2乗則を用いたデジタル乗算器アルゴリズムとFPGA実装の検討

群馬大学大学院　理工学府
電子情報部門
佐々木秀　小林春夫

OUTLINE

- 研究背景
- 検討乗算アルゴリズム
- 検討アルゴリズム乗算回路の設計とシミュレーション検証
- 各方式の比較
- 今後の検討回路
- まとめ

OUTLINE

- 研究背景
- 検討乗算アルゴリズム
- 検討アルゴリズム乗算回路の設計とシミュレーション検証
- 各方式の比較
- 今後の検討回路
- まとめ

研究背景

演算器
- ・加算器　・減算器
- ・乗算器　・除算器

デジタル計算機、DSPチップ、通信用SoCに広く使用

乗算器は加算器、減算器よりも頻繁に使う

複雑な計算を要するチップ内に複数乗算器が必要

乗算器は回路規模が大きいため低減が必要
小規模回路、低消費電力、高速化
のため様々な提案・実現されてきた

デジタル加算の仕組み

小学校で習った方式（筆算）を2進数に変換する

10進数での演算

25	被加数
＋39	加数
64	結果

2進数での演算

011001	: 25_{10}	被加数
＋100111	: 39_{10}	加数
1000000	: 64_{10}	結果

デジタル乗算の仕組み

小学校で習った方式 → 乗数に応じた部分積を求め、それらを足し合わせて積を得る

10進数での演算

25	被乗数
×39	乗数
45	部分積
18	
15	
＋ 6	
975	積

2進数での演算をベース

011001 : 25_{10}	被乗数
×100111 : 39_{10}	乗数
011001	部分積
011001	
011001	
000000	
000000	
＋011001	
001111001111 : 975_{10}	積

部分積の和の
計算回数が多くなる

全加算器の二次元配列を用いた乗算器

```
        0 1 1 0 0 1 : 25₁₀   被乗数
      × 1 0 0 1 1 1 : 39₁₀   乗数
```

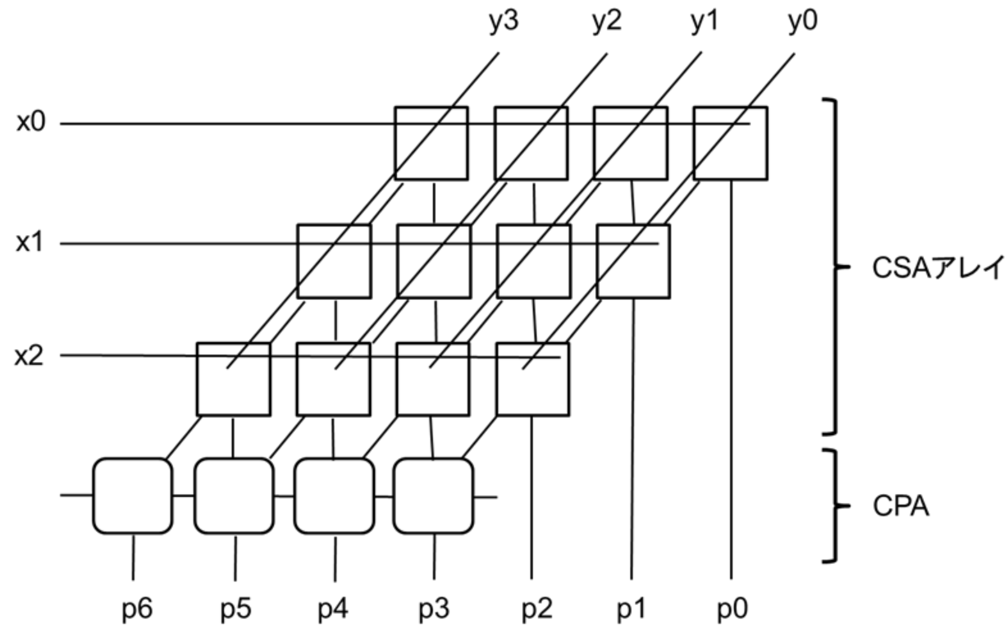
$011001:25_{10}$ 被乗数

$\times 100111:39_{10}$ 乗数

部分積:

$$
\begin{array}{r}
011001 \\
011001 \\
011001 \\
000000 \\
000000 \\
+011001 \\
\hline
001111001111
\end{array}
$$

積 $001111001111:975_{10}$



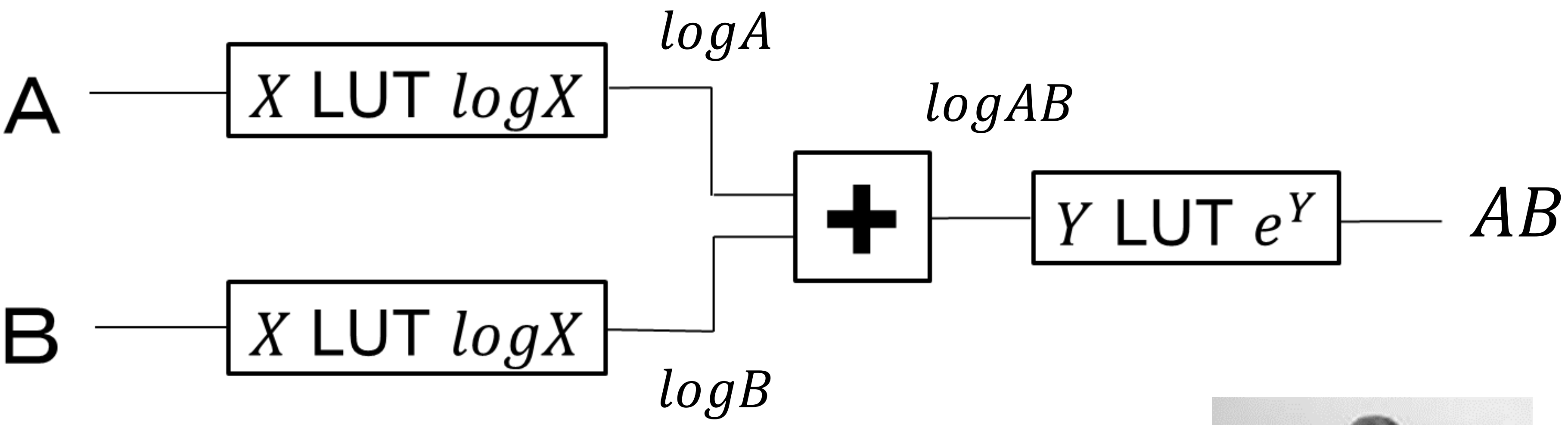
部分積＆加算のため全加算器の２次元配列

例: 8bit × 8bit の場合
直接的構成では8×8=64個の全加算器が必要

回路規模・消費電力・演算時間が大

対数を用いた乗算器

乗算を、対数を用いて計算する方式



高精度で対数、指数を得るためには
ビット数
LUTサイズ → 大

対数の生みの親
ジョン・ネイピア

研究の目的

従来方式（加算器の二次元配列）
・回路規模
・消費電力
・演算時間
➡ 大

従来方式（対数）
・ビット数
・LUTサイズ
大

加算器の二次元配列、対数を使わない
デジタル乗算回路の設計を行う

回路規模・消費電力・演算時間の縮小

OUTLINE

- 研究背景
- 検討乗算アルゴリズム
- 検討アルゴリズム乗算回路の設計とシミュレーション検証
- 各方式の比較
- 今後の検討回路
- まとめ

検討する乗算アルゴリズム

2乗則に基づく乗算アルゴリズム

$$AB = \frac{1}{2}\left[(A+B)^2 - A^2 - B^2\right]$$

乗算を加算、減算、2乗、ビットシフトで実現

- 加算1回、減算2回
- 2乗はルックアップテーブル(LUT)で実現
- 1/2倍はシフト演算(実際は配線変更のみ)

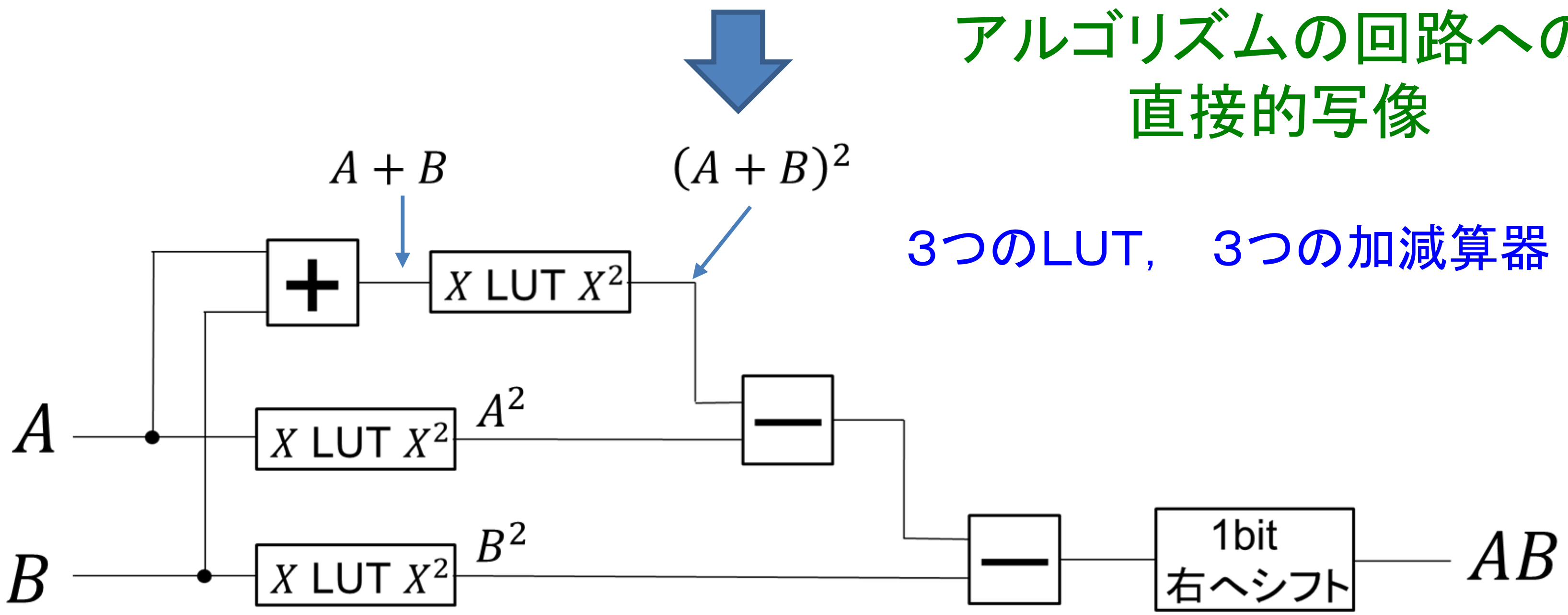
直接的構成		検討する構成
乗算器1つ(24bit) 加算回数：23回	→	加算回数：1回 減算回数：2回 LUT参照：3回

