小面積並列乗算器の構成法に関する研究

川島 裕崇

小面積並列乗算器の構成法に関する研究

要旨

半導体設計製造技術の進展により，VLSI の集積度はますます向上し多くの機能が 1 つのチップ上に搭載されるようになっている．乗算は多くのアプリケーションで用いられる基本的な算術演算の一つである．現在では多くのプロセッサや専用回路では並列乗算器によって乗算を行い，乗算命令を高速に実行している．並列乗算器を搭載していないプロセッサでは，乗算命令は加算などの繰り返しによって実現されるが，乗算命令の実行時間が非常に長くなる．近年では大規模な汎用プロセッサのみではなく，多くの組込み機器用プロセッサや特定用途向け集積回路でも並列乗算器が搭載されている．並列乗算器を搭載することにより，1 サイクルから数サイクル程度で乗算命令を実行できるようになる．

加算器，乗算器などの算術演算回路は，回路の構成法によって性能が大きく異なる．そのため，演算の種類ごとに多くの構成法が研究され，実用化されている．並列乗算器では，性能を向上させるために，主に遅延時間の削減を目的とした研究が多く行われてきた．一方，並列乗算器は加算器や論理演算回路などと比較して回路規模が大きくなる．組込み機器用プロセッサなどの小規模なプロセッサでは，キャッシュ等の回路面積の大きいリソースを搭載していない場合が多く，並列乗算器の占める面積の割合が大きい傾向にある．また，高い性能が求められるシステムでは，GPU などのメニーコアデバイスが利用され始めている．このようなデバイスでは高い演算性能を持つ多数のコアが実装されており，回路面積の点で並列乗算器の占める割合が大きくなる．このような並列乗算器の面積の占める割合が大きいようなハードウェアでは，並列乗算器を小面積化することによる全体の面積へのインパクトが大きいと考えられる．本論文では，小面積な並列乗算器の構成法として，Karatsuba アルゴリズムに基づく並列乗算器とオペランドの和を用いた部分積生成法に基づく並列乗算器を提案する．

Karatsuba アルゴリズムに基づく小面積並列乗算器の構成法では，Karatsuba アルゴリズムを並列乗算器に適用した場合に効率の良い構成法を示す．Karatsuba アルゴリズムは乗算の乗数，被乗数を上位ビット，下位ビットに分割し，それぞれの乗算結果の統合方法を工夫することにより，乗算全体の演算量を削減する．Karatsuba アルゴリズムは，ソフトウェアでは多倍長乗算を行うために用いられることが多い．Karatsuba アルゴリズムを用いると少ないセル数で並列乗算器を構成できる．本論文では，Karatsuba アルゴリズムに並列乗算器に適用するのに適した変更を加

えた．Karatsubaアルゴリズムに基づく並列乗算器を設計し評価したところ，既存の並列乗算器と比較して入力ビット幅が32ビットの場合は約10%，64ビットの場合は約25%小面積となった．また，Karatsubaアルゴリズムを用いて小面積な並列乗算器構成する場合，有効である入力ビット幅について検討し，考察を行った．$0.35\mu m$から$90nm$のプロセスで検討したところ，Karatsubaアルゴリズムは入力のビット幅が18ビットから36ビットでは1回適用することが有効であり，それ以上では2回適用することが有効であることがわかった．

次にオペランドの和を利用した部分積生成法に基づく並列乗算器を提案する．部分積をオペランドの和を用いて表現する事によって，部分積のビット数を約半分に削減する．部分積のビット数を削減することにより，部分積の加算回路を小面積化できるため，乗算器全体の面積を削減することができる．本論文では部分積が生成される順序に合わせて部分積の配置と部分積加算回路の構成を工夫することにより，小面積な構成でも高速に乗算を行うことを可能にした．また，オペランドの加算を並列化することで部分積生成を高速化している．提案した部分積生成法を用いた並列乗算器は，従来の並列乗算器と比較して32ビットで約30%，64ビットで約35%小面積となった．

従来の半導体製造プロセスでは配線のためのリソースが限られていたため，小面積な並列乗算器を構成するためには，素子面積が小さくかつ配線が容易な構成法を選択することが有効と考えられてきた．一方，半導体製造技術の進歩により近年の半導体製造プロセスでは，十分な配線のためのリソースが与えられるようになってきた．本研究の成果は，配線の容易さよりも素子面積を優先して設計することにより，小面積な並列乗算器を設計可能であることを示している．

Studies on Small Area Parallel Multipliers

Abstract

More and more functions are implemented on a semiconductor chip due to advance of semiconductor manufacturing technology. The multiplication is a fundamental arithmetic operation which is used in various applications. In processors without parallel multipliers, the multiply operation is implemented as iterations of the add and other operations. In such case, the multiply operation requires longer latency. Currently, many processors have hardware parallel multipliers, such as micro controllers for embedded systems, dedicated circuits. Parallel multipliers in these processors can finish a multiply operation within several cycles.

The performance of arithmetic circuit such as adders and multipliers varies by the construction methods. Then many construction methods have been developed for each arithmetic operation. For parallel multipliers, many researches aims to reduce the delay time. And only a few methods for reducing the circuit area exist. Parallel multipliers are quite large scale circuits, and occupy a large amount of processor cores. Reducing the area of parallel multipliers can contribute to reducing the amount of area of processor cores. In this dissertation, two construction methods for reducing the circuit area of parallel multipliers, a construction method based on Karatsuba algorithm and a construction method based on partial product generation utilizing the sum of the operands, are shown.

First, parallel multipliers based on Karatsuba algorithm are proposed. Karatsuba algorithm is used to perform multi-word multiplication in software implementations. Karatsuba algorithm separate operands into upper and lower half words, and merge products of half-word multiplications into final product efficiently. Parallel multipliers can be constructed in small amount of logic circuit by applying Karatsuba algorithm. In this dissertation, Karatsuba algorithm is modified to be suitable for applying parallel multipliers. Evaluations show that 32-bit and 64-bit parallel multipliers with modified Karatsuba algorithm is approximately 10% and 25% smaller than existing parallel multipliers. This dissertation also shows that Karatsuba algorithm works effectively for reducing circuit area for 18-bit to 36-bit multiplication. For multiplication of larger input width, it is more effective to apply Karatsuba algorithm recursively.

Second, parallel multipliers based on partial product generation utilizing the sum of the operands is proposed. The amount of partial product bits are reduced into a half of basic partial products by utilizing the sum of the operands. The circuit area of partial product accumulators can be reduced by reducing the amount of partial product bits, and the circuit area parallel multipliers also can be reduced. In this dissertation, the arrangement of partial product is optimized by considering the output order of the sum of the operands. The performance of multipliers utilizing the sum of operands can be improved with small area overhead by this optimization. 32-bit and 64-bit parallel multipliers utilizing the sum of operands are approximately 30% and 35% smaller than existing multipliers.

Previous semiconductor manufacturing processes can provide only limited resources for routing. It have been believed that selecting construction methods which have good routability and need small amount of cell area is effective for constructing small area parallel multipliers. On the other hand, recent manufacturing processes provide sufficient resource for routing. In this dissertation, small area parallel multipliers can be designed by construction methods which focus on reducing the amount of cell area under the design environment with sufficient routing resources.

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究の背景

半導体設計製造技術の進展により，VLSI の集積度はますます向上し多くの機能が 1 つのチップ上に搭載されるようになっている．乗算は多くのアプリケーションで用いられる基本的な算術演算の一つであり，現在では多くのプロセッサや専用回路では並列乗算器によって乗算を行っている．並列乗算器は，入力の全ビットが 1 サイクルで与えられ，組み合わせ回路で実現される回路である．並列乗算器を搭載していないプロセッサでは，乗算命令は加算などの繰り返しによって実現され，乗算命令の実行時間が非常に長くなる．近年では大規模な汎用プロセッサのみではなく，多くの組込み機器用プロセッサや特定用途向け集積回路でも並列乗算器が搭載されている．並列乗算器を搭載することにより，1 サイクルから数サイクル程度で乗算命令を実行できるようになる．

並列乗算器は加算器や論理演算回路などと比較して規模が非常に大きい回路である．組込み機器用プロセッサなどの小規模なプロセッサでは，キャッシュ等の回路面積の大きいリソースを搭載していない場合が多く，並列乗算器の占める面積の割合が大きい傾向にある．また，高い性能を求められるシステムでは，GPU(Graphics Processing Unit) などのメニーコアデバイスが利用され始めている．このような高い演算性能を要求されるデバイスでは各コアに並列乗算器を搭載していることが多く，並列乗算器の全体の回路面積に占める割合が大きくなると考えられる．このようなハードウェアでは，並列乗算器を小面積化することによる全体の面積へのインパクトが大きいと考えられる．回路面積を削減することにより，さまざまな利点が生まれる．まず，回路の面積を削減することにより，1 枚のウエハ上でより多くのチップを製造することができるため，製造コストを削減することができる．メニーコアデバイスでは，小面積化によって生まれた余剰面積にさらにコアを実装することにより，デバイスあたりのコア数を増やすことができるため，間接的に性能の向上に寄与できる可能性もある．半導体の微細化により，リーク電流による消費電力の増大が問題となっており，さらなる微細化により，半導体の消

費電力全体に対して，リーク電流による消費電力が支配的になるといわれている．リーク電流は回路の面積に比例して増加するため，回路を小面積化することによって削減可能である．

コストや設計期間が重視される ASIC では，スタンダードセル方式を用いた設計が広く採用されている．ASIC では演算回路もスタンダードセル方式で設計されることが多くなってきている．スタンダードセル方式では，セルと呼ばれる汎用部品を単位として回路の設計を行う．スタンダードセル方式においては，回路の面積は回路を構成する素子面積とそれらを接続する配線の面積によって決まる．配線量の多い回路ではセルの上だけでは配線が収まりきらず，回路の面積を広げて配線を行う必要がある．そのため，従来から小面積な演算回路を設計するためには，用いる素子面積が小さく，かつ配線がしやすい構造を持つ演算回路の構成法が用いられてきた．近年の半導体製造プロセスでは，配線に用いるメタル層の数が増加している．配線層数の増加により，セルの上を通ることのできる配線量が増加する．そのため，配線量が多くなっても回路面積の増加は小さく抑えられるようになってきている．すなわち，配線量の影響が小さくなり，相対的に素子面積が回路面積に直接反映される．小面積な演算回路を構成するためには，配線量が多くても用いる素子面積が小さくなることに重点を置いて構成法を選択することが有効であると考えられる．

加算器，乗算器などの算術演算回路は，回路の構成法によって性能が大きく異なる．そのため，演算の種類ごとに多くの構成法が研究され，実用化されている．並列乗算器では，性能を向上させるために，主に遅延時間の削減を目的とした研究が多く行われてきた．

1.2 論文の概要

本研究は，回路面積の小さい並列乗算器の構成法を開発することを目的とする．本論文では従来手法よりも小面積な並列乗算器を構成できる二つの構成法を示す．

Karatsuba アルゴリズムに基づく並列乗算器の構成法では，Karatsuba アルゴリズムを並列乗算器に適用した場合に効率の良い構成法を示す．Karatsuba アルゴリズムは，ソフトウェアでは多倍長乗算を行うために用いられる．Karatsuba アルゴリズムは乗算の乗数，被乗数をそれぞれ上位ビット，下位ビットに分割する．通常は分割された数の組み合わせの 4 回の乗算が必要となるが，Karatsuba アルゴリズムでは 3 回の乗算で済むように工夫されている．Karatsuba アルゴリズムを用いる

と少ないセル数で並列乗算器を構成できると期待される．本論文では，Karatsubaアルゴリズムに並列乗算器に適用するのに適した変更を加えた．Karatsubaアルゴリズムに基づく並列乗算器を設計し評価したところ，既存の並列乗算器と比較して入力ビット幅が32ビットの場合は約10%，64ビットの場合は約25%小面積となった．また，Karatsubaアルゴリズムを用いて小面積な並列乗算器を構成する場合に有効である入力ビット幅について検討し，考察を行った．0.35μmから90nmのプロセスで検討，評価したところ，Karatsubaアルゴリズムは入力のビット幅が18ビットから36ビットでは1回適用することが有効であり，それ以上では2回適用することが有効であることがわかった．

次にオペランドの和を利用した部分積生成法に基づく並列乗算器を提案する．部分積をオペランドの和を用いて表現する事によって，部分積のビット数を約半分に削減する．部分積のビット数を削減することにより，部分積の加算回路を小面積化できるため，乗算器全体の面積を削減することができる．本論文では部分積が生成される順序に合わせて部分積の配置と部分積加算回路の構成を工夫することにより，小面積な構成でも高速に乗算を行うことを可能にした．また，オペランドの加算を並列化することで部分積生成を高速化している．提案した部分積生成法に基づく並列乗算器は，従来の並列乗算器と比較して32ビットで約30%，64ビットで約35%小面積となった．

本論文の構成は以下のとおりである．2章で既存の並列乗算器の構成法について紹介する．特に本論文の提案手法の比較対象となる手法について詳しく説明する．3章ではKaratsubaアルゴリズムに基づく小面積並列乗算器の構成法を示す．4章でオペランドの和を利用した小面積並列乗算器の構成法を示す．5章では本論文で提案する二つの手法を比較し、最も面積の小さい構成について考察する。6章でまとめを行う．

