal to Noise Ratio (PSNR) を利用する．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し計測する．PSNR の数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 5, 6 および 7 に示す．表 5 は，AC 乗算器に対してエラー補正を行わない場合 (図 11 (a) と図 11 (b), 11 (c) および 11 (d) の比較) の検証結果である．表 6 は，AC 乗算器に対して上位 4 ビットのエラー補正を行った場合 (図 11 (a) と図 11 (b), 11 (e) および 11 (f) の比較) の，表 7 は，AC 乗算器に対して上位 8 ビットのエラー補正を行った場合 (図 11 (a) と図 11 (b), 11 (g) および 11 (h) の比較) の検証結果である．また，表 5, 6 および 7 内の括弧内の数字は，提案する AC 乗算器や Liu らの AC 乗算器の PSNR の値と正確な結果が得られる乗算器の PSNR の値との差 (増減の割合) を意味する．

PSNR を利用した検証結果から，以下のことがいえる．

- エラー補正を行わない場合，どちらの AC 乗算器を用いても，正確な結果が得られる乗算器との PNSR の値の差は大きい．
- 上位 4 ビットをエラー補正する場合，12 枚の画像の平均では，提案する AC 乗算器を用いると，正確な結果が得られる乗算器の PNSR の値と同程度となる．
- 上位 8 ビットをエラー補正する場合，12 枚のどの画像について，どちらの AC 乗算器を用いても，正確な結果が得られる乗算器の PNSR の値と同程度の (わずかに高い) 値になる．

したがって，平滑化フィルタ処理をする場合，上位 4 ビット以上に対してエラー補正を行えば，提案する AC 乗算器は PSNR の値では正確な結果が得られる乗算器とそれほど変わらないといえる．

(a) 元画像　(b) 乗算器
(c) 提案 (補正なし)　(d) Liu (補正なし)
(e) 提案 (上位 4 ビット)　(f) Liu (上位 4 ビット)
(g) 提案 (上位 8 ビット)　(h) Liu (上位 8 ビット)

図 11　フィルタ処理後の画像 (LENNA)

表 5　PSNR 測定結果の比較 (補正なし)

	乗算器	提案する AC 乗算器		Liu らの AC 乗算器	
	PSNR	PSNR	増減の割合	PSNR	増減の割合
画像名	[dB]	[db]	(%)	[dB]	(%)
Airplane	26.40	22.14	(-16.12)	17.18	(-32.40)
BARBARA	23.68	23.39	(-1.19)	22.31	(-5.78)
BOAT	28.34	28.47	(+0.46)	26.62	(-6.09)
BRIDGE	23.01	22.09	(-4.02)	21.02	(-8.69)
Building	27.03	23.25	(-13.97)	22.20	(-17.88)
Cameraman	25.30	24.88	(-1.66)	23.96	(-5.31)
LAX	22.70	22.64	(-0.25)	22.24	(-2.04)
LENNA	27.99	26.67	(-4.77)	23.78	(-15.05)
Lighthouse	23.60	20.98	(-11.11)	19.99	(-15.32)
Text	24.85	21.65	(-12.86)	21.12	(-15.01)
WOMAN	28.12	27.70	(-1.48)	25.16	(-10.53)
girl	30.13	29.73	(-1.30)	27.62	(-8.32)
平均	25.93	24.47	(-5.69)	22.82	(-11.87)

また，平滑化フィルタ処理において，提案する AC 乗算器は，上位 12 ビットに対してエラー補正を行った場合，正確な結果が得られる乗算器とまったく同じ乗算結果を出力

表 6 PSNR 測定結果の比較 (上位 4 ビット補正)

画像名	乗算器 PSNR [dB]	提案する AC 乗算器 PSNR [db]	提案する AC 乗算器 増減の割合 (%)	Liu らの AC 乗算器 PSNR [dB]	Liu らの AC 乗算器 増減の割合 (%)
Airplane	26.40	26.43	(+0.11)	25.98	(-1.57)
BARBARA	23.68	23.76	(+0.35)	23.36	(-1.35)
BOAT	28.34	29.16	(+2.87)	28.65	(+1.08)
BRIDGE	23.01	22.85	(-0.72)	22.72	(-1.27)
Building	27.03	25.31	(-6.35)	25.98	(-3.86)
Cameraman	25.30	25.31	(+0.03)	25.01	(-1.14)
LAX	22.70	22.74	(+0.18)	22.49	(-0.95)
LENNA	27.99	28.10	(+0.39)	27.01	(-3.50)
Lighthouse	23.60	22.39	(-5.12)	22.53	(-4.53)
Text	24.85	23.41	(-5.77)	24.52	(-1.31)
WOMAN	28.12	28.58	(+1.62)	27.59	(-1.87)
girl	30.13	30.30	(+0.58)	28.75	(-4.56)
平均	25.93	25.69	(-0.99)	25.38	(-2.07)

表 7 PSNR 測定結果の比較 (上位 8 ビット補正)