検討アルゴリズムの実現回路

$$AB = \frac{1}{2}[(A+B)^2 - A^2 - B^2]$$

アルゴリズムの回路への
直接的写像

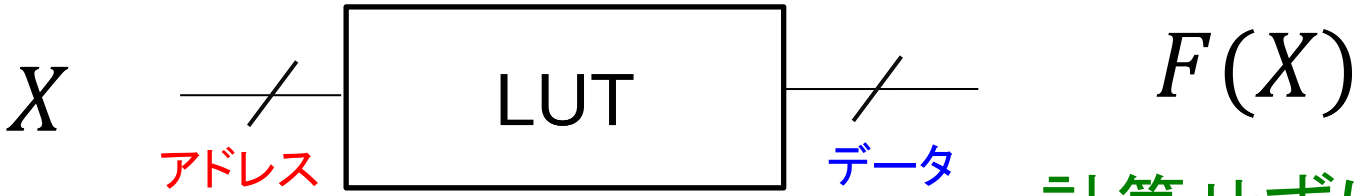
3つのLUT，　3つの加減算器



ルックアップテーブル (LUT) とは

複雑な信号処理 ⇒ 計算時間がかかる

LUTを使うと・・・・・



計算せずに出力

メモリ (ROM, RAM)

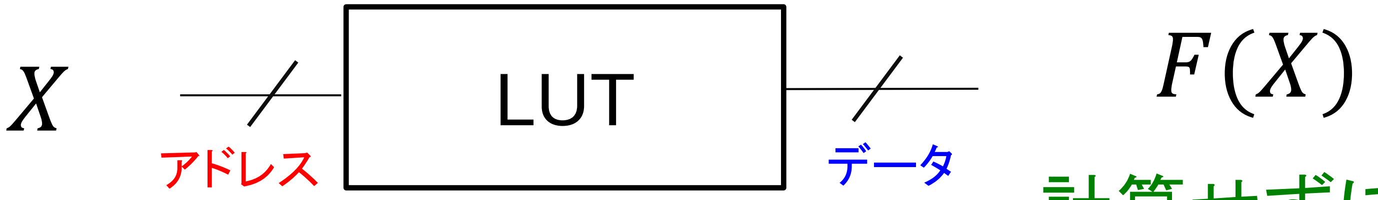
アドレス	メモリ
1	1
2	4
3	9
⋮	⋮
100	10000

データ
1
4
9
⋮
10000

ルックアップテーブル (LUT) とは

複雑な信号処理 → 計算時間がかかる

LUTを使うと・・・・・



計算せずに出力

メモリ (ROM, RAM)

<u>メモリの参照処理になり、効率化</u>

※Sin, Cos, 2乗等任意演算がLUTで計算可能

LUTの利点、欠点

利点

・簡単に２乗演算計算
・メモリの参照処理のみ
・回路設計が容易
・計算速度の向上

欠点

・演算ビット数が多い

↓

回路規模が大

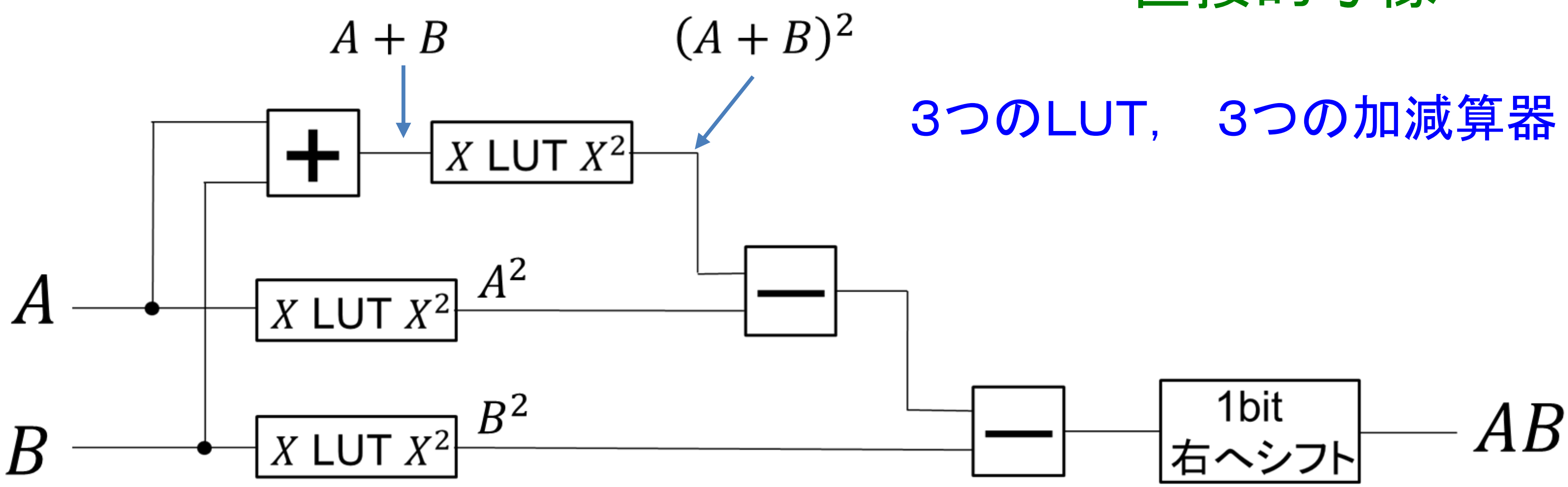
LUTを上位ビット、下位ビットに分割し
計算量･回路量を低減する方式
(Divide & Conquer)
を検討