第2章　準備

本章では，本論文で扱う並列乗算器の基本的な構成と，既存の並列乗算器の構成法について紹介する．

2.1　乗算器の概要

2.1.1　本論文で扱う乗算器

乗算器は入力値が 1 サイクルで入力されるのか，複数サイクルに分けて入力されるのかによって構成が大きく異なる．入力値が複数サイクルに分けて入力される構成を直列乗算器 (Serial Multiplier) と呼ぶ．入力値が 1 サイクルで入力される構成を並列乗算器 (Parallel Multiplier) と呼ぶ．また，入力値の一方が 1 サイクルで，他方が複数サイクルで入力されるような構成の場合，直並列乗算器 (Serial-Parallel Multiplier) と呼ばれる．

本論文では，n ビット $\times n$ ビットの並列乗算器を対象とする．本論文では，符号なし整数は 2 進表現，符号付き整数は 2 の補数表現で表されるものとする．並列乗算器は被乗数 X，乗数 Y を入力とし，積 $P = X \times Y$ を出力する．本論文では整数のビット表現は ‘[’ および ‘]’ で囲んで表す．すなわち，X のビット表現は $[x_{n-1}x_{n-2}\cdots x_1x_0]$ とし，Y，P についても同様とする．符号なし乗算では X，Y は n ビット符号なし整数であり，P は $2n$ ビット符号なし整数である．それぞれの値はビット表現を用いて次のように計算される．

$$X = \sum_{i=0}^{n-1} 2^i x_i \tag{2.1}$$

$$Y = \sum_{i=0}^{n-1} 2^i y_i \tag{2.2}$$

$$P = \sum_{i=0}^{2n-1} 2^i p_i \tag{2.3}$$

符号付き乗算では数表現として 2 の補数表現を用いるものとする．X，Y は n ビットの 2 の補数表現の整数である．積 P は $2n-1$ ビットの 2 の補数表現の整数とす

る．それぞれの値はビット表現を用いて次のように計算される．

$$X = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \tag{2.4}$$

$$Y = -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} 2^i y_i \tag{2.5}$$

$$P = -2^{2n-1}p_{2n-1} + \sum_{i=0}^{2n-2} 2^i p_i \tag{2.6}$$

2.1.2 並列乗算器の評価項目

- 面積

  1.1 節で述べたとおり，並列乗算器は回路規模が大きいため小面積な並列乗算器を構成することには多くの利点がある．本論文では 2 種類の面積の値を用いている．一つは素子面積である．素子面積は回路を構成するセルの面積の合計を指す．一般的には論理合成ツールによって出力される面積の値が素子面積に当たる．もう一つは回路面積である．回路面積は，セルの配置や配線を行ったあとの面積を指す．一般的には配置配線ツールによって出力される面積の値が回路面積に当たる．

  配線が複雑な回路では，素子面積以外に配線を行うための領域を確保しなければならない場合がある．その場合，素子面積に対して回路面積が大きくなる．並列乗算器においても配線が複雑になる構成法では，素子面積に対して回路面積が増大する場合が考えられる．本論文では，並列乗算器の面積の評価には主に回路面積を用いている．

- 遅延時間

  遅延時間は，入力が与えられてから出力の全ビットが到着するまでの時間を指す．並列乗算器は加算器や論理演算回路と比較して計算時間が長いため，遅延時間を短くすることが重要となる．並列乗算器が VLSI にとってクリティカルパスとなる場合には，VLSI の動作周波数が並列乗算器の遅延時間によって決まる可能性も考えられる．

  本論文では，以下のように遅延時間を算出する．まず，論理合成ツールによって論理設計，最適化を行う．この時点でツールが出力する遅延時間は配線による遅延時間を含んでおらず，見積り精度が十分でない．次に配置配線ツールを用いて配置配線設計を行う．配置配線の結果からの配線遅延に関連する

パラメータを抽出する．抽出されたパラメータを論理合成ツールに入力することによって，配線遅延を考慮した並列乗算器全体の遅延時間が算出される．

- 配線の複雑さ
  並列乗算器では構成法によって配線の複雑さが大きく異なる．配線の複雑さは配置，配線モデルを定義し，構成法を解析することによって評価することができる [1]．実際の設計においては，配線が複雑になると，配線遅延の増大により乗算器全体の遅延時間が増大したり，配線を行うために回路面積が大きくなるということが考えられる．本論文では，配線の複雑さは遅延時間や回路面積に反映されると考え，配線の複雑さに関して直接的な評価は行わない．

- 消費電力
  近年の半導体回路では消費電力が重要な評価項目となっている．並列乗算器は回路規模が大きいため，消費電力も大きくなると考えられる．本論文では，消費電力は評価の対象外とする．

2.1.3　並列乗算器の構成

ほとんどの並列乗算器は主に次の 3 つの回路で構成される．

1. 部分積生成回路 (Partial Product Generator)
2. 部分積加算回路 (Partial Product Accumulator)
3. 最終加算器 (Final Adder)

部分積生成回路は被乗数と乗数の 1 桁の積を算出する．被乗数と乗数の 1 桁の積を部分積と呼ぶ．部分積のビット数と部分積の個数は部分積生成回路の構成法によって変わる．

部分積加算回路では，生成された部分積を 2 つの数になるまで加算する．一般的には桁上げ保存加算器 (Carry Save Adder) を用いて加算を行う．桁上げ保存加算器は 3 つの数を加算して 2 つの数を出力する．桁上げ保存加算器は全加算器 (Full Adder) を並べた構成であり，桁上げが伝搬しないため全加算器 1 段分の遅延時間で処理が実行される．

最終加算器は部分積加算回路から出力される 2 つの数を加算し，最終的な積を出力する．最終加算器は様々な桁上げ伝搬加算器の構成法を用いて構成される．桁

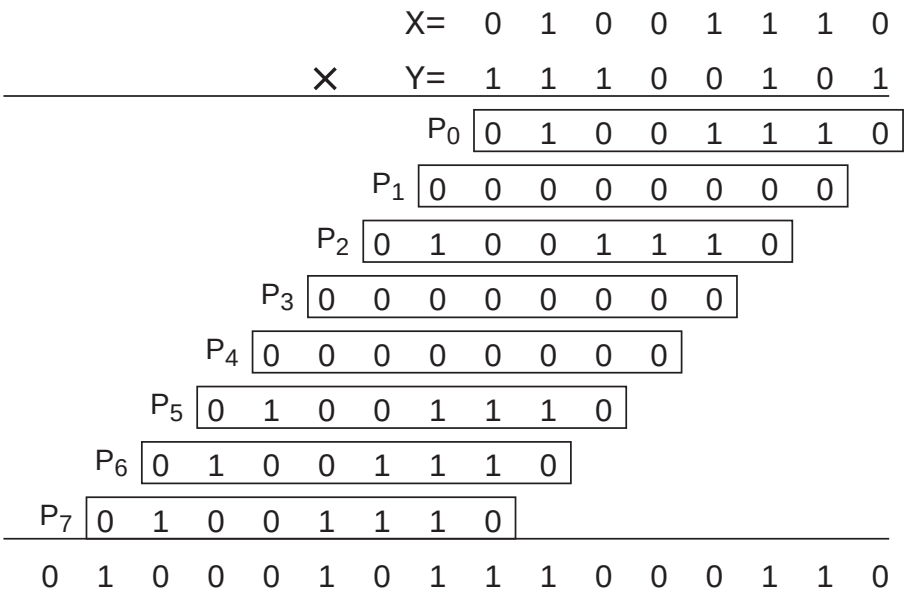


図 2.1: 8 ビット符号なし乗算の基本的な部分積

上げ伝搬加算では桁上げが伝搬するため，加算の完了には入力となる 2 つの数の桁数に依存した遅延時間がかかる．部分積加算回路の出力はビット位置によって到着タイミングが異なる．また，部分積生成回路，部分積加算回路の構成によっても到着タイミングが異なる．そのため，最適な最終加算器の構成はこれらを考慮したうえで決定される．

2.2 部分積生成回路の構成法

2.2.1 基本的な部分積生成法

最も基本的な部分積生成法は，被乗数と乗数の 1 ビットの積を算出して部分積とする．2^i の重みを持つ部分積 P_i は次の式で計算される．

$$P_i = 2^i \cdot X \cdot y_i \tag{2.7}$$

部分積の各ビットは AND 回路で算出される．それぞれの部分積のビット数は n ビットであり，部分積の数は n 個となる．部分積の総ビット数は n^2 ビットとなる．8 ビット符号なし整数乗算の部分積の例を図 2.1 に示す．符号付き乗算の場合は各部分積に符号拡張が必要となるため，わずかに部分積のビット数が増える．

2.2.2　基数 4 の Booth の方法

基数 4 の Booth の方法は乗数を変換し，桁数を約半分に減らす．乗数の桁数が約半分になることにより部分積の個数も約半分に削減できるため，部分積加算の高速化，小面積化が期待できる．基数 4 の Booth の方法では，乗数を桁集合が $\{-2,-1,0,1,2\}$ からなる符号付き 4 進数に変換する．ここでは n を偶数とし，n ビット符号なし整数の変換について説明する．次の式で符号付き 4 進数 b_i が計算される．

$$b_i = -2y_{2i+1} + y_{2i} + y_{2i-1} \tag{2.8}$$

変換後の乗数 Y は $\frac{1}{2}+1$ 桁となり，各桁は 4 のべき乗の重みを持つ．乗数 Y と b_i の関係は次のようになる．

$$Y = \sum_{i=0}^{\frac{1}{2}n} 2^{2i} b_i \tag{2.9}$$

基数 4 の Booth の方法を用いて生成される部分積 P'_i は次のように計算される．

$$P'_i = 4^i \cdot X \cdot b_i \tag{2.10}$$

b_i の桁集合が $\{-2,-1,0,1,2\}$ であるため，部分積の計算はシフトと 2 の補数計算で実現できる．そのため，入力から部分積生成までは入力のビット長に依存せず，定数時間で完了する．8 ビット符号なし整数乗算の基数 4 の Booth の方法による部分積の例を図 2.2 に示す．符号なし乗算であっても部分積は符号付き整数となるため符号拡張が必要となり，部分積 1 個あたりのビット数は増加する．

Booth の方法は，最初に Booth によって基数 2 の手法が提案され [2]，後に McSorley によって基数 4 以上に拡張された [3]． 基数 2，基数 8，基数 16 などでも Booth の方法を使用して部分積を生成することができるが，並列乗算器では基数 4 の Booth の方法が最もよく用いられている．

2.2.3　その他の手法

2 の補数表現の 2 数の乗算では部分積も 2 の補数表現となるため，各部分積に符号拡張ビットが必要になる．Baugh らの手法では，部分積の符号拡張ビットを整理して符号拡張ビットの総ビット数を削減することができる [4]．伊藤らは乗数と被乗数のビット毎の論理和，論理積を計算し，これらを利用して部分積を生成することで消費電力の小さい並列乗算を可能としている [9]．

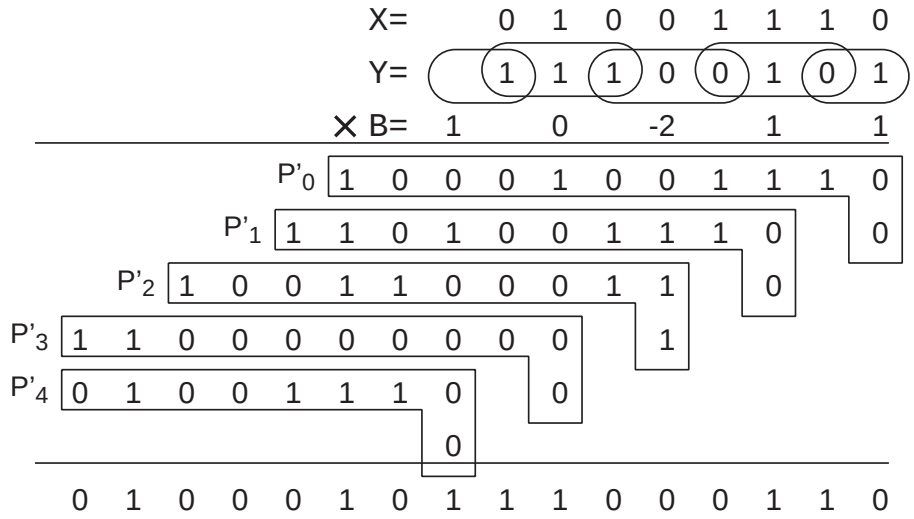


図 2.2: 8 ビット符号なし乗算の基数 4 の Booth の方法による部分積

2.3 部分積加算回路の構成法

2.2 節の手法で生成した部分積は，部分積加算回路で 2 数になるまで加算される．多くの手法では，部分積の加算に桁上げ保存加算器 (Carry Save Adder) と呼ばれる回路を用いる．8 ビット桁上げ保存加算器を図 2.3 に示す．桁上げ保存加算器は 3 つの数 A，B，C を入力とし，2 つの数 S，R を計算する．

$$S + R = A + B + C \tag{2.11}$$

桁上げ保存加算器は全加算器を並べた構成となっており，全加算器 1 段分の遅延時間で計算が可能である．S の各ビットが全加算器の和出力，R の各ビットが桁上げ出力に対応する．多くの部分積加算回路の構成法は，桁上げ保存加算器の組み合わせ方によって特徴付けられる．本節では 2.2.1 節で示した基本的な部分積生成法によって生成された部分積を例として，部分積加算回路を説明する．

2.3.1 配列型の部分積加算回路

配列型の部分積加算回路は，桁上げ保存加算器を配列型に並べた構成で，桁上げ保存加算器の出力に順次部分積を加算していく．配列型の部分積加算回路の構成を図 2.4 に示す．1 段目の桁上げ保存加算器の入力は P_0，P_1，P_2 である．2 段目以降の桁上げ保存加算器は，前段の桁上げ保存加算器の出力と 1 つの部分積を入