画像名	乗算器 PSNR [dB]	提案する AC 乗算器 PSNR [db]	提案する AC 乗算器 増減の割合 (%)	Liu らの AC 乗算器 PSNR [dB]	Liu らの AC 乗算器 増減の割合 (%)
Airplane	26.40	26.51	(+0.44)	26.53	(+0.49)
BARBARA	23.68	23.74	(+0.25)	23.74	(+0.25)
BOAT	28.34	28.54	(+0.69)	28.56	(+0.76)
BRIDGE	23.01	23.08	(+0.29)	23.08	(+0.30)
Building	27.03	27.13	(+0.38)	27.15	(+0.46)
Cameraman	25.30	25.39	(+0.34)	25.39	(+0.35)
LAX	22.70	22.75	(+0.22)	22.74	(+0.19)
LENNA	27.99	28.16	(+0.60)	28.16	(+0.60)
Lighthouse	23.60	23.64	(+0.17)	23.66	(+0.23)
Text	24.85	24.89	(+0.17)	24.92	(+0.27)
WOMAN	28.12	28.30	(+0.63)	28.30	(+0.64)
girl	30.13	30.40	(+0.91)	30.40	(+0.91)
平均	25.93	26.04	(+0.42)	26.05	(+0.45)

する．そのため，平滑化フィルタ処理において，正確な結果が得られる乗算器を使用した場合とまったく同じフィルタ処理後の画像が必要であれば，上位 12 ビットを補正した提案する AC 乗算器を使用すればよい．

また，提案する AC 乗算器だけでなく，Liu らの AC 乗算器でも，同様のことがいえる．しかし，表 4 に示すように，同じビット数のエラー補正を行う場合，提案する AC 乗算器のほうが Liu らの AC 乗算器よりも，回路規模が小さく，計算時間が短くなる．そのため，平滑化フィルタ処理においては，Liu らの AC 乗算器より提案する AC 乗算器のほうが優れていると考えられる．

6. おわりに

本研究では，AC 乗算器を実装するにあたって，乗算で行われる部分積の加算に注目し，その部分積の加算に適した AC 全加算器を提案した．そして，提案手法の AC 全加算器を使用した AC 乗算器と正確な結果が得られる一般的な乗算器の回路規模および計算時間を調べ，それらを比較した．

その結果，AC 全加算器を用いることで発生する出力の誤差に対する補正を行わない場合，提案手法の AC 乗算器は，一般的な乗算器と比較して，回路規模が 81.8%，計算時間が 86.6%削減できることがわかった．また，提案手法の AC 乗算器が，既存研究として紹介した Liu らの AC 乗算器よりも回路規模，計算時間の観点から，優れた AC 乗算器であることを示した．そして，平滑化フィルタ処理を用いてソフトウェア検証を行い，提案手法の AC 乗算器の有用性を示した．

本研究では，乗算器の構成を変更することで AC 乗算器を実現する手法を用いた．しかし，他にも AC 乗算器を実現する手法が存在する．そのような手法では，一般的な乗算器を使用することが多いが，その代わりに提案手法の AC 乗算器を用いることで，回路規模や計算時間，消費電力の観点から，より優れた AC 乗算器が実現できる可能性があると考えられる．

謝辞 本研究に関して，貴重な助言，ご指摘をしていただきました東京工業大学大学院理工学研究科通信情報工学専攻 原祐子准教授，立命館大学理工学部電子情報工学科 冨山宏之教授に深く感謝いたします．本研究は JSPS 科研費 15H02679 の助成を受けたものです．

参考文献

[1] Babi, Z., Avramovi, A. and Buli, P.: An iterative logarithmic multiplier, *Microprocessors and Microsystems*, Vol. 35, No. 1, pp. 23 – 33 (2011).
[2] George, J., Marr, B., Akgul, B. E. S. and Palem, K. V.: Probabilistic Arithmetic and Energy Efficient Embedded Signal Processing, *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, New York, NY, USA, ACM, pp. 158–168 (2006).
[3] Han, J. and Orshansky, M.: Approximate computing: An emerging paradigm for energy-efficient design, *Test Symposium (ETS), 2013 18th IEEE European*, pp. 1–6 (2013).
[4] Hashemi, S., Bahar, R. I. and Reda, S.: DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, Piscataway, NJ, USA, IEEE Press, pp. 418–425 (2015).
[5] Kulkarni, P., Gupta, P. and Ercegovac, M.: Trading Accuracy for Power with an Underdesigned Multiplier Architecture, *VLSI Design (VLSI Design), 2011 24th International Conference on*, pp. 346–351 (2011).
[6] Kyaw, K. Y., Goh, W. L. and Yeo, K. S.: Low-power high-speed multiplier for error-tolerant application.
[7] Lau, M. S., Ling, K.-V. and Chu, Y.-C.: Energy-aware Probabilistic Multiplier: Design and Analysis, *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, New York, NY, USA, ACM, pp. 281–290 (2009).
[8] Liu, C., Han, J. and Lombardi, F.: A Low-power, High-performance Approximate Multiplier with Configurable Partial Error Recovery, *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, 3001 Leuven, Belgium, Belgium, European Design and Automation Association, pp. 95:1–95:4 (2014).
[9] Mahdiani, H., Ahmadi, A., Fakhraie, S. and Lucas, C.: Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications, *Circuits and Systems I: Regular Papers, IEEE Transactions on*, Vol. 57, No. 4, pp. 850–862 (2010).