実現回路の改良案

$$AB = \frac{1}{2}[(A+B)^2 - A^2 - B^2]$$

アルゴリズムの回路への
直接的写像

3つのLUT，　3つの加減算器



※演算するbit数によってLUTのサイズが大きくなってしまう

計算量・回路量を低減する方式(Divide & Conquer) を用いる

Divide & Conquer によるLUT規模削減

$$AB = \frac{1}{2}[(A+B)^2 - A^2 - B^2]$$

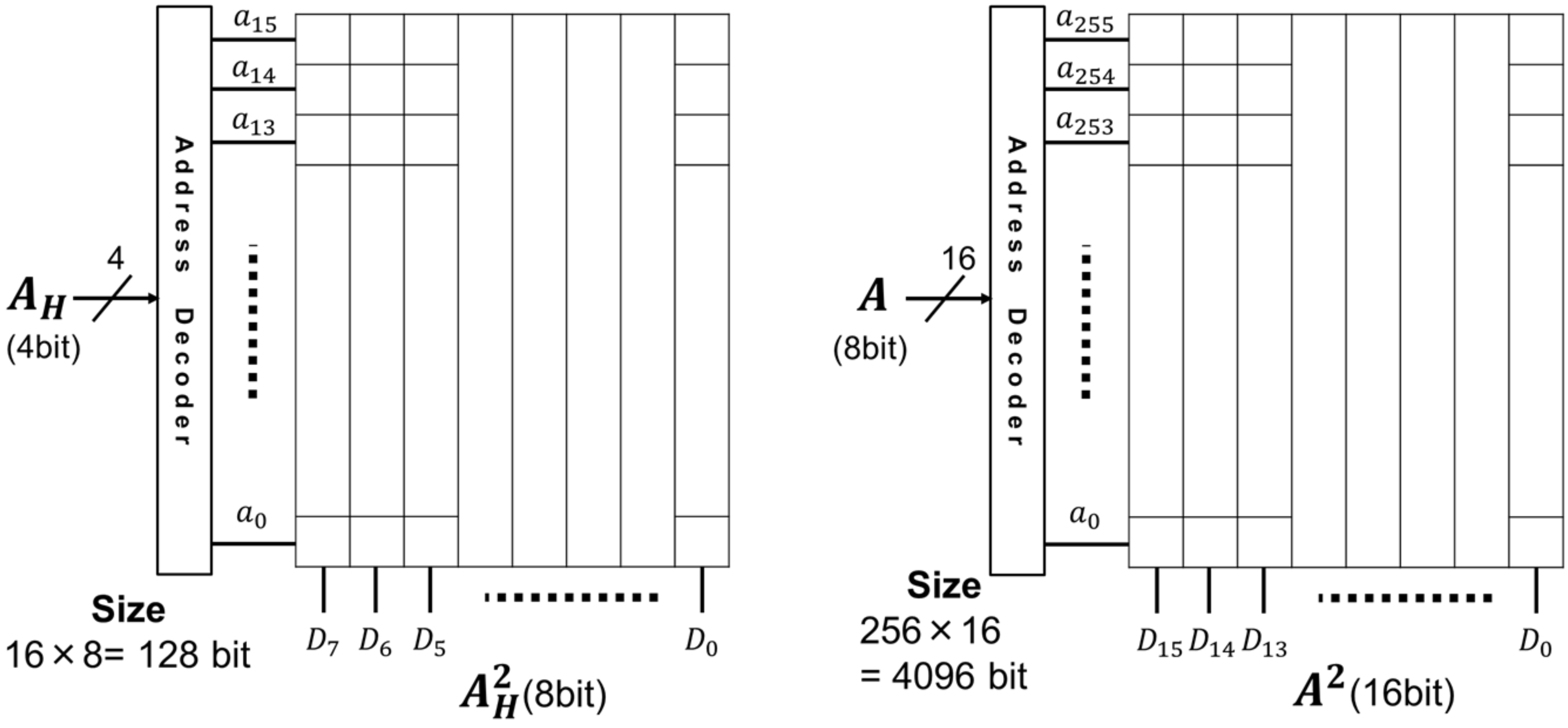
２乗演算：LUTで計算

LUTサイズ低減のため
A, B, A+B のそれぞれの上位ビット、下位ビットに分割し
計算量低減する方式

古代ローマ帝国：
支配下の都市相互の連帯を禁じ、
都市毎に処遇に格差をつける
分割統治(divide & conquer)で
征服した都市の反乱を抑える。

LUTで扱うビット数

LUTは扱うbit数が大きいほど、必要なbit数が多くなる。

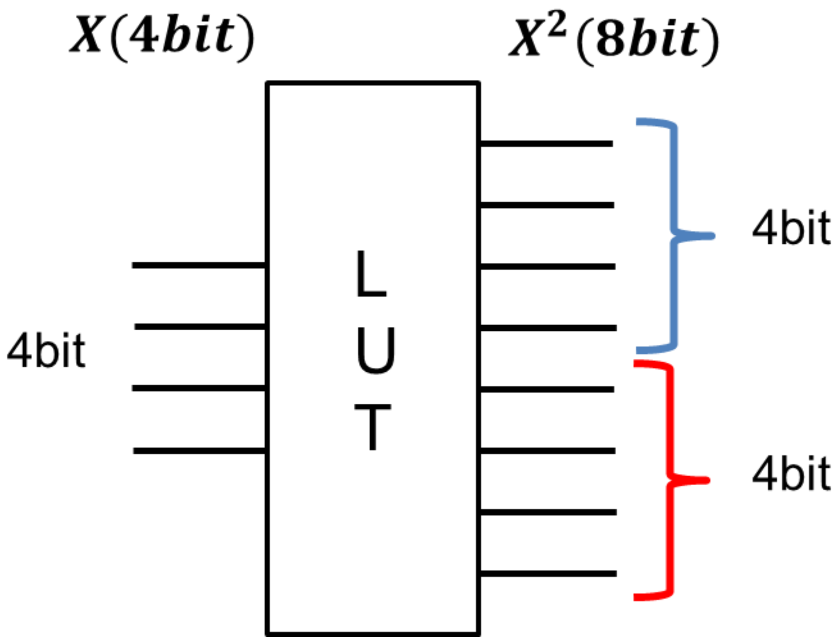


4bitでは128bit　　　8bitでは4096bit

Divide&Conquer法を用いることで必要なbit数を削減

検討するDivide & Conquer法

出力8bitの場合: 上下4bitずつ分割



bit数を減らして計算

シフト演算で桁を揃える

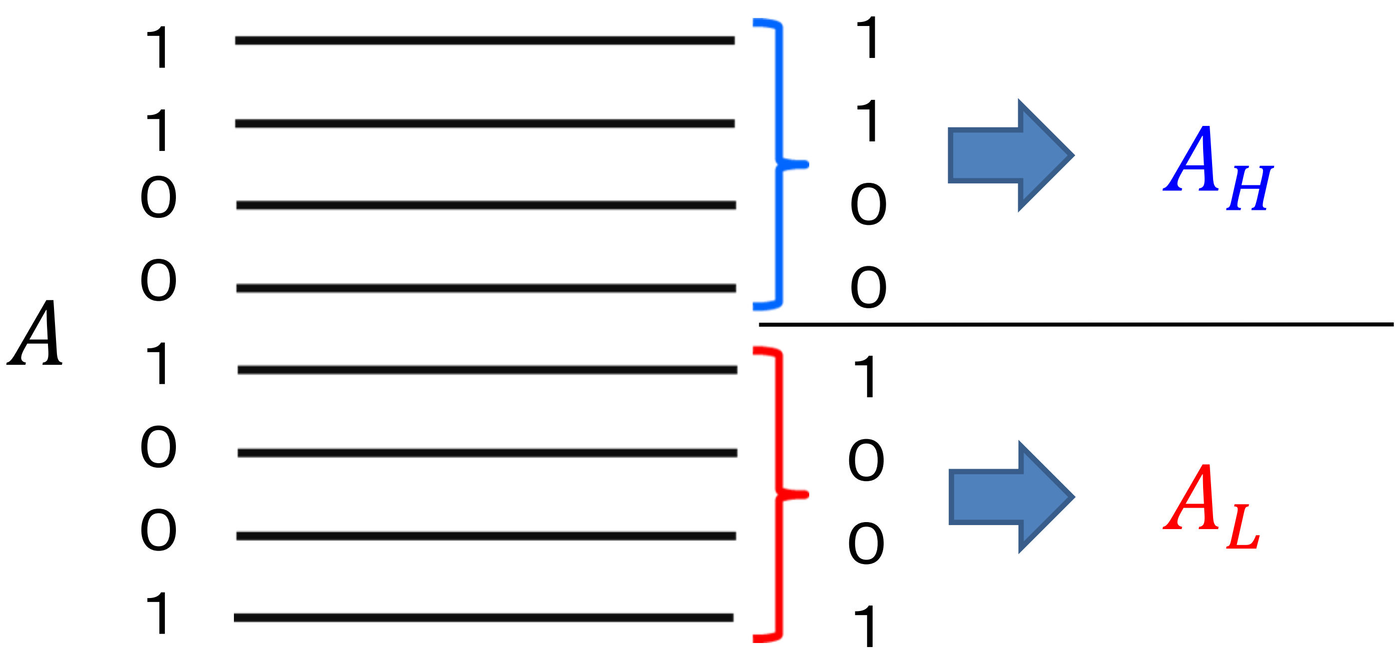
演算で用意する