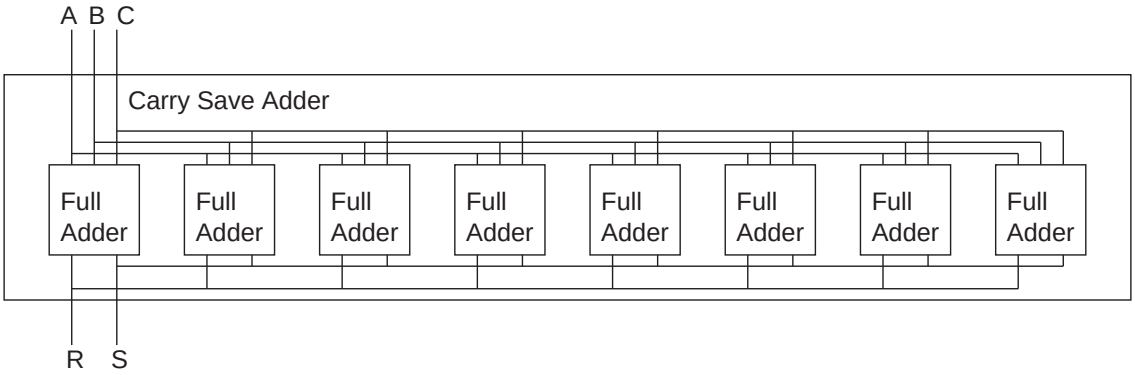


図 2.3: 8 ビット桁上げ保存加算器

力とし，出力は次段の桁上げ保存加算器に接続する．配列型の部分積加算回路の計算を式で示すと次のとおりとなる．

$$S_0 + R_0 = P_0 + P_1 + P_2 \tag{2.12}$$

$$S_i + R_i = S_{i-1} + R_{i-1} + P_{i+2} \qquad (0 < i < n-2) \tag{2.13}$$

配列型の部分積加算回路の出力は S_{n-3} と R_{n-3} となる．配列型は部分積を 1 つずつ加算していくため，部分積の個数に比例する遅延時間がかかる．配置配線設計においては，配線が容易であるため素子面積に対する回路面積の増加は小さく抑えられる．

2.3.2　Wallace 木型の部分積加算回路

Wallace 木型の部分積加算回路は，桁上げ保存加算器を Wallace 木と呼ばれる構成で接続した回路である [5]．図 2.5 に Wallace 木の構成の一例を示す．1 段目の桁上げ保存加算器は部分積を 3 個ずつ加算する．n 個の部分積がある場合，$\lfloor \frac{1}{3}n \rfloor$ 個の桁上げ保存加算器を用いて加算する．加算後の部分積の個数は $\lceil \frac{2}{3}n \rceil$ 個となる．これを繰り返すことで，最終的に部分積が 2 個になるまで加算を行う．桁上げ保存加算器は木状の構成となり，配列型の部分積加算回路よりも必要な段数が小さくなる． そのため，配列型加算回路よりも高速に加算を行うことができる．一方，配置配線設計では配線が複雑になるという特徴があり，配線の混雑度の制限によっては回路の面積が大きくなる場合がある．また，配線が長くなるため，配線による遅延時間が問題となる場合も考えられる．Wallace 木型の部分積加算回路は，桁上げ保存加算器を Wallace 木と呼ばれる構成で接続した回路である

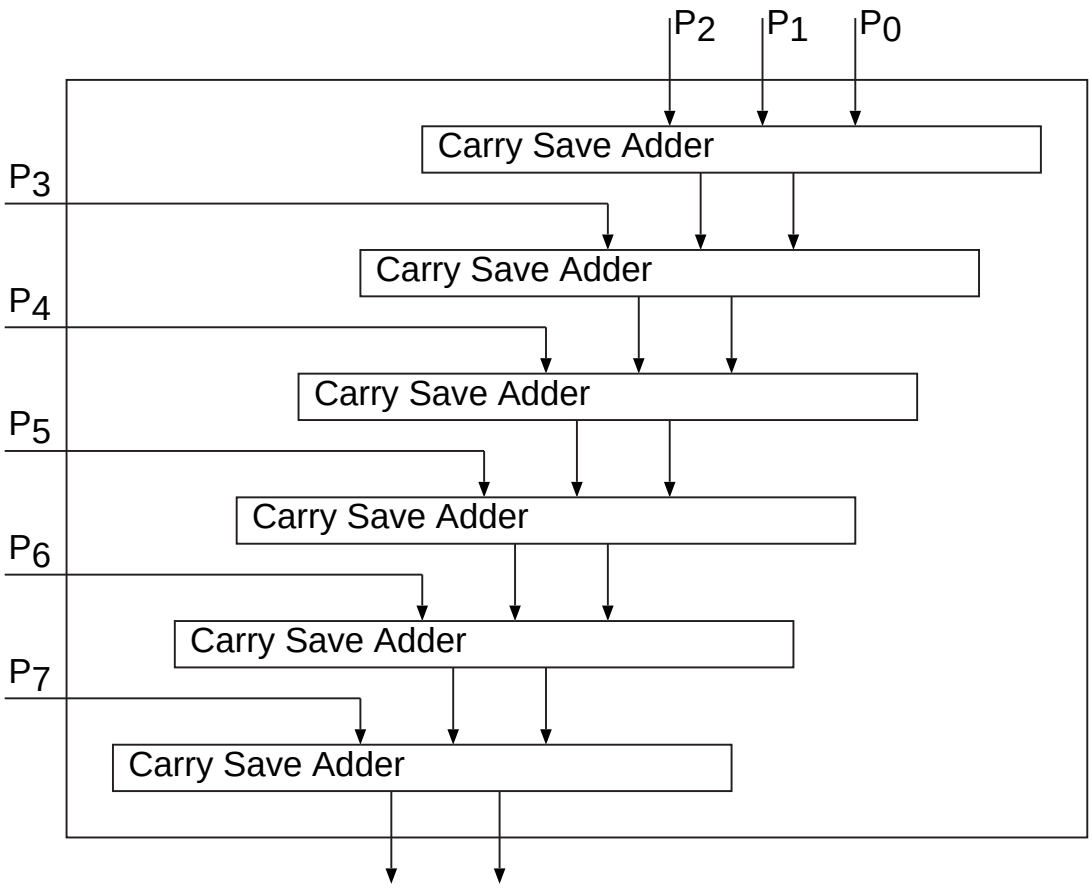


図 2.4: 配列型の部分積加算回路

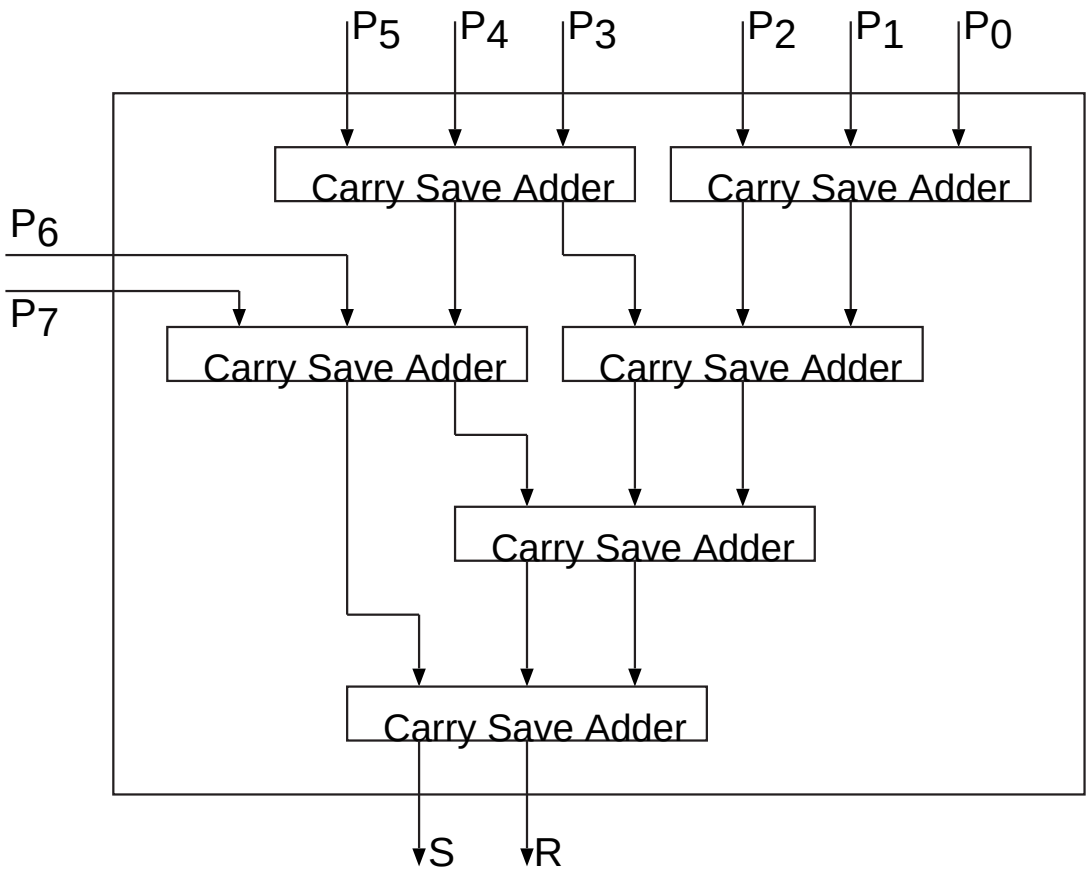


図 2.5: Wallace 木を用いたの部分積加算回路

2.3.3 その他の手法

Wallace 木では桁上げ保存加算器を用いて加算を行なっているため，木構造の対称性が低く配線が複雑になりやすい．そこで，4-2 加算木では 4-2 コンプレッサと呼ばれる回路を用いて加算木を構成している [7]．4-2 加算木は Wallace 木と比較して木構造の対称性が高いため，配置配線が容易になる．冗長 2 進加算を用いた乗算器 [8] では，桁集合 {-1,0,1} からなる冗長 2 進数を用いて部分積の加算を高速に行なっている．加算には桁上げ保存加算器ではなく，冗長二進加算器を用いる．加算木は二分木構造になるため，こちらも Wallace 木等と比較して配置配線が容易になる．

2.4 最終加算器の構成法

最終加算は部分積加算回路から出力された 2 数を加算し，最終的な積を計算する．最終加算器の構成には，桁上げ伝搬加算器の構成法が用いられる．

最も単純な桁上げ伝搬加算器の構成法は，順次桁上げ加算である．全加算器を直列に接続した構成となっており，全加算器の桁上げビットを次段の全加算器に入力する．入力のビット数に比例する遅延時間が必要となるが，最も小面積で実現可能である．より高速に動作する桁上げ伝搬加算器の構成法としては，桁上げ選択加算，桁上げ先見加算等，多数提案されている．これらの多くは並列プレフィクス加算の考え方を用いて整理できる．並列プレフィクス加算は各桁での桁上げの発生，伝搬の条件を求めておく．桁上げの発生，伝搬条件を用いて，ある桁に下位の桁から桁上げが到達するかどうかを計算する．桁上げの到達条件を計算する回路構成によって，前述の桁上げ選択加算や桁上げ先見加算などが実現可能である．これら以外にも並列プレフィクス加算の考えに基づいた構成が多数提案されている．

最終加算器は部分積加算回路の出力を入力として用いる．部分積加算回路の出力はビット位置により出力されるタイミングが異なるため，最終加算器の入力もビット位置により到着タイミングが異なる．そこで並列乗算器の最終加算器では，入力の到着タイミングを考慮して入力のビット位置によって異なる構成を組み合わせるということが行われる．入力の到着タイミングは部分積加算回路の構成によって大きく異なる．また，部分積生成回路の構成や，半導体回路の特性によっても影響を受けるため，最適な構成を決定することは難しい．そこで，入力の到着タイミングを与えることにより，加算器を自動合成する手法が提案されている [10]．

2.5　本研究の位置づけ

本研究は，回路面積の小さい並列乗算器の構成法を開発することを目的とする．一般に回路を小面積化すると遅延時間など他の性能が低下する可能性があるため，設計時に性能のトレードオフを考慮することが重要である．本論文で提案する並列乗算器の構成法は，既存の構成法では実現できなかった小面積で並列乗算器を実現する構成法であり，面積と他の性能のトレードオフにおいては，最も面積の小さい領域をさらに小面積化する構成法である．そのため，本論文では面積の評価を最も重視し，小面積化に伴って他の性能が低下することについては許容する．面積以外の評価については，並列乗算器にとって重要であると考えられる遅延時間についてのみ評価を行う．

本論文では，Karatsubaアルゴリズムに基づく並列乗算器とオペランドの和を用いた部分積生成法に基づく並列乗算器の二つの構成法を示す．一つ目の構成法として，Karatsubaアルゴリズムに基づく並列乗算器を提案する．Karatsubaアルゴリズムは多倍長乗算を効率良く行うアルゴリズムである．これまでにいくつかのKaratsubaアルゴリズムを乗算器へ適用した研究が行われている．[13] ではガロア体上の乗算を行う専用回路に対してKaratsubaアルゴリズムを適用し，従来よりも小面積な乗算器を実現している．[14, 15] では整数の多倍長乗算を行う専用回路でKaratsubaアルゴリズムを用いている．これらの専用回路では順序回路による繰り返し演算で多倍長乗算を行っている．一方，一般的なプロセッサやマイコンに搭載される数十ビット程度の整数乗算を行う並列乗算器にKaratsubaアルゴリズムを適用した例は見られない．そこで，本論文では数十ビット程度の整数を入力とする並列乗算器を対象とする．Karatsubaアルゴリズムを適用した並列乗算器は，本章で示した部分積生成回路，部分積加算回路，最終加算器からなる構成にはならない．本論文では，Karatsubaアルゴリズムを用いて並列乗算器を構成するにあたって回路での実現に適したアルゴリズムの変更を行い，その構成法を示す．

二つ目の構成法として，オペランドの和を用いた部分積生成法に基づく並列乗算器を提案する．オペランドの和を利用した部分積生成法は，2.3節の部分積生成に分類される手法である．オペランドの和を用いる手法は，入力が1ビットずつ与えられる直列乗算器において研究が行われていた [18, 19, 20]．本論文では，並列乗算器でオペランドの和を利用して部分積を生成することにより，部分積のビット数を約半分に削減した．基数4のBoothの方法でも部分積のビット数を約半分に削減しているが，本手法では部分積生成の回路の面積をより小さく抑えること

が可能であり，より小面積な並列乗算器を実現している．また，部分積が生成される順序に合わせて部分積の配置を最適化し遅延，面積を削減した．さらにオペランドの加算を複数に分割し，並列に行うことにより，高速かつ小面積な並列乗算器を構成できることを示す.