メモリを小さくできる

「方法序説」
難問は、それを解くのに適切かつ必要なところまで分割せよ

ルネ・デカルト
René Descartes 1596 - 1650
フランス生まれの哲学者、数学者。合理主義哲学の祖

提案するDivide & Conquer法

例：8bit（A=11001001）について考える。

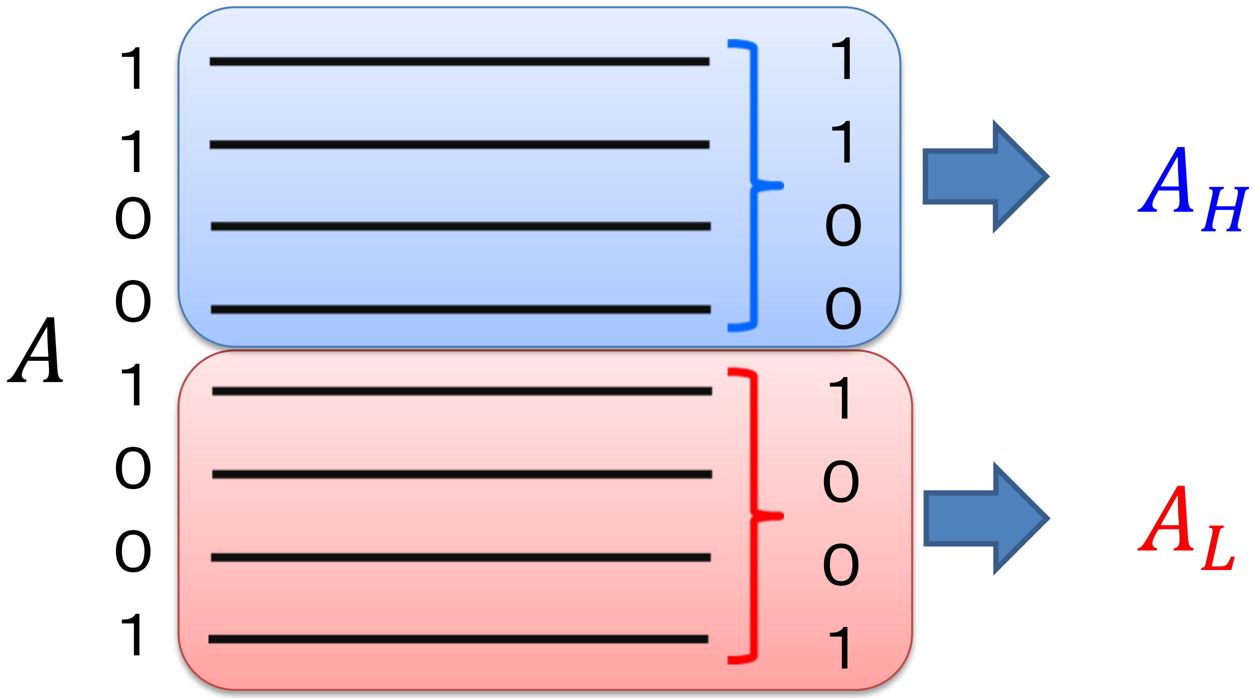


$$A^2 = A_H^2(8bit左シフト) + 2A_HA_L(4bit左シフト) + A_L^2$$

8bit x 8bit 乗算 $[A^2]$ を

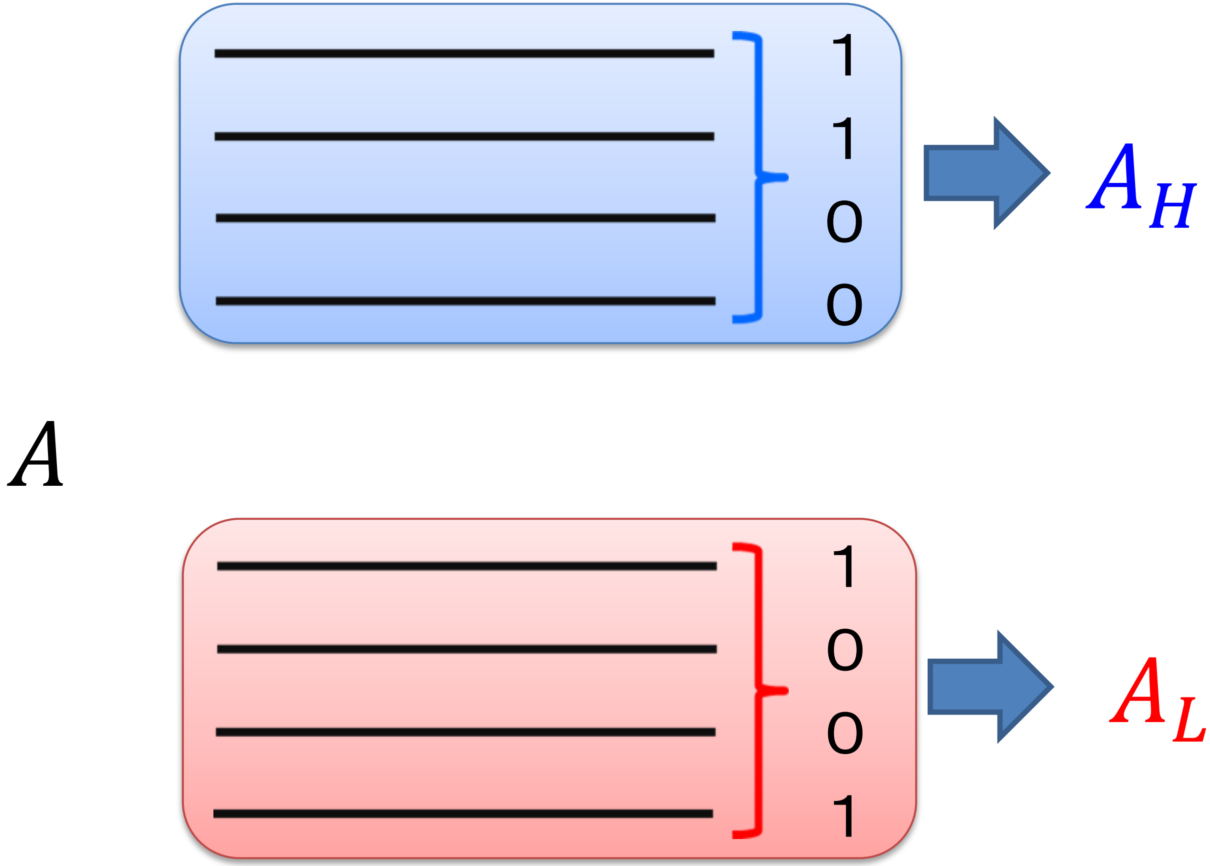
4bit $[A_H]$, $[A_L]$ 毎の計算で求める。

提案するDivide & Conquer法



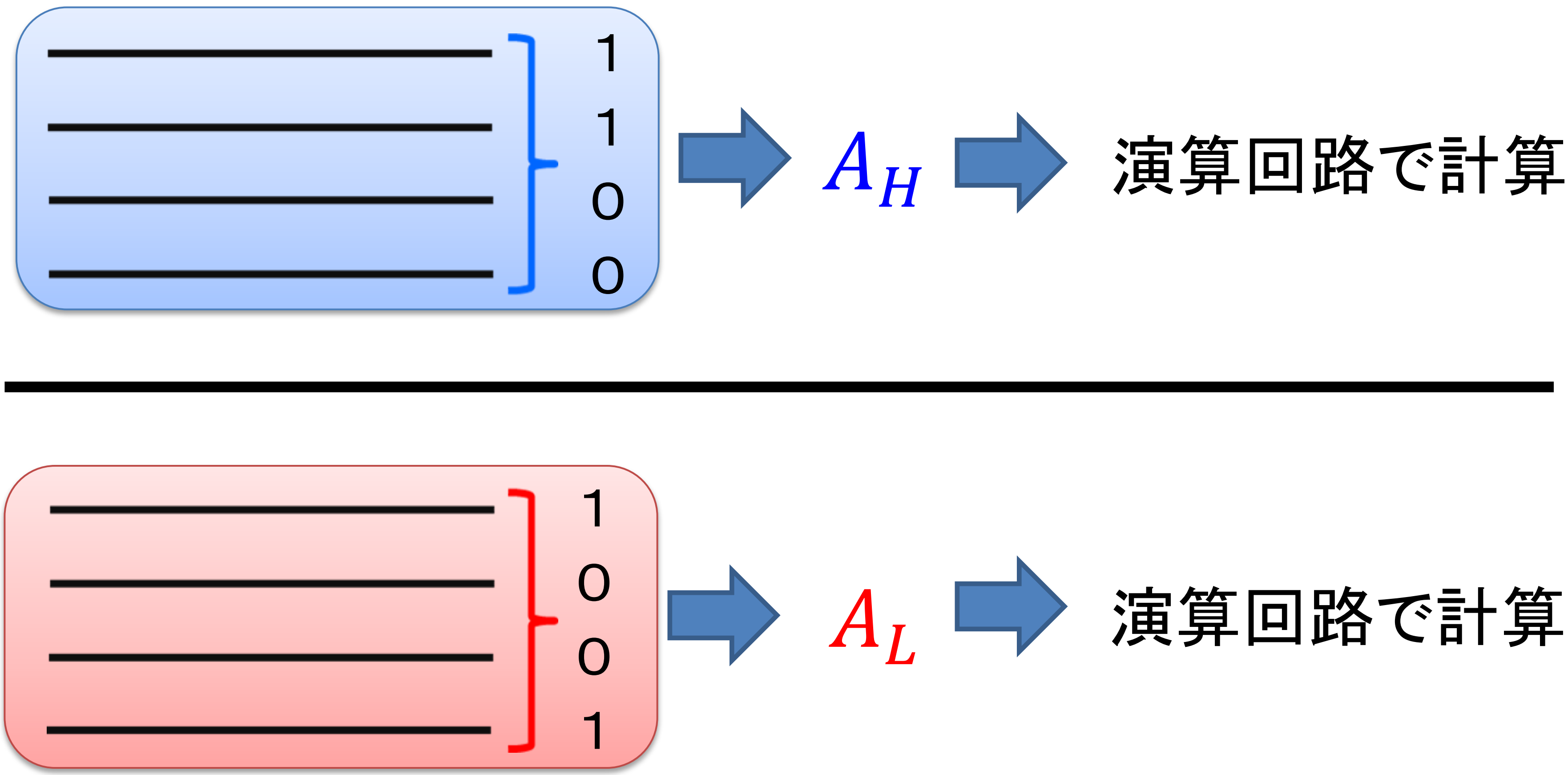
入力、出力された値を上下に分割（Divide）

提案するDivide & Conquer法



入力、出力された値を上下に<u>分割（Divide）</u>

提案するDivide & Conquer法

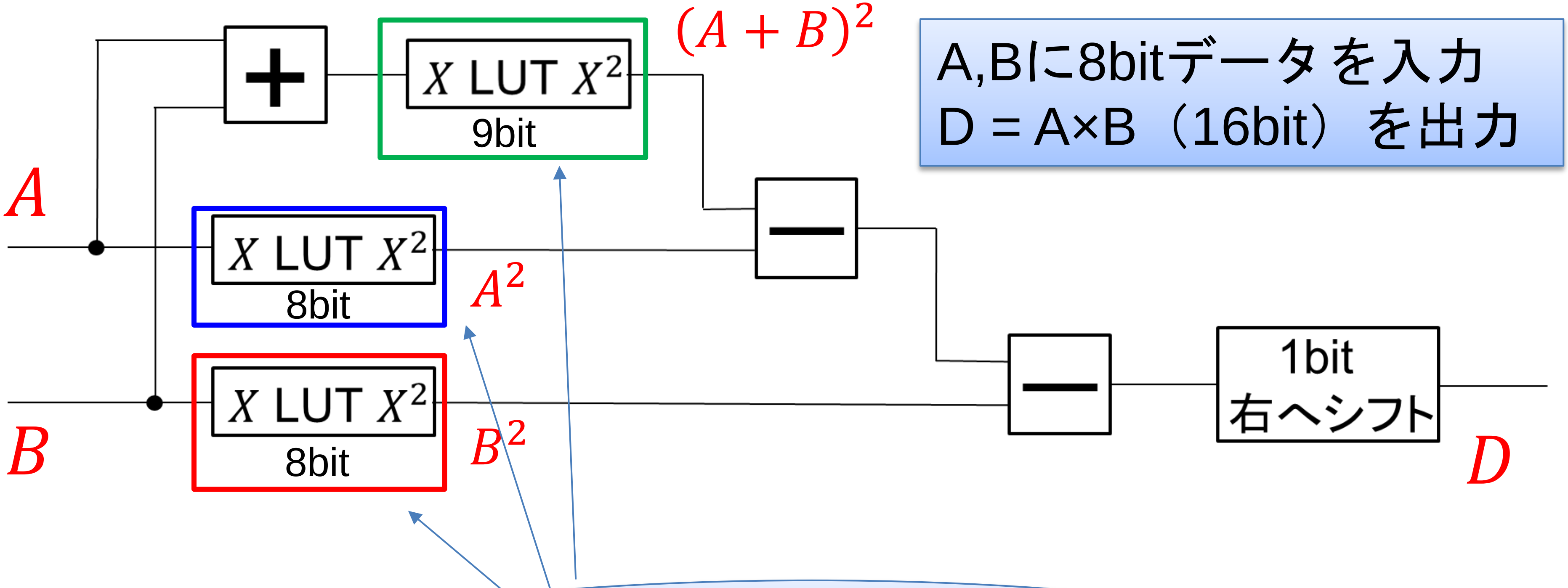


分割した値をそれぞれ計算：<u>征服（Conquer）</u>

OUTLINE

- 研究背景
- 検討乗算アルゴリズム
- **<u>検討アルゴリズム乗算回路の設計とシミュレーション検証</u>**
- 各方式の比較
- 今後の検討回路
- まとめ

検討乗算回路(8bit)（Divide&Conquer 無）

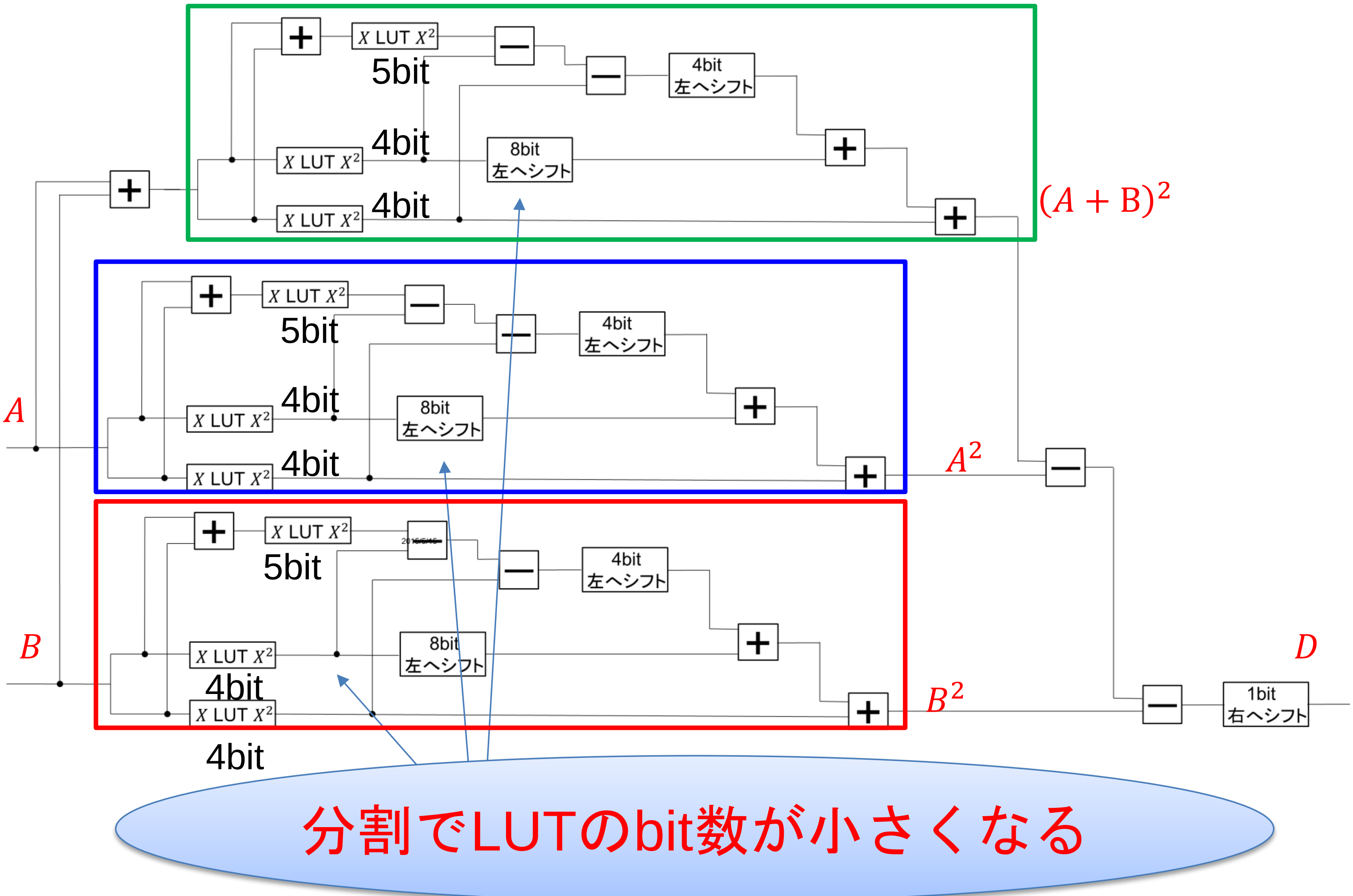


A,Bに8bitデータを入力
D = A×B（16bit）を出力

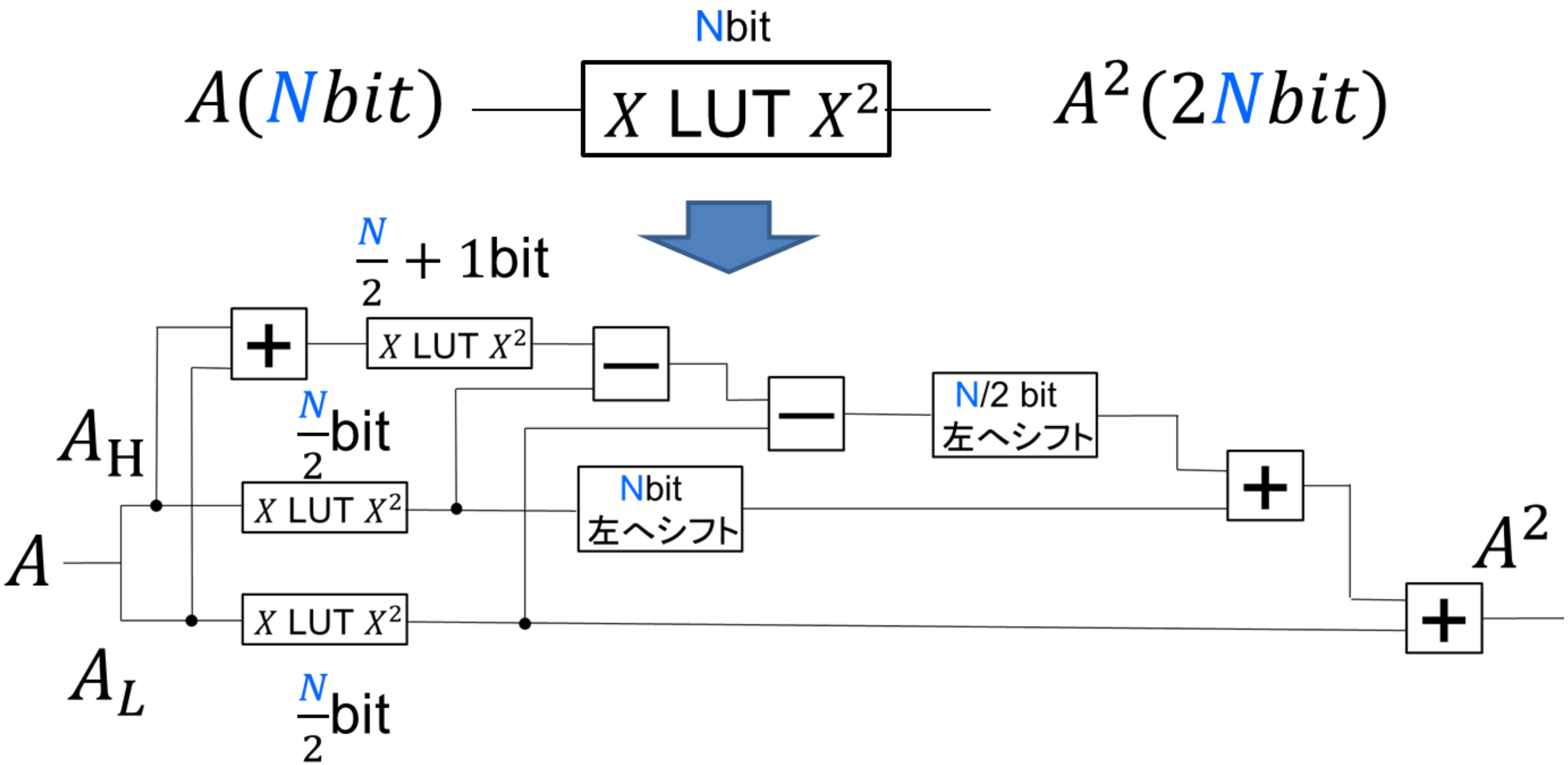
8bit用のLUTを用意するため
回路規模が大きくなる

この回路をさらに変換する

検討乗算回路(8bit)（Divide&Conquer有）



検討アルゴリズム乗算回路(Nbit)