第3章　Karatsubaアルゴリズムに基づく小面積並列乗算器

3.1　はじめに

本章では，少ないセル数で乗算器を構成できるアルゴリズムとして，Karatsubaアルゴリズム [11, 12] に着目し，小面積な乗算器を構成する．Karatsubaアルゴリズムは，少ない乗算命令回数で多倍長乗算を効率よく実行するアルゴリズムである．Karatsubaアルゴリズムを並列乗算器に適用した場合，必要なセル数は少ないが，配線が複雑になる．配線の複雑さから，従来はKaratsubaアルゴリズムは並列乗算器には向かないと考えられてきた．しかし，Karatsubaアルゴリズムを用いた並列乗算器は必要なセル数が少ないため，配線層数が多い場合には小面積な乗算器を構成できると考えられる．

本章では，数十ビット程度の並列乗算器を対象とし，Karatsubaアルゴリズムに基づく小面積な乗算器を提案する．提案乗算器では，必要なセル数が少なくなるようにKaratsubaアルゴリズムの計算順序を変更した．また，内部に複数現れる桁上げ伝搬加算器を削減することによって高速化した．

Karatsubaアルゴリズムをそのまま回路として実現した基本構成と，さらに小面積化，高速化を施した構成の2種類を複数のセルライブラリを用いて設計し，評価を行った．Karatsubaアルゴリズムに基づく乗算器は，基本構成で配列型乗算器と比較して回路面積が32ビットで約10%，64ビットで約25%小さくなった．また，小面積化，高速化手法を適用したところ，基本構成と比較して遅延面積積で32ビットで13〜19%，64ビットで23〜33%の改善がみられた．

以降，3.2節でKaratsubaアルゴリズムの説明を行う．3.3節で提案手法とそれを用いた乗算器の構成について述べ，3.4節で提案手法に基づく乗算器を評価する．3.5説ではKaratsubaアルゴリズムの再帰的適用についての考察を行う．3.6節で本章のまとめを行う．

3.2 Karatsuba アルゴリズム

Karatsuba アルゴリズムは少ない乗算命令回数で多倍長乗算を行うアルゴリズムである．n ビット数 X,Y の乗算を考える．X,Y の上位 $\frac{1}{2}n$ ビット，下位 $\frac{1}{2}n$ ビットをそれぞれ X_H, X_L, Y_H, Y_L とすると，X,Y は

$$X = 2^{\frac{1}{2}n}X_H + X_L \tag{3.1}$$

$$Y = 2^{\frac{1}{2}n}Y_H + Y_L \tag{3.2}$$

と表現される．この表現を用いると X と Y の積は

$$XY = \left(2^{\frac{1}{2}n}X_H + X_L\right)\left(2^{\frac{1}{2}n}Y_H + Y_L\right) \tag{3.3}$$

$$= 2^n X_H Y_H + X_L Y_L + 2^{\frac{1}{2}n}\left(X_H Y_L + X_L Y_H\right) \tag{3.4}$$

と表される．この段階では，$\frac{1}{2}n$ ビット同士の乗算が 4 回行われている．ここで，$X_H Y_L + X_L Y_H$ を以下のように変形する．

$$X_H Y_L + X_L Y_H = \left(X_H + X_L\right)\left(Y_H + Y_L\right) - X_H Y_H - X_L Y_L \tag{3.5}$$

この変形により，XY は

$$XY = 2^n X_H Y_H + X_L Y_L + 2^{\frac{1}{2}n}\left(\left(X_H + X_L\right)\left(Y_H + Y_L\right) - X_H Y_H - X_L Y_L\right) \tag{3.6}$$

となる．$X_H Y_L + X_L Y_H$ の計算に $X_H Y_H$ と $X_L Y_L$ を利用することにより，変形前は 2 回必要であった乗算が，変形後は $(X_H + X_L)(Y_H + Y_L)$ の 1 回のみとなっている．

$$P_1 = X_H Y_H \tag{3.7}$$

$$P_2 = X_L Y_L \tag{3.8}$$

$$P_3 = \left(X_H + X_L\right)\left(Y_H + Y_L\right) \tag{3.9}$$

として整理すると，

$$XY = 2^n P_1 + P_2 + 2^{\frac{1}{2}n}\left(P_3 - P_1 - P_2\right) \tag{3.10}$$

となる．式 (3.4) で示されるように，通常の乗算では $\frac{1}{2}n$ ビット $\times \frac{1}{2}n$ ビットの乗算が 4 回と加算が 2 回必要となる．ここで，式 (3.4) では数式上では 3 回の加算が現れているが，$2^n X_H Y_H + X_L Y_L$ はビットの重なりがなく実際には加算を行う必要が

ないため，必要な加算は2回となる．Karatsubaアルゴリズムを用いて乗算を行う場合，P_1，P_2 はそれぞれ1回の乗算で，P_3 は2回の加算と1回の乗算で求められる．2^nP_1 と P_2 は n ビットずれており重なる部分がないため，$2^nP_1+P_2$ は加算を行う必要がない．そのため，P_1，P_2，P_3 から XY を求めるためには，3回の加減算が必要となる．これらを合計すると，Karatsubaアルゴリズムを用いることで n ビット乗算は約 $\frac{1}{2}n$ ビットの乗算3回と加算5回で実行できる．Karatsubaアルゴリズムは再帰的に適用することができる．アルゴリズム中に現れる3回の乗算をさらにKaratsubaアルゴリズムを用いて計算することができる．

3.3　Karatsubaアルゴリズムに基づく乗算器の構成法

Karatsubaアルゴリズムを用いると，少ない演算量で乗算を実行できる．したがって，Karatsubaアルゴリズムを用いて並列乗算器を構成した場合，少ないセル数で構成できると考えられる．本章ではKaratsubaアルゴリズムに基づく乗算器をKaratsuba乗算器と呼ぶ．3.3.1 ，3.3.2 ではKaratsuba乗算器の基本構成について述べる．3.3.3 ではKaratsuba乗算器の小面積化手法，3.3.4 では用いるセル数を増やさずに高速化する手法を提案する．

3.3.1　Karatsuba乗算器の基本構成

本節では，式 (3.10) を1回だけ用いた場合のKaratsuba乗算器の構成について述べる．式 (3.10) に基づく符号なし整数乗算器の構成を図3.1に示す．本論文では，図3.1の構成の乗算器を基本Karatsuba乗算器と呼ぶ．図中のCPAは桁上げ伝搬加算器 (CarryPropagateAdder) を示す．基本Karatsuba乗算器では2つの桁上げ伝搬加算器と3つの乗算器を用いて P_1，P_2，P_3 を算出する．$-P_1$，$-P_2$ を得るためには，P_1，P_2 をビット反転し，さらにそれぞれの最下位桁に1を加算する．これらを加算して積を求める．本章では，P_1，$-P_1$，P_2，$-P_2$，P_3 から最終的な乗算結果を求めるための回路を加算モジュール (Addition Module) と呼ぶ．

Karatsubaアルゴリズムを適用することで小面積となる範囲を見積もる．並列乗算器は，主に全加算器，半加算器，AND回路によって構成される．このうち，並列乗算器の面積の多くは全加算器に占められる．ここでは全加算器数を概算し，比較することによっておおまかな有効範囲の見積もりとする．簡単のために半加算器，AND回路の個数については考えないこととする．

まず，P_1，P_2，P_3 を算出するために必要な全加算器数を求める．P_1，P_2 はそれぞれ 1 個の $\frac{1}{2}n$ ビット乗算器，P_3 は 2 個の $\frac{1}{2}n$ ビット加算器と 1 個の $\frac{1}{2}n+1$ ビット乗算器が必要である．$\frac{1}{2}n$ ビット加算器は $\frac{1}{2}n-1$ 個の全加算器で構成できる．$\frac{1}{2}n$ ビット乗算器，$\frac{1}{2}n+1$ ビット乗算器に必要な全加算器数をそれぞれ $C_{mult}\left(\frac{1}{2}n\right)$，$C_{mult}\left(\frac{1}{2}n+1\right)$ とする．これらを合計すると $2C_{mult}\left(\frac{1}{2}n\right)+C_{mult}\left(\frac{1}{2}n+1\right)+n-2$ となる．

次に，P_1，P_2，$-P_1$，$-P_2$，P_3 を加算するために必要な全加算器数を見積もる．P_1，P_2，$-P_1$，$-P_2$ はそれぞれ n ビット，P_3 は $n+2$ ビットである．P_1，P_2，$-P_1$，$-P_2$，P_3 を合わせると $5n+2$ ビットとなる．これを全加算器を用いて $2n$ ビットまで加算する．全加算器は 1 個につき，3 ビットを入力し，2 ビットが出力されるため，全加算器を 1 個用いることで 1 ビット減らすことができる．そのため，加算によって減るビット数と必要な全加算器数が一致する．$5n+2$ ビットを $2n$ ビットまで加算するために必要な全加算器数は $3n+2$ 個となる．8 ビット Karatsuba 乗算器における加算モジュールの入力を図 3.2 に示す．P_1 の下位から i ビット目を $P_1[i]$ と表す．P_2，P_3 についても同様である．

以上より，Karatsuba アルゴリズムに基づく乗算器に必要な全加算器数を $C_K(n)$ とすると，

$$\begin{aligned} C_K(n) &= 2C_{mult}\left(\frac{1}{2}n\right) && (3.11)\\ &\quad +C_{mult}\left(\frac{1}{2}n+1\right)+4n && (3.12)\end{aligned}$$

となる．Karatsuba 乗算器の内部で配列型乗算器や Wallace 乗算器を用いた場合を考える．n ビットの配列型乗算器や Wallace 乗算器では，部分積が n^2 ビット生成され，これを加算して $2n$ ビットの出力を得る．n^2 ビットを $2n$ ビットまで減らすため，n ビットの配列型乗算器，Wallace 乗算器には n^2-2n 個の全加算器が必要である．そのため，

$$\begin{aligned} C_{mult}\left(\frac{1}{2}n\right) &= \frac{1}{4}n^2-n && (3.13)\\ C_{mult}\left(\frac{1}{2}n+1\right) &= \frac{1}{4}n^2-1 && (3.14)\end{aligned}$$

となる．このとき Karatsuba 乗算器の全加算器数は

$$\begin{aligned} C_K(n) &= 2\left(\frac{1}{4}n^2-n\right)+\left(\frac{1}{4}n^2-1\right)+4n && (3.15)\\ &= \frac{3}{4}n^2+2n-1 && (3.16)\end{aligned}$$

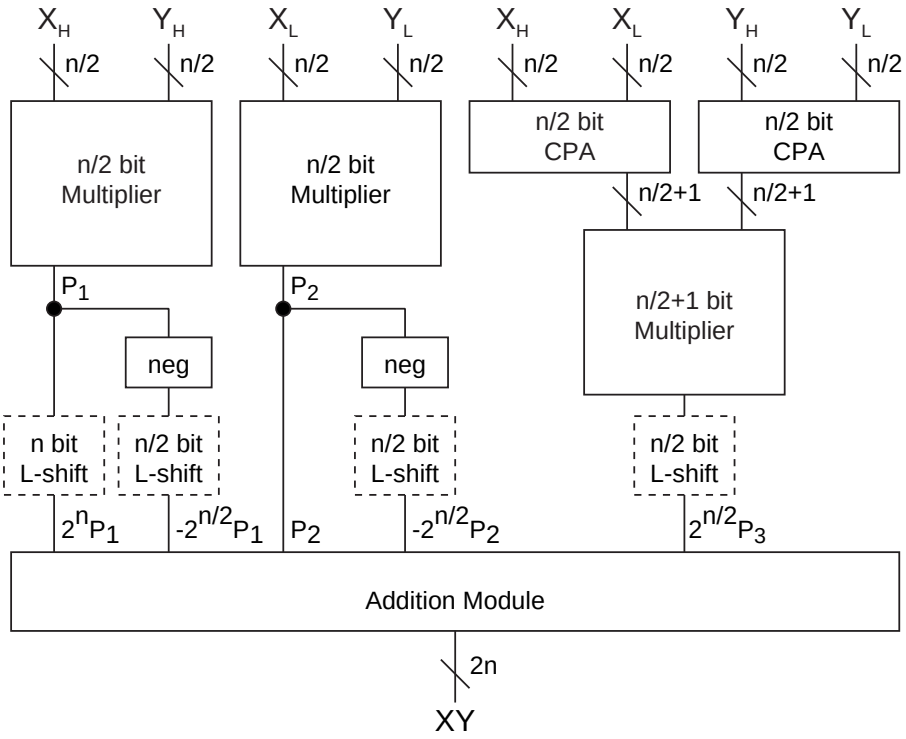


図 3.1: 基本 Karatsuba 乗算器の構成

となる．配列型乗算器と Karatsuba 乗算器の全加算器数を比較すると，18 ビット以上で配列型乗算器よりも Karatsuba 乗算器の全加算器数が少なくなる．そのため，入力幅が 18 ビット周辺で配列型乗算器よりも Karatsuba 乗算器のほうが小面積となると予想される．