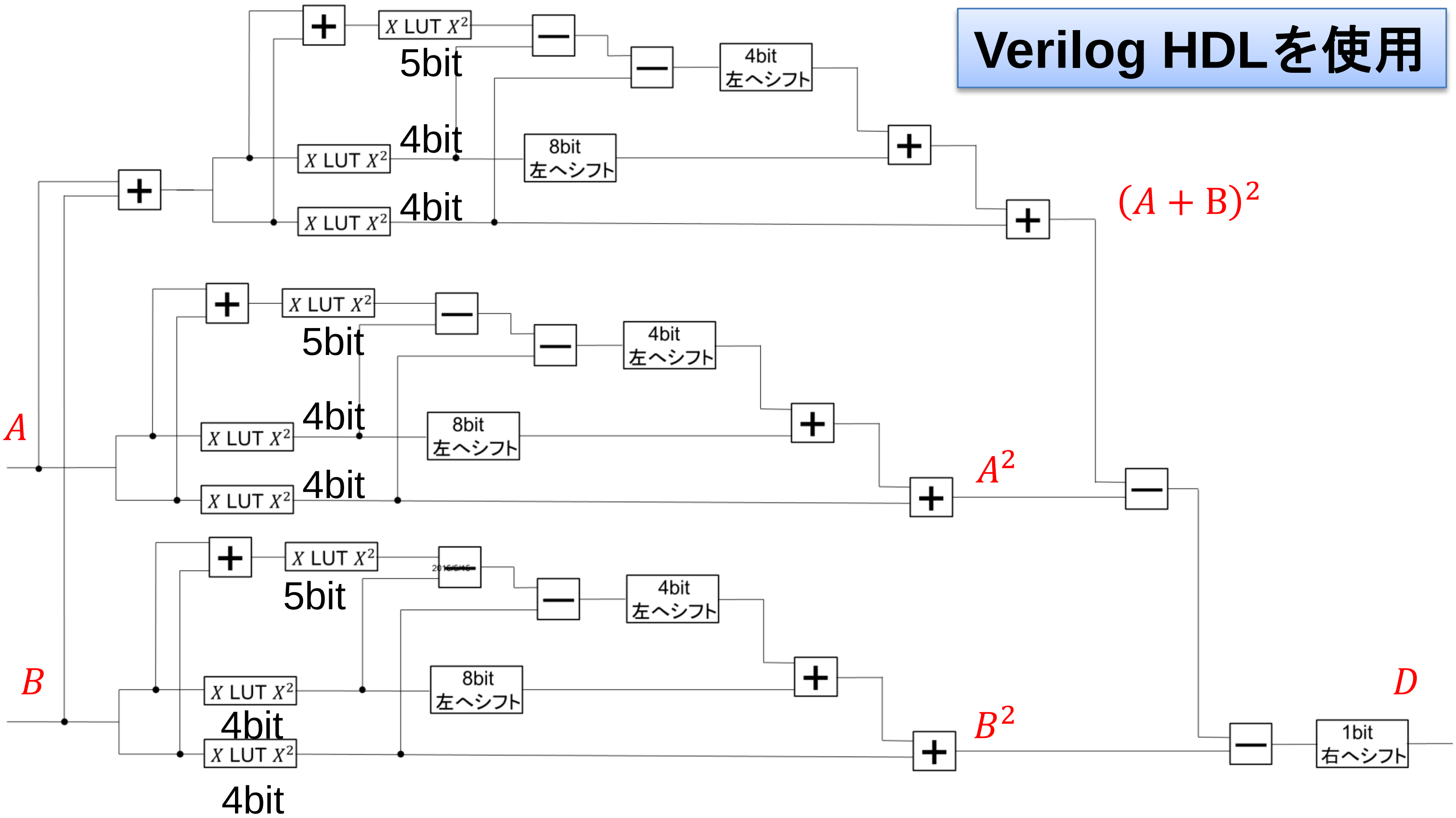


Divide & Conquer 方式

X回使用し、LUTサイズを$\left(\frac{1}{2}\right)^X$にできる

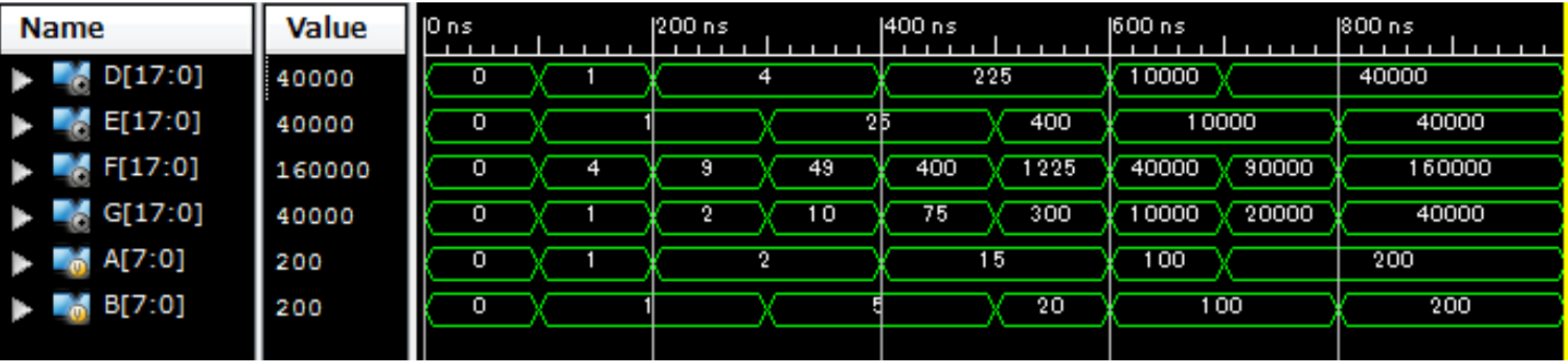
HDLシミュレーションによる検証

Verilog HDLを使用



8bit×8bit回路構成で正しい乗算結果が得られることを検証

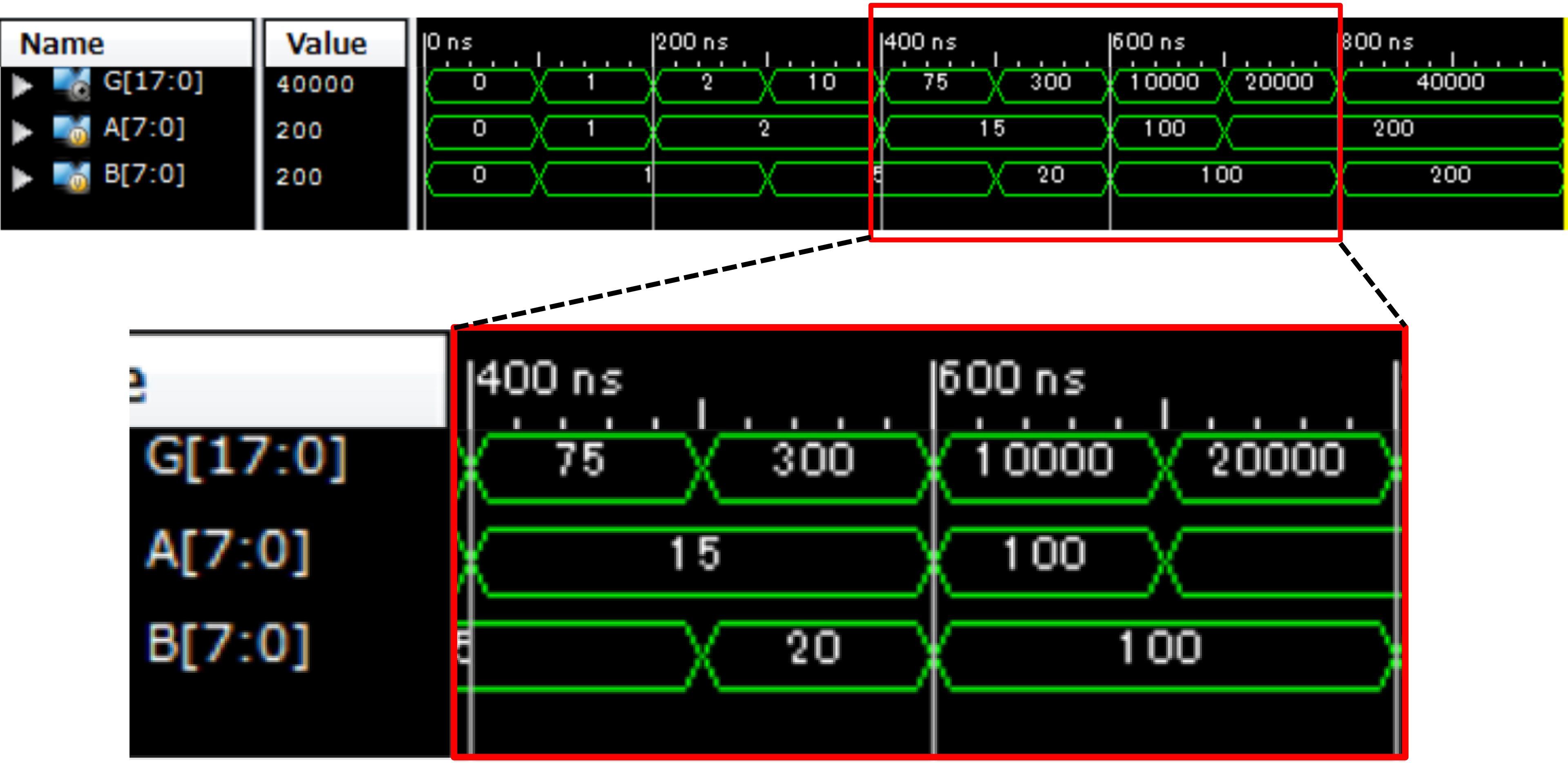
RTLシミュレーション (8bit × 8bit)



入力値A , Bの値を100ns、200ns毎に変化させ、
その区間における演算結果を波形上に表示

$$D(18bit) = A^2$$
$$E(18bit) = B^2$$
$$F(18bit) = (A + B)^2,$$
$$G(18bit) = \frac{1}{2}\{(A + B)^2 - A^2 - B^2\}$$

RTLシミュレーション（8bit × 8bit)



入力Ａ,Ｂに対しＧ＝Ａ×Ｂが出力

RTLシミュレーション

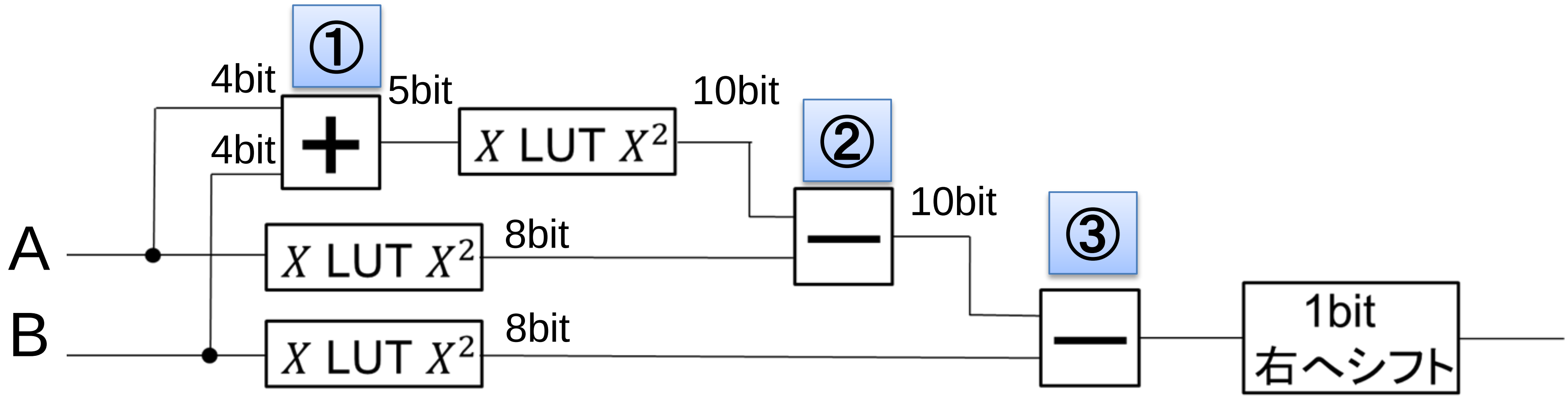
- 同様に8bit×8bitのRTLシミュレーションを行い、
 256×256通りの結果を確認した