3.3.2 Karatsubaアルゴリズムの再帰的適用による小面積化

本節では乗算器に Karatsuba アルゴリズムを再帰的に適用した場合を考える．

Karatsuba アルゴリズムを用いた乗算器は $\frac{1}{2}n$ ビット乗算器 3 個と式 (3.10) の 5 回の加算を行う加算器が必要となる．一方，配列型乗算器は $\frac{1}{2}n$ ビット乗算器約 4 個分のに相当する．配列型乗算器などと比較すると，$\frac{1}{2}n$ ビット乗算器約 1 個分のセルが削減でき，一方で加算器の追加が必要となる．n がある値よりも大きい場合には，追加する加算器よりも削減できる $\frac{1}{2}n$ ビット乗算器のセル数のほうが多くなる．このとき，Karatsuba 乗算器は配列型乗算器よりも少ないセル数で構成できる．一方，n がある値よりも小さい場合には，Karatsuba 乗算器が配列型乗算器よりも多くのセルを必要とするようになる．

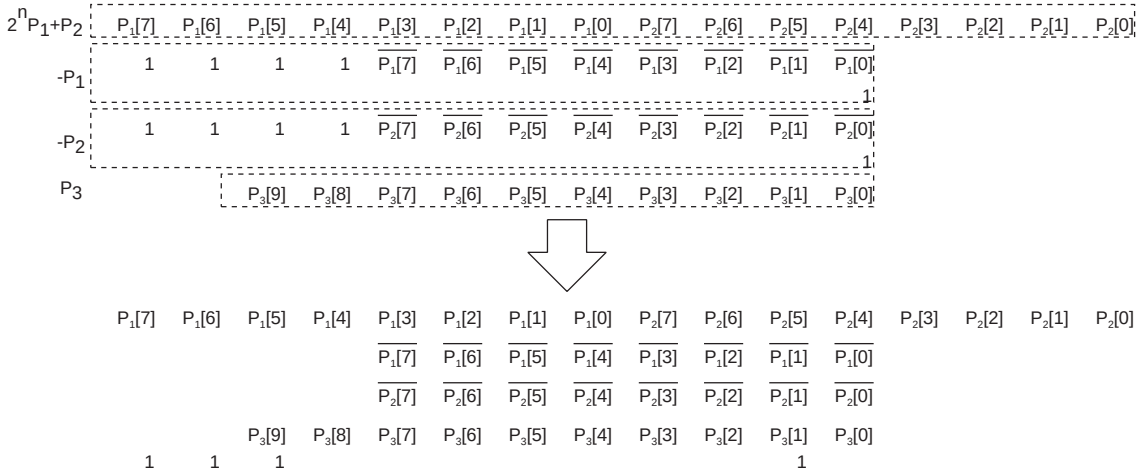


図 3.2: Karatsuba 乗算器の加算モジュールの入力

Karatsuba アルゴリズムを再帰的に適用し，乗算器を分解していくと，内部で用いる乗算器のサイズは小さくなっていく．そのため，再帰的に適用することによってセル数を少なくすることのできる最小のサイズがあると考えられる．

Karatsuba アルゴリズムを 2 回適用した場合の全加算器数を求める．n は 4 の倍数とする．式 (3.16) と同様に考えると，

$$C_K\left(\frac{1}{2}n\right) = \frac{3}{16}n^2 + n - 1 \tag{3.17}$$

$$C_K\left(\frac{1}{2}n+1\right) = \frac{3}{16}n^2 + 2n + 5 \tag{3.18}$$

となる．これらを用いると，Karatsuba アルゴリズムを 2 段適用した乗算器に必要な全加算器数 C_{K2} は

$$C_{K2}(n) = 2\left(\frac{3}{16}n^2 + n - 1\right) + \left(\frac{3}{16}n^2 + 2n + 5\right) + 4n \tag{3.19}$$

$$= \frac{9}{16}n^2 + 8n + 3 \tag{3.20}$$

となる．

式 (3.16) と式 (3.20) を比較すると，36 ビット以上で 1 段よりも 2 段の Karatsuba 乗算器の全加算器数が少なくなる．そのため，36 ビット前後で 2 段の Karatsuba 乗算器のほうが小面積となると予想される．

式 (3.16)，式 (3.20) から，いくつかの n について得られた必要な全加算器数を表 3.1 に示す．16 ビット以下では配列型乗算器，32 ビットでは 1 段の Karatsuba 乗算器，64 ビット以上では 2 段の Karatsuba 乗算器が最も全加算器数が少なくなっている．

表 3.1: 配列型乗算器，Karatsuba 乗算器の全加算器数

	配列型乗算器	Karatsuba 乗算器	
		1 段	2 段
8bit	48	63	103
16bit	224	223	275
32bit	960	831	835
64bit	3968	3199	2819
128bit	16128	12543	10243

3.3.3　計算順序の変更によるセル数の削減

本節では，計算の順序を変更することによってセル数を削減する手法を提案する．3.3.1 の構成では，P_1，P_2 を分岐して，適切にシフト，反転した上で加算モジュールでそれぞれを加算している．P_1，P_2 それぞれを分岐するのではなく，加算してから分岐することによって加算モジュールの入力ビット数を減らすことができる．これは，式 (3.10) に以下の変形を行うことに相当する．

$$\begin{aligned} XY &= 2^n P_1 + P_2 + 2^{\frac{1}{2}n}\left(P_3 - P_1 - P_2\right) && (3.21)\\ &= -2^{\frac{1}{2}n}\left(-2^{\frac{1}{2}n}+1\right)P_1 + \left(-2^{\frac{1}{2}n}+1\right)P_2 + 2^{\frac{1}{2}n}P_3 && (3.22)\\ &= \left(-2^{\frac{1}{2}n}P_1 + P_2\right)\left(-2^{\frac{1}{2}n}+1\right) + 2^{\frac{1}{2}n}P_3 && (3.23) \end{aligned}$$

ここで，

$$P_4 = -2^{\frac{1}{2}n}P_1 + P_2 \tag{3.24}$$

とすると

$$XY = \left(-2^{\frac{1}{2}n}+1\right)P_4 + 2^{\frac{1}{2}n}P_3 \tag{3.25}$$

となる．

ソフトウェアで式 (3.25) を用いた場合，乗算 3 回と加算 5 回が必要となり，演算量は式 (3.10) と変わらない．一方，組合せ回路では，式 (3.25) に基づいて回路を構成することにより，必要な全加算器数を削減できる．式 (3.25) を用いた Karatsuba 乗算器の構成を図 3.3 に示す．この構成における必要な全加算器数を求める．P_1，P_2，P_3 の算出に必要な全加算器数は 3.3.2 と同様である．P_1，P_2 はそれぞれ n ビッ

表 3.2: 計算順序を変更した Karatsuba 乗算器の全加算器数

	1 段	2 段
8bit	59	101
16bit	215	267
32bit	815	815
64bit	3167	2775
128bit	12479	10151

トであり $\frac{1}{2}n$ ビットが重なっているため，P_4 を求めるためには全加算器 $\frac{1}{2}n$ 個が必要となる．P_3 は $n+2$ ビットであり，P_4 は $\frac{3}{2}n$ ビットとなるため，P_4，$-P_4$，P_3 を合わせて $4n+2$ ビットとなる．本手法を適用した場合の加算モジュールへの入力を図 3.4 に示す．P_4 は 2 の補数表現で表され，最上位ビットをそろえるために $\frac{1}{2}n$ ビットの拡張が必要となる．これらを合計すると $\frac{9}{2}n+2$ ビットとなり，$2n$ ビットまで加算するため $\frac{5}{2}n+2$ 個の全加算器が必要となる．以上より，Karatsuba アルゴリズムに基づく乗算器の全加算器数は

$$C_{K'}(n) = \frac{3}{4}n^2 + \frac{3}{2}n - 1 \tag{3.26}$$

となる．3.3.2 の構成と比較すると，加算モジュールで必要な全加算器数が n 個削減できる一方，P_4 の算出に $\frac{1}{2}n$ 個の全加算器が必要となるため，これを差引いて $\frac{1}{2}n$ 個の全加算器が削減できる．Karatsuba アルゴリズムを 2 段適用した場合の全加算器数は，3.3.2 と同様に考えると，

$$C_{K2'}(n) = \frac{9}{16}n^2 + \frac{29}{4}n + 7 \tag{3.27}$$

となる．

式 (3.26)，式 (3.27) より得られた必要な全加算器数を表 3.2 に示す．必要な全加算器数は表 3.1 の Karatsuba 乗算器の全加算器数よりも少なくなっている．また，表 3.1 と同様，64 ビット以上で 2 段の Karatsuba 乗算器の全加算器数が少なくなっており，32 ビット以上で再帰的に Karatsuba アルゴリズムを適用すると全加算器数を減らすことができる．

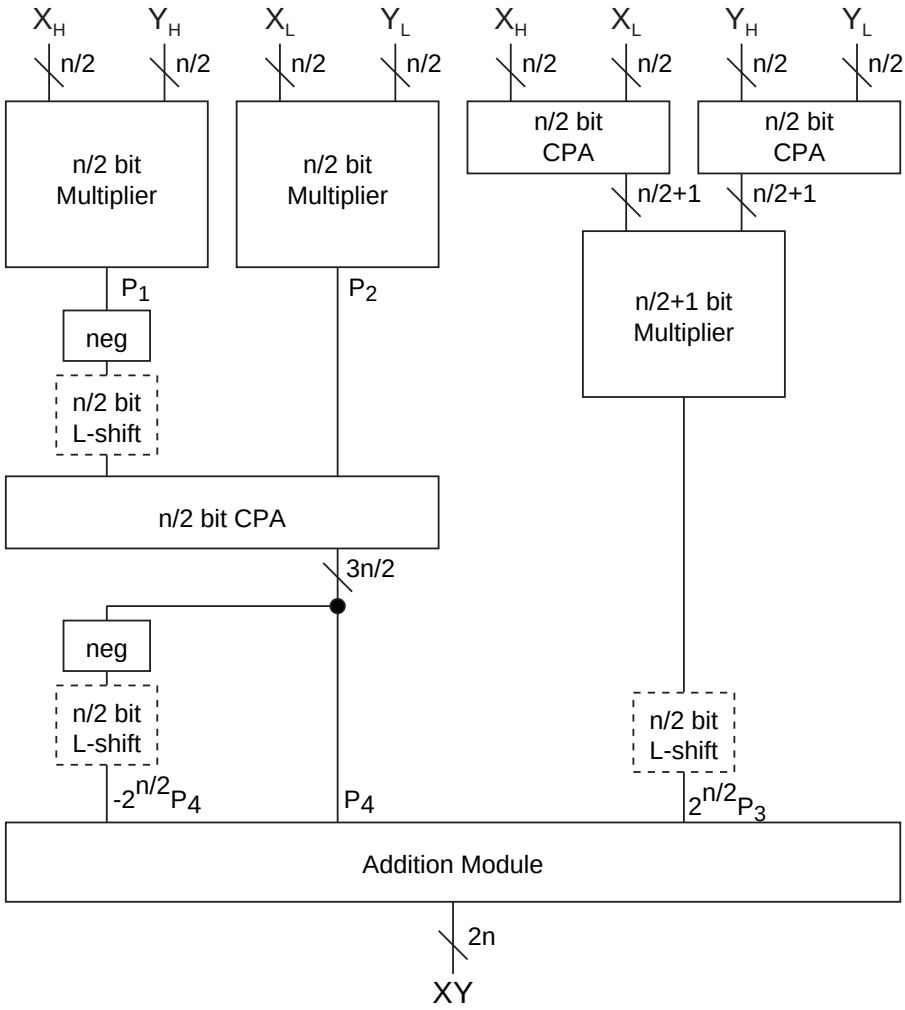


図 3.3: P_4 を用いた Karatsuba 乗算器の構成

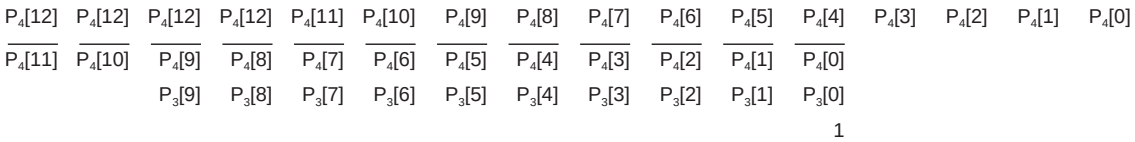


図 3.4: P_4 を用いた加算モジュールの入力

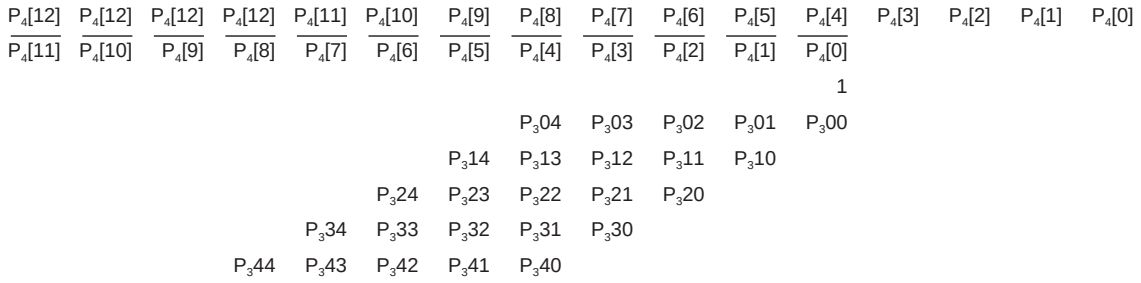


図 3.5: P_3 の部分積を用いた加算モジュールの入力

3.3.4 桁上げ伝搬加算の削減による高速化

本節では，用いる全加算器数を増やさずに Karatsuba 乗算器を高速化する手法を提案する． 図 3.3 の構成では，7 個の桁上げ伝搬加算器を用いている． すなわち，X_H と X_L，Y_H と Y_L の加算を行う加算器，3 個の乗算の最終加算器，$-2^{\frac{n}{2}}P_1 + P_2$ を行う加算器，加算モジュールの最後に現れる桁上げ伝搬加算である． 桁上げ伝搬加算は大きな遅延時間が必要となるため，これを減らすことによって高速化が期待できる．