- 16bit×16bitのRTLシミュレーションを行い、
 65536×65536通りの結果を確認した

OUTLINE

- 研究背景
- 検討乗算アルゴリズム
- 検討アルゴリズム乗算回路の設計とシミュレーション検証
- 各方式の比較
- 今後の検討回路
- まとめ

4×4bitの乗算器比較説明

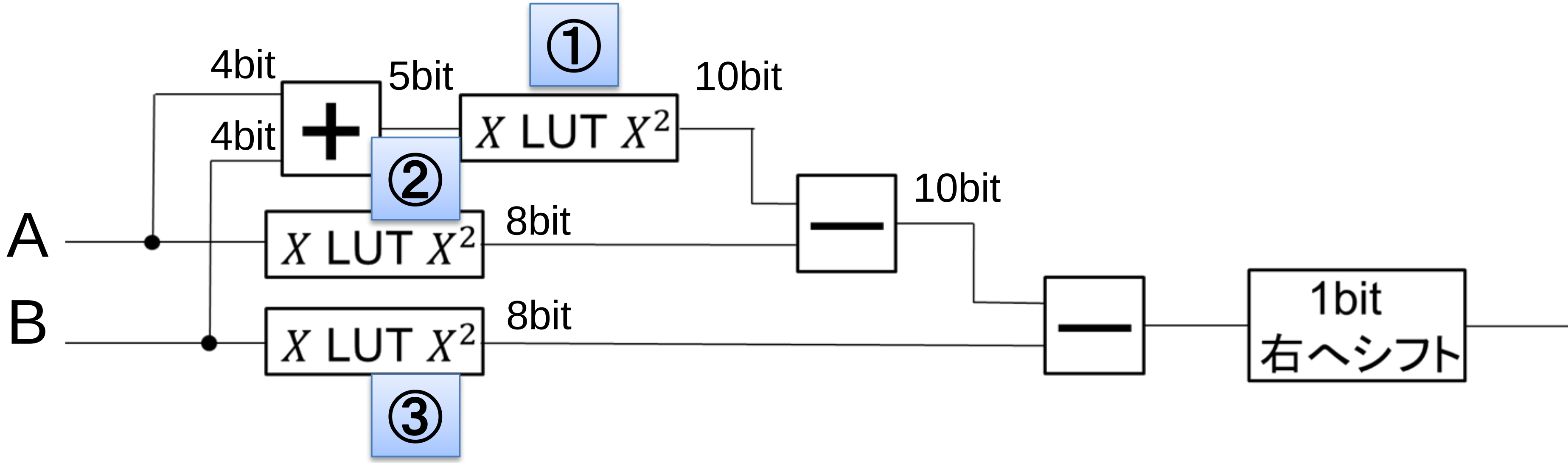


全加算器の数は加数、被加数どちらか高いほうを参照にするので
①：4個
②：10個
③：10個

合計24個

加算器1個、減算器2個あるので
加算(減算)回数は合計3回
LUTは3個

4×4bitの乗算器比較説明



各LUTの入力アドレスの数と、出力bitの数を掛け合わせると

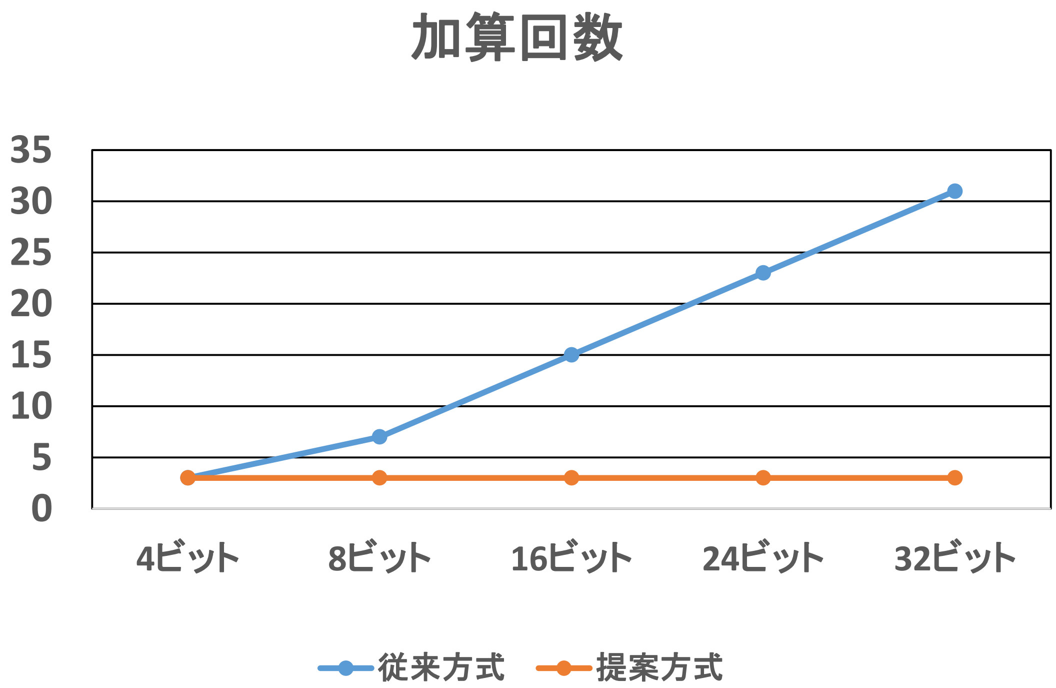
①：入力アドレス：5bit($2^5 = 32$) 出力bit＝10bit,
32×10＝320bit

②：入力アドレス：4bit($2^4 = 16$) 出力bit＝8bit,
16×8＝128bit

③：1入力アドレス：4bit($2^4 = 16$) 出力bit＝8bit,
16×8＝128bit

合計：320+128+128=**576bit**

各乗算器の比較表



図は加算回数をグラフ化している。

従来方式よりも提案方式のほうが加算回数が少なくなっている。

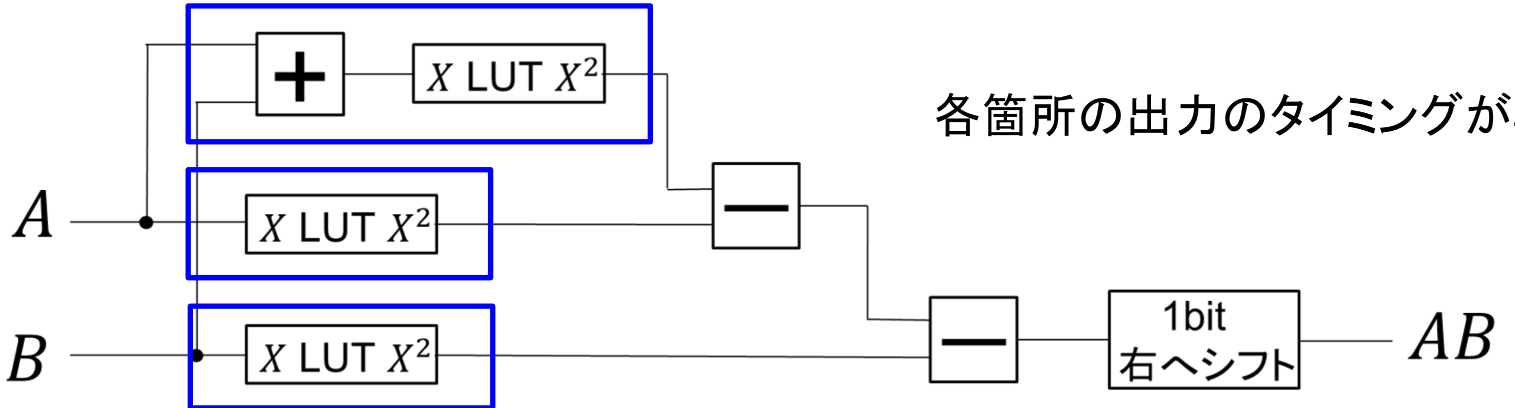
OUTLINE

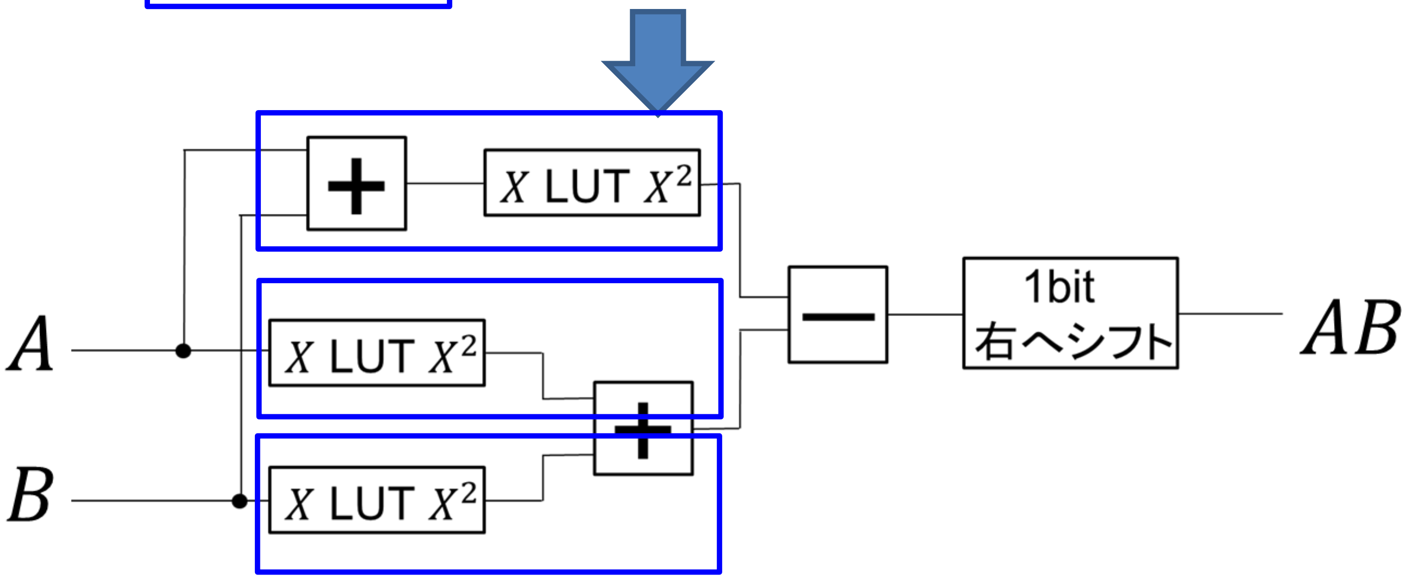
- 研究背景
- 検討乗算アルゴリズム
- 検討アルゴリズム乗算回路の設計とシミュレーション検証
- 各方式の比較
- <u>今後の検討回路</u>
- まとめ