まず，P_3 の計算に着目する．図 3.3 では，乗算器を用いて P_3 を求め，加算モジュールで P_4，$-2^{\frac{1}{2}n}P_4$ と加算している．P_3 の部分積を加算モジュールの入力とし，P_4，$-2^{\frac{1}{2}n}P_4$ と加算するように変更する．P_3 の部分積を用いた加算モジュールの入力を図 3.5 に示す．この変更によって P_3 を求める乗算器の最終加算が必要なくなる．加算するビット数は変化していないため，必要な全加算器数は変化しない．

次に，P_4 の計算に着目する．P_4 は $-2^{\frac{1}{2}n}P_1$ と P_2 を加算することによって得られる．P_4 を算出するためには，P_1 を求めるための乗算の最終加算，P_2 を求めるための乗算の最終加算，$-P_1$ と P_2 の加算の 3 つの桁上げ伝搬加算器を用いている．このうち，P_1，P_2 を求める乗算器の最終加算の桁上げ伝搬加算器を削除できる．そのためには，$-P_1$，P_2 を部分積を統合して加算し，P_4 を算出する．その場合，$-P_1$ と P_2 の部分積が必要となる．$-P_4$ の各部分積は負数となるため，符号拡張が必要である．この符号拡張は 1 ビットで済む．図 3.6 に 8 ビット Karatsuba 乗算器において，P_4 を算出するために加算する $-P_1$ と P_2 の部分積を示す．ここで P_1ij は P_1 の i 個目の部分積の最下位から j ビット目を表す．P_2ij についても同様である．$-P_1$ を扱うために，P_1 の部分積の値を反転し，最下位桁に 1 を追加している．また，$-P_1$ の各部分積の符号拡張は，事前に足し合わせることにより 1 ビットで

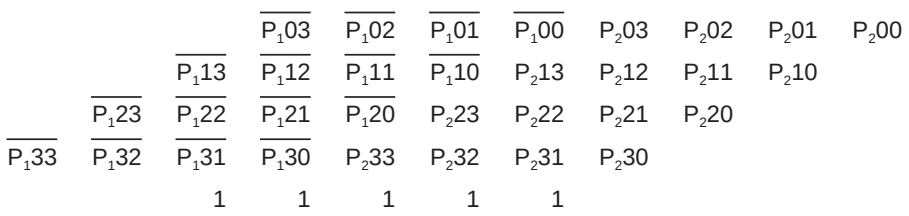


図 3.6: $-P_1$，P_2 の部分積

済む．図中の 5 ビットの 1 のうち，最上位の 1 ビットが符号拡張ビットであり，下位 4 ビットが部分積の 2 の補数表現の補正ビットである．これらの手法を用いることで図 3.3 で 7 個用いていた桁上げ伝播加算器は 4 個まで減らすことができる．Karatsuba アルゴリズムを 2 段適用した場合には，21 個の桁上げ伝搬加算器を 13 個まで減らすことができる．

本論文では，基本 Karatsuba 乗算器に 3.3.3 節と本節の手法を適用した構成を改良 Karatsuba 乗算器と呼ぶ．改良 Karatsuba 乗算器の構成を図 3.7 に示す．図中の PP は部分積 (Partial Product)，PPG は部分積生成回路 (Partial Product Generator) を示す．

3.4　評価

3.4.1　評価環境

本章では Karatsuba 乗算器の評価を行う．比較には以下の乗算器を用いる．

- 配列型乗算器
- Wallace 乗算器
- 基本 Karatsuba 乗算器
- 改良 Karatsuba 乗算器

基本 Karatsuba 乗算器では図 3.1 の構成を用いている．改良 Karatsuba 乗算器は基本 Karatsuba 乗算器に対して 3.3.3，3.3.4 の手法を適用した乗算器である．改良 Karatsuba 乗算器でも内部の部分積加算には Wallace 木を用いている．乗算器の設計には以下のプロセスのセルライブラリを用いた．

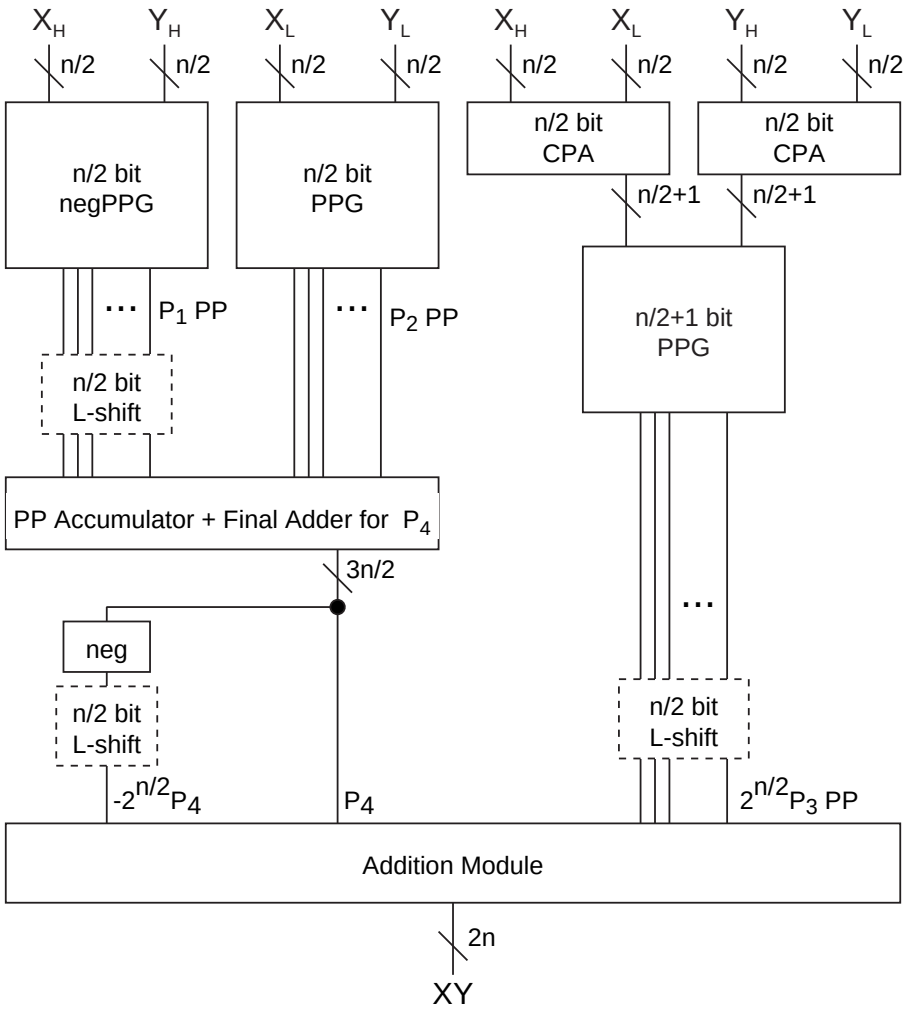


図 3.7: 改良 Karatsuba 乗算器の構成

- ROHM0.35μm/メタル 3 層
- ROHM0.18μm/メタル 5 層
- STARC90nm/メタル 6 層

論理最適化にはSynopsys 社Design Compilerを用いた．レイアウトツールはROHM0.35μm, STARC90nm では Synopsys 社 Astro，ROHM0.18μm では Cadence 社 SOC Encounter を用いた．

32 ビット，64 ビット乗算器を比較し，Karatsuba 乗算器を評価した．32 ビットでは 1 回，64 ビットでは 2 回 Karatsuba アルゴリズムを適用する．すなわち，64 ビット基本 Karatsuba 乗算器の内部では 32 ビット，33 ビットの基本 Karatsuba 乗算器を用いている．さらに，それぞれの内部では 16 ビット，17 ビット，18 ビットの Wallace 乗算器を用いている．それぞれの乗算器は以下の構成をとる．

- 回路面積が最小となる構成
- 遅延時間と回路面積の積 ($D \times A$) が最小となる構成

回路面積が最小となる構成では，桁上げ伝搬加算に順次桁上げ加算器を用いた．$D \times A$ が最小となる構成では，桁上げ伝搬加算器に Synopsys 社の DesignWare を用い，乗算器の $D \times A$ が最小となるように最適化を行った．最適化を行うにあたり，制約条件として回路面積を最小とし，遅延制約を段階的に変化させた．

配置配線設計では，全ての乗算器で回路面積に対してセルの面積の合計が 95% 以上となった．遅延時間はバックアノテーションを行い，論理合成ツールによって求めた．回路面積が最小となる構成では，全加算器数は表 3.1，表 3.2 とほぼ一致した．回路面積が最小となる構成の乗算器の評価を図 3.8 に，$D \times A$ が最小となる構成の乗算器の評価を図 3.9 に示す．それぞれ，配列型乗算器の回路面積と遅延時間を 1 として正規化し，棒グラフで回路面積、プロットで遅延時間を表す．評価値についてはそれぞれ表 3.3，表 3.4 に示す．

3.4.2　Karatsuba 乗算器の評価

まず，基本 Karatsuba 乗算器と改良 Karatsuba 乗算器を比較することによって，3.3.3 節と 3.3.4 節で提案した構成法の効果を評価する．改良 Karatsuba 乗算器を基本 Karatsuba 乗算器と比較すると，回路面積が最小となる構成では，32 ビットでは最

大約 4%，64 ビットでは最大約 2%小面積となっている．遅延時間は ROHM0.35μm，ROHM0.18μm で改良 Karatsuba 乗算器の方が大きくなっている．これは P_4 を用いることで，基本 Karatsuba 乗算器よりも桁上げ伝搬加算器の遅延時間が大きくなったためであると考えられる．$D \times A$ が最小となる構成では，どの評価条件においても改良 Karatsuba 乗算器は基本 Karatsuba 乗算器よりも小面積かつ高速となっており，最適化によって高速化しやすくなったことがわかる．以上の評価より，3.3.3 節，3.3.4 節の構成法は Karatsuba 乗算器の改善に有効であることが示された．

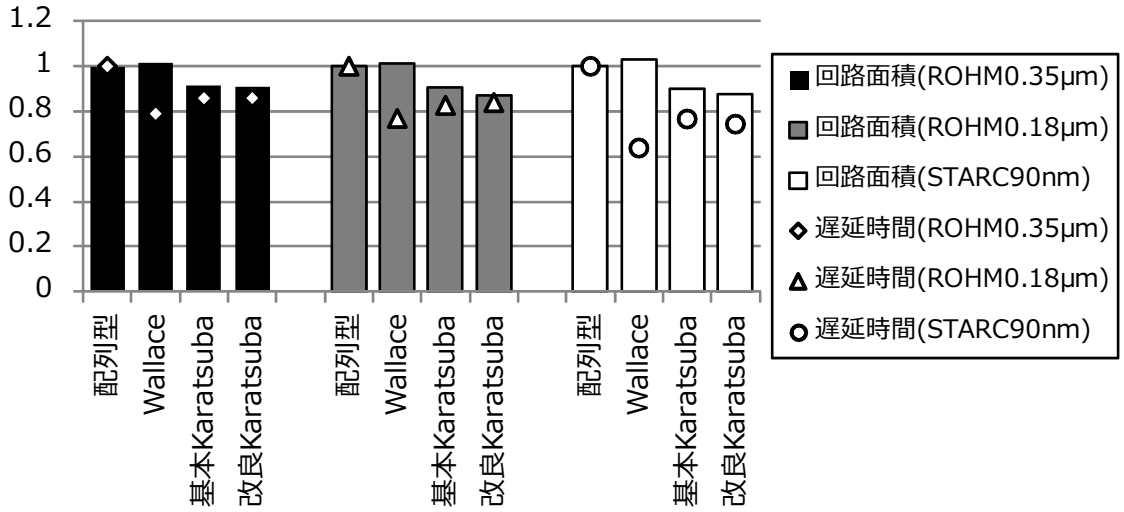
次に既存の並列乗算器の構成法と改良 Karatsuba 乗算器を比較することによって，Karatsuba 乗算器の有用性を示す．回路面積が最小となる構成では，32 ビットでは約 10%，64 ビットでは 20〜30%小面積となっている．$D \times A$ が最小となる構成で改良 Karatsuba 乗算器と Wallace 乗算器を比較すると，遅延時間では Wallace 乗算器が優れている．Karatsuba アルゴリズムを用いると，クリティカルパスに 2 回の桁上げ伝搬加算が含まれるため，Wallace 乗算器よりも遅延時間が大きくなったものと考えられる．