今後の検討回路

- 演算時間を考慮した回路設計
- 演算器を逐次的に使用した回路設計
- 別のアルゴリズムを用いた回路設計

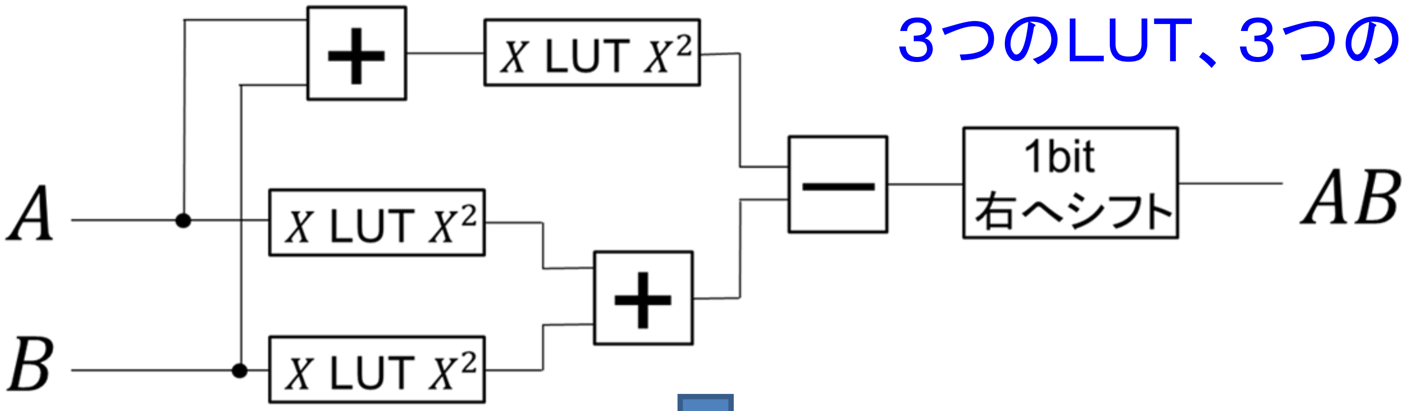
演算時間を考慮した回路設計

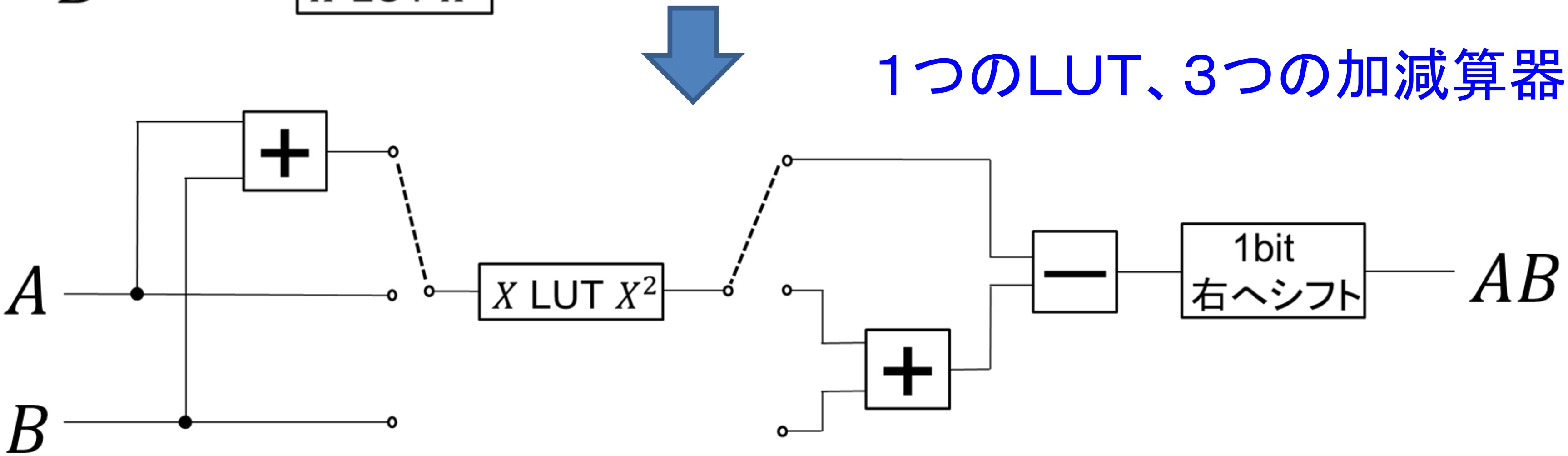




どの経路も加算器とLUTを通るので出力タイミングが等しい

乗算器を逐次的に使用した回路設計



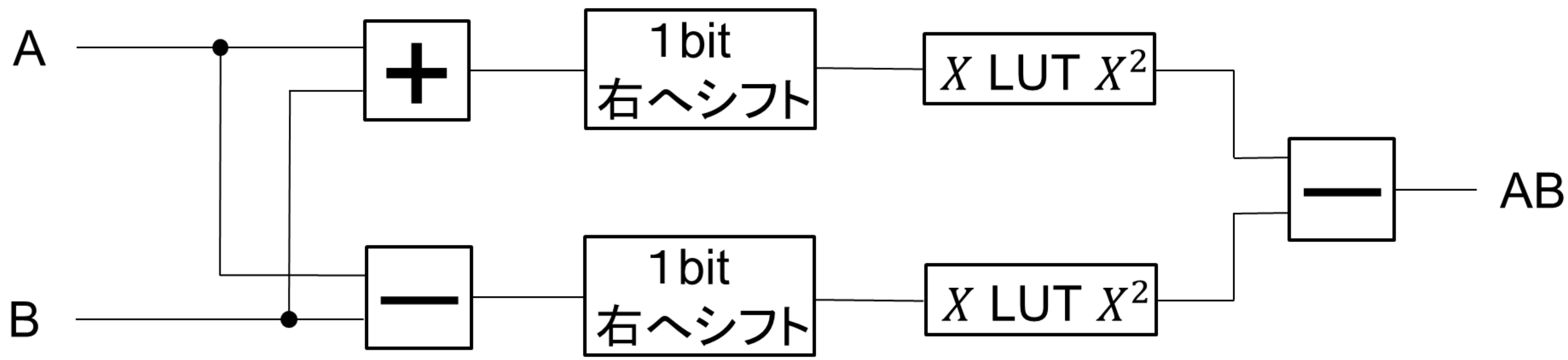


- 1つのLUTを$A^2, B^2, (A+B)^2$ の計算に逐次的に使用
  全体として3分の1の回路量、計算時間は3倍

別のアルゴリズムを用いた回路設計

次の計算アルゴリズムも有力である。

$$AB = \frac{1}{4}\{(A+B)^2-(A-B)^2\}$$



加算回数:1回

減算回数:2回

LUT参照回数:2回

OUTLINE

- 研究背景
- 検討乗算アルゴリズム
- 検討アルゴリズム乗算回路の設計とシミュレーション検証
- 各方式の比較
- 今後の検討回路
- まとめ

まとめ

- ２乗則による乗算アルゴリズム･回路を検討
- Divide & Conquer 法により回路量削減を検討
- 検討アルゴリズムで乗算器をRTLレベル設計
  シミュレーションによる検証
- 従来方式と提案方式の比較をおこなった。

今後の課題

- 回路規模低減、計算スピード向上のための回路改良
- 従来法との回路量、計算スピードの比較･評価
- 他方式での乗算器検討
- Divide & Conquer方式を用いた回路との比較

Ｑ＆Ａ（千葉での学会）

Ｑ１：既存の乗算ブロックとの比較をし、どの点（速度、面積、消費電力）で優れているのかを知りたい。

Ａ：今後の検討課題として扱わせていただきます。

Ｑ２：LUTの計算スピードを同一のものとして扱っている印象を受けた。LUTはサイズによって扱う時間が異なるので単にLUTを使えばいいというわけではないと思う。

Ａ：同一視していましたので、今後、それらの速度についても検討したいと思います。

Ｑ＆Ａ（千葉での学会）

Q3：乗算器は様々な手法がある。この研究のどこからが新しい手法なのか？　提案されている手法はすでに存在するのでは？　Divide&Conquer方式もすでに存在するのでは？

A：自分が調べた限りでは、そういった論文は見つからなかった。なのでこの研究そのもの$\frac{1}{2}\{(A+B)^2 - A^2 - B^2\}$や、Divide&Conquerが新しいものだと自分は考えています。

Final Statement

再帰的アルゴリズム (Recursive Algorithm)

$$\underset{2N}{A} \times \underset{2N}{B} \quad \rightarrow \quad \underset{N}{A_H} \times \underset{N}{A_L}, \quad \underset{N}{B_H} \times \underset{N}{B_L}$$

Recursion (Fractal) is beautiful !