3.5　考察

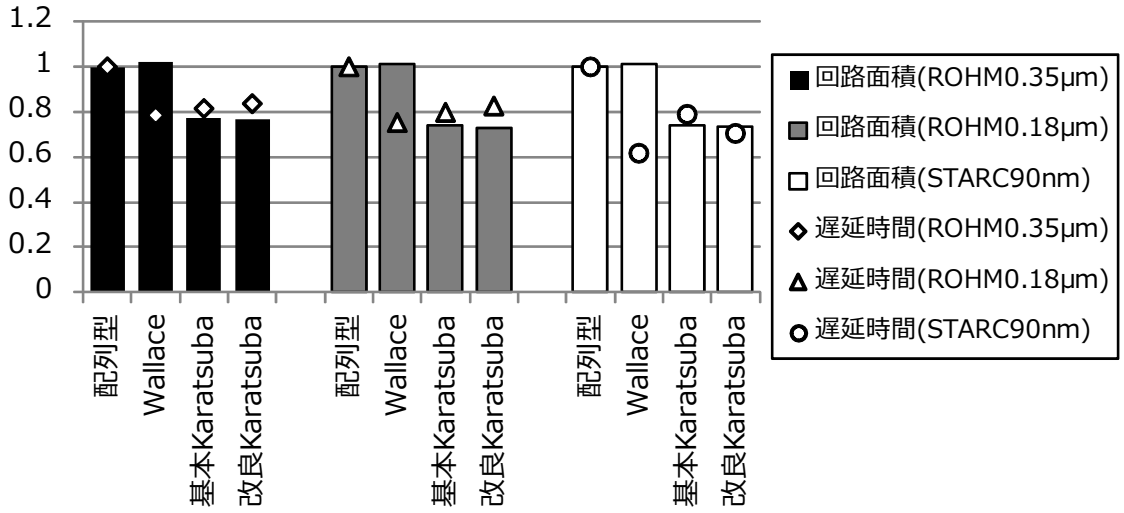
3.3.1 節，3.3.2 節では，全加算器数を用いて Karatsuba アルゴリズムを適用することによって小面積化が可能な入力のビット幅の範囲を見積もった．実際には，全加算器以外のセルや用いるセルライブラリによって有効な範囲は変動する．そこで本節では，0.35μm から 90nm のプロセスのセルライブラリを用いて並列乗算器を設計し，Karatsuba 乗算器が配列型乗算器よりも小面積となる入力ビット幅の範囲を調べる．また，Karatsuba アルゴリズムを再帰的に適用することで小面積となる範囲についても調べる．

図 3.10，図 3.11，図 3.12，にそれぞれ ROHM0.35μm，ROHM0.18μm，STARC90nm プロセス用セルライブラリを用いた配列型乗算器と Karatsuba 乗算器の素子面積の比較を示す．詳細な評価値については表 3.5 に示す．配列型乗算器，Karatsuba アルゴリズムを 1 回適用した構成，Karatsuba アルゴリズムを 2 回適用した構成の 3 種類の符号なし乗算器を設計した．Karatsuba 乗算器の構成は基本 Karatsuba 乗算器とした．素子面積が最小になる構成で評価を行う．

配列型乗算器と 1 段の Karatsuba 乗算器を比較すると，どのセルライブラリでも 16 ビットでは配列型乗算器の方が小面積で，18 ビットでは Karatsuba 乗算器の方

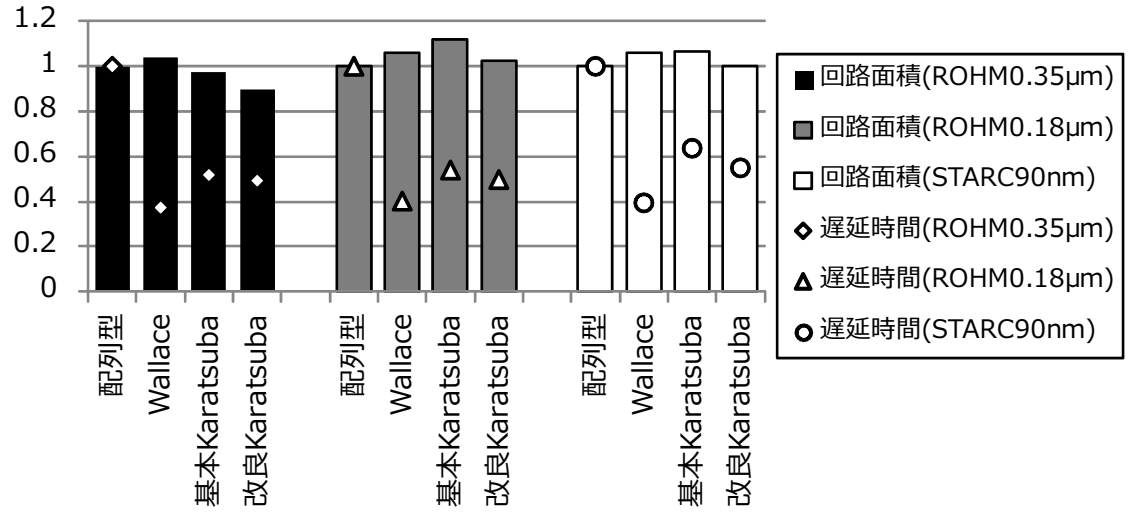


(a) 32 ビット Karatsuba 乗算器の評価

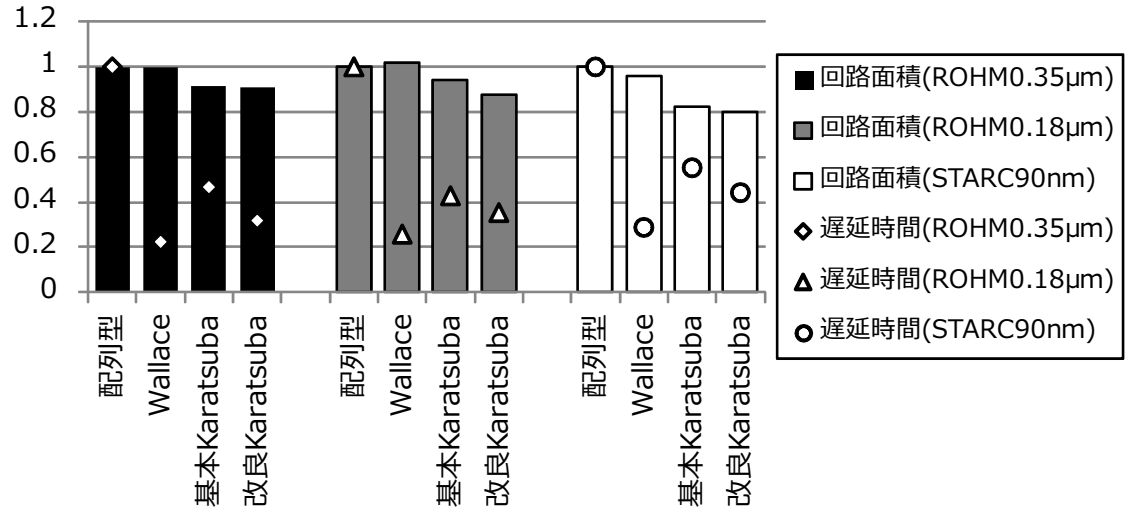


(b) 64 ビット Karatsuba 乗算器の評価

図 3.8: 回路面積が最小となる構成の Karatsuba 乗算器の評価



(a) 32 ビット Karatsuba 乗算器の評価



(b) 64 ビット Karatsuba 乗算器の評価

図 3.9: $D \times A$ が最小となる構成の Karatsuba 乗算器の評価

表 3.3: 面積が最小となる構成の Karatsuba 乗算器の評価

			回路面積 (μm^2)	遅延時間 (ns)
ROHM0.35μm	32bit	配列型乗算器	507299	32.32
		Wallace 乗算器	515861	25.52
		基本 Karatsuba 乗算器	465462	27.75
		改良 Karatsuba 乗算器	460385	27.76
	64bit	配列型乗算器	2066402	69.85
		Wallace 乗算器	2114112	54.83
		基本 Karatsuba 乗算器	1592628	56.93
		改良 Karatsuba 乗算器	1590741	58.43
ROHM0.18μm	32bit	配列型乗算器	101333.3	27.16
		Wallace 乗算器	102787.6	20.88
		基本 Karatsuba 乗算器	91841.9	22.45
		改良 Karatsuba 乗算器	88090.6	22.79
	64bit	配列型乗算器	410470.5	55.41
		Wallace 乗算器	415283.9	41.71
		基本 Karatsuba 乗算器	304565.2	44.22
		改良 Karatsuba 乗算器	298521.4	45.78
STARC90nm	32bit	配列型乗算器	23268.34	14.12
		Wallace 乗算器	23913.52	9.00
		基本 Karatsuba 乗算器	20978.59	10.82
		改良 Karatsuba 乗算器	20334.53	10.50
	64bit	配列型乗算器	95517.70	29.16
		Wallace 乗算器	96820.51	17.99
		基本 Karatsuba 乗算器	70649.60	23.02
		改良 Karatsuba 乗算器	70426.50	20.55

表 3.4: $D \times A$ が最小となる構成の Karatsuba 乗算器の評価

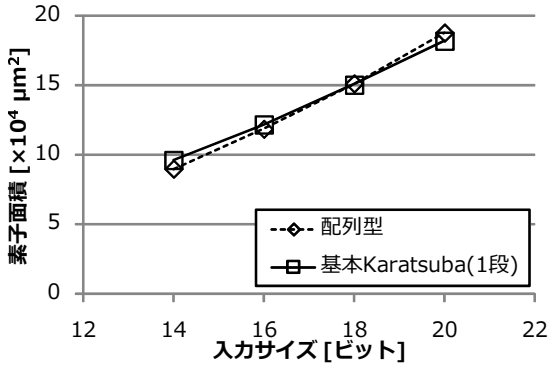
			回路面積 (μm^2)	遅延時間 (ns)
ROHM0.35μm	32bit	配列型乗算器	739596	17.76
		Wallace 乗算器	769566	6.63
		基本 Karatsuba 乗算器	721634	9.20
		改良 Karatsuba 乗算器	664218	8.75
	64bit	配列型乗算器	2324094	39.91
		Wallace 乗算器	2324094	8.93
		基本 Karatsuba 乗算器	2122848	18.63
		改良 Karatsuba 乗算器	2118480	12.69
ROHM0.18μm	32bit	配列型乗算器	113386.0	15.20
		Wallace 乗算器	120356.4	6.11
		基本 Karatsuba 乗算器	127050.3	8.20
		改良 Karatsuba 乗算器	116283.5	7.56
	64bit	配列型乗算器	443249.5	30.96
		Wallace 乗算器	451339.9	7.99
		基本 Karatsuba 乗算器	416122.4	13.25
		改良 Karatsuba 乗算器	387766.2	10.94
STARC90nm	32bit	配列型乗算器	32191.3	6.44
		Wallace 乗算器	34180.6	2.54
		基本 Karatsuba 乗算器	34232.4	4.10
		改良 Karatsuba 乗算器	32191.3	3.54
	64bit	配列型乗算器	126849.2	13.37
		Wallace 乗算器	121423.9	3.85
		基本 Karatsuba 乗算器	104367.5	7.39
		改良 Karatsuba 乗算器	101136.5	5.91

が小面積となっている．1 段と 2 段の Karatsuba 乗算器を比較すると，32〜36 ビットで 1 段よりも 2 段の Karatsuba 乗算器の方が小面積となっている．本論文で用いているセルライブラリでは，18 ビットよりも大きい入力ビット幅では Karatsuba アルゴリズムの適用によって並列乗算器の小面積化が可能であり，さらに 36 ビットよりも大きい並列乗算器では Karatsuba アルゴリズムを 2 回適用したほうが小面積となることがわかった．

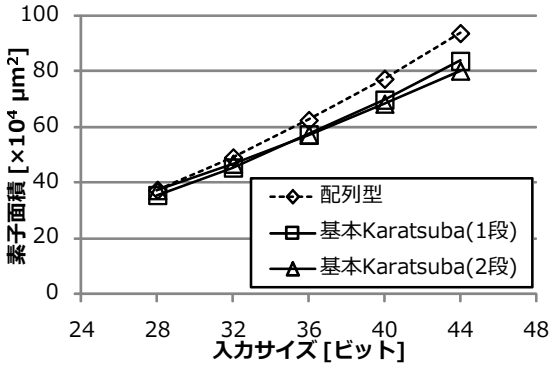
3.6　まとめ

本章では，近年の配線層数の増加により，配線量が多くても少ないセル数で演算回路を構成することによって，小面積な演算回路を構成できると考え，Karatsuba アルゴリズムに基づく並列乗算器を提案した．18 ビット以上の乗算器で Karatsuba アルゴリズムを適用することにより，既存の乗算器よりも小面積となることがわかった．36 ビット以上では Karatsuba アルゴリズムを再帰的に適用することにより，さらに小面積となった．本性での Karatsuba アルゴリズムに基づく並列乗算器の評価により，素子面積が小さくなることに重点を置いて構成法を選択することにより，小面積な乗算器が構成できることが確認できた．

本章では，並列乗算器に適した構成となるように Karatsuba アルゴリズムを変更する手法を提案した．Karatsuba アルゴリズムの計算順序を変更することで必要なセル数を削減し，Karatsuba 乗算器で使用される桁上げ伝搬加算を削減することで用いる全加算器数を増やさずに高速化した．提案手法を適用した改良 Karatsuba 乗算器は，基本 Karatsuba 乗算器と比べ回路面積，遅延時間の両方で改善がみられた．提案手法を用いることにより，Karatsuba 乗算器をさらに小面積化，高速化することが可能であることが確認できた．

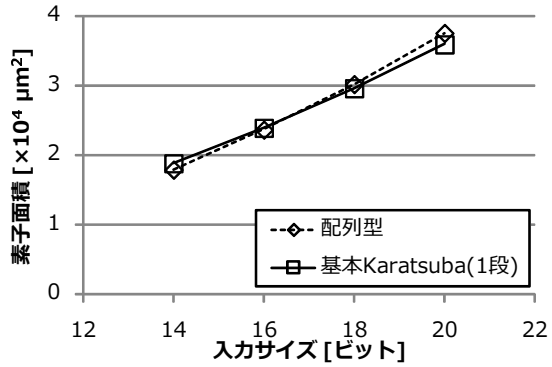


(a) 14 ビットから 20 ビット

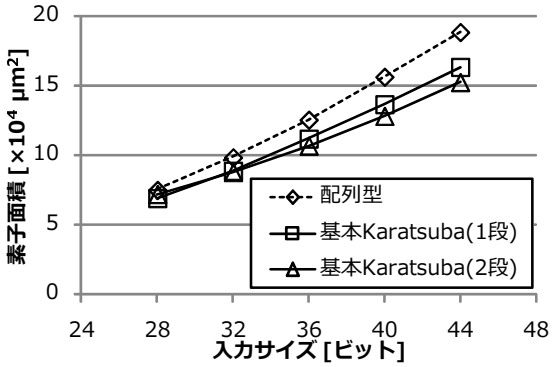


(b) 28 ビットから 44 ビット

図 3.10: ROHM0.35μm プロセスにおける入力ビット幅と素子面積の関係

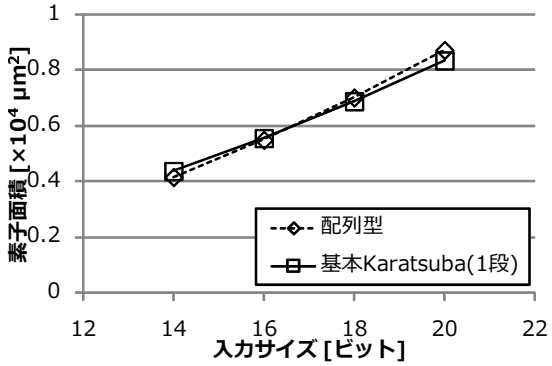


(a) 14 ビットから 20 ビット

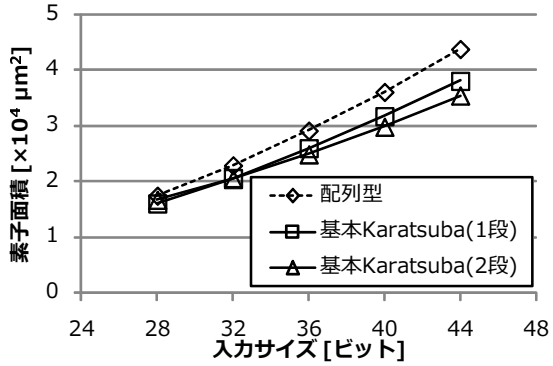


(b) 28 ビットから 44 ビット

図 3.11: ROHM0.18μm プロセスにおける入力ビット幅と素子面積の関係



(a) 14 ビットから 20 ビット



(b) 28 ビットから 44 ビット

図 3.12: STARC90nm プロセスにおける入力ビット幅と素子面積の関係

表 3.5: Karatsuba アルゴリズムが有効な入力ビット幅の評価 (μm^2)

		配列型乗算器	基本 Karatsuba 乗算器	
			1 段	2 段
ROHM0.35μm	14bit	90065	96306	-
	16bit	118775	121944	-
	18bit	151445	150552	-
	20bit	188075	182155	-
	28bit	374195	355612	372854
	32bit	491015	454330	468391
	36bit	625316	574043	572227
	40bit	774012	698681	684499
	44bit	938548	838579	803442
ROHM0.18μm	14bit	17938	18850	-
	16bit	23698	23957	-
	18bit	30259	29663	-
	20bit	37620	35978	-
	28bit	75063	69205	72099
	32bit	98584	88846	87904
	36bit	125644	112028	106974
	40bit	156603	136782	128395
	44bit	188762	163667	152722
STARC90nm	14bit	4160	4375	-
	16bit	5499	5562	-
	18bit	7024	6889	-
	20bit	8736	8356	-
	28bit	17446	16044	16708
	32bit	22919	20623	20445
	36bit	29161	25999	24884
	40bit	36127	31747	29854
	44bit	43838	38110	35483

第4章　オペランドの和を利用した小面積並列乗算器

4.1　はじめに

本章ではオペランドの和を利用した部分席生成法に基づく並列乗算器を提案する．オペランドの和を利用した部分積生成法は，CPP（Compound Partial Product）と呼ばれる新しい部分積を生成する．CPP を用いることで乗算全体の部分積のビット数を約半分に削減でき，部分積加算回路の面積を小さくすることができる．また，CPP の配置を工夫することで乗算を高速化する手法も提案する．この手法はオペランドの和の計算を複数に分割し，並列化することで高速化を図る．

評価結果より，CPP を用いた乗算器は従来の配列型乗算器よりも約 30%，基数 4 の Booth の方法を用いた乗算器よりも約 10%小面積となった．遅延時間はこれらの乗算器よりも小さくなった．

本章の構成は以下のとおりである．4.2 節でオペランドの和を利用した部分積生成法について述べる．4.3 節で乗算器の構成法について述べ，4.4 節で CPP を用いた乗算器の高速化手法について述べる．4.5 節で評価について述べ，4.6 節でまとめを行う．

4.2　オペランドの和を利用した部分積生成法

4.2.1　符号なし乗算

本節では，n ビット符号なし乗算におけるオペランドの和を利用した部分積生成法を説明する．本節では，X，Y の下位 i ビット目までを X_i，Y_i と表す．X_i，Y_i のビット表現はそれぞれ $[x_i x_{i-1} \cdots x_1 x_0]$，$[y_i y_{i-1} \cdots y_1 y_0]$ となる．X_i，Y_i を符号なし整数とした場合の値は，$\sum_{j=0}^{i} 2^j x_j$，$\sum_{j=0}^{i} 2^j y_j$ となる．

$X \times Y$ は X_{n-2} と Y_{n-2} を用いて次のように変形できる．

$$\begin{aligned} X \times Y &= \left(\sum_{i=0}^{n-1} 2^i x_i\right) \times \left(\sum_{i=0}^{n-1} 2^i y_i\right) && (4.1)\\ &= \left(2^{n-1}x_{n-1} + X_{n-2}\right) \times \left(2^{n-1}y_{n-1} + Y_{n-2}\right) && (4.2)\\ &= X_{n-2} \times Y_{n-2} + 2^{n-1}(2^{n-1}x_{n-1}y_{n-1} + x_{n-1}Y_{n-2} + y_{n-1}X_{n-2}) && (4.3) \end{aligned}$$

$X_{n-2} \times Y_{n-2}$ を繰り返し変形すると，次の式が得られる．

$$X \times Y = x_0 y_0 + \sum_{i=1}^{n-1} 2^i \left(2^i x_i y_i + x_i Y_{i-1} + y_i X_{i-1}\right) \quad (4.4)$$

本章では $2^i x_i y_i + x_i Y_{i-1} + y_i X_{i-1}$ を Compound Partial Product(CPP) と呼ぶこととする．$i = k$ の時の CPP を P_k と表す．CPP は乗数と被乗数の和を利用することにより，効率よく生成することができる．P_i の値は x_i と y_i の値の組み合わせによって，次のように 4 つに場合分けすることができる．

$$P_i = \begin{cases} 0 & \texttt{if } (x_i, y_i) = (0,0) \\ X_{i-1} & \texttt{if } (x_i, y_i) = (0,1) \\ Y_{i-1} & \texttt{if } (x_i, y_i) = (1,0) \\ 2^i + X_{i-1} + Y_{i-1} & \texttt{if } (x_i, y_i) = (1,1) \end{cases} \quad (4.5)$$

$(x_i, y_i) = (1,1)$ の場合には，$X_{i-1} + Y_{i-1}$ を計算する必要がある．ここで，オペランドの和に関する定義を行う．$S = X + Y$ とする．S は $n+1$ ビット符号なし整数で，ビット表現を $[s_n s_{n-1} s_{n-2} \cdots s_1 s_0]$ とする．また，S_i を $[s_i s_{i-1} \cdots s_1 s_0]$ とする．P_i は S_i を用いて次のように表現できる．

$$P_i = \begin{cases} [\,0 \;\; 0 \;\; 0 \;\; 0 \;\; \cdots 0 \;\; 0\,] & \texttt{if } (x_i, y_i) = (0,0) \\ [\,0 \;\; 0 \;\; x_{i-1}\, x_{i-2} \cdots x_1 \;\; x_0\,] & \texttt{if } (x_i, y_i) = (0,1) \\ [\,0 \;\; 0 \;\; y_{i-1}\, y_{i-2} \cdots y_1 \;\; y_0\,] & \texttt{if } (x_i, y_i) = (1,0) \\ [\,s_i \;\; \overline{s_i} \;\; s_{i-1}\, s_{i-2} \cdots s_1 \;\; s_0\,] & \texttt{if } (x_i, y_i) = (1,1) \end{cases} \quad (4.6)$$

ここで，$\overline{s_i}$ は s_i のビット否定を示す．$(x_i, y_i) = (1,1)$ の場合には，$2^i + X_i + Y_i$ を計算する必要がある．$X_i + Y_i = S_i$ であるので，

$$\begin{aligned} 2^i + X_i + Y_i &= 2^i + S_i && (4.7)\\ &= [0100\cdots00] + [0 s_i s_{i-1} s_{i-2} \cdots s_1 s_0] && (4.8)\\ &= [s_i \overline{s_i} s_{i-1} s_{i-2} \cdots s_1 s_0] && (4.9) \end{aligned}$$

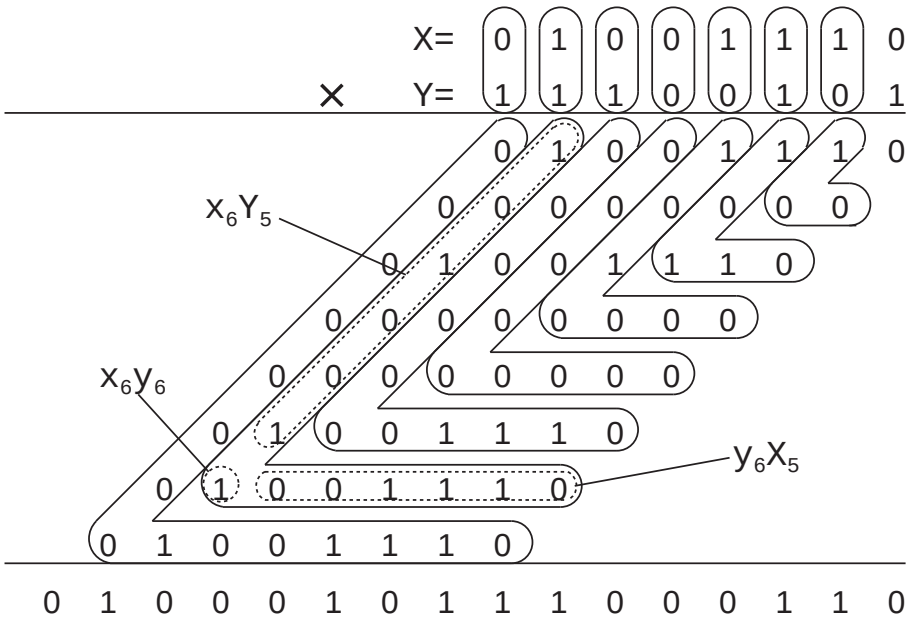


図 4.1: 8 ビット符号なし乗算の基本的な部分積

となり，式 (4.6) が得られる．

P_i は S の一部を用いて計算できるため，すべての i に対して S_i を計算する必要はなく，一度だけ S を計算すればよい．n ビット乗算において，2.2.1 節で示した基本的な部分積生成法で生成された部分積は乗算全体で n^2 ビットであるのに対し，オペランドの和を利用した部分積生成法によって生成された部分積は全体で $\frac{1}{2}n^2+\frac{3}{2}n-1$ ビットとなる．すなわち，提案した部分積生成法では部分積のビット数を約半分に削減することが可能である．最終的に，CPP を用いると乗算は以下のように計算できる．

$$X \times Y = x_0 y_0 + \sum_{i=1}^{n-1} 2^i P_i. \tag{4.10}$$

図 4.1，図 4.2 に X = [01001110]，Y = [11100101] とする 8 ビット符号無し乗算の例を示す．このとき，S = [100110011] となる．この例に対して，図 4.2 に CPP を，図 4.1 に基本的な部分積を示す．図 4.2 で楕円で囲まれた部分積ビットが CPP である．図 4.1 における “L 字型” の部分積ビットがそれぞれ CPP に対応する．例えば，$(x_3, y_3) = (1, 0)$ に対して P_3 は [00$y_2y_1y_0$]=[00101] となっている．P_4，P_5，P_6 はそれぞれ [000000]，[00$x_4x_3x_2x_1x_0$]=[0001110]，[$s_6\overline{s_6}s_5s_4s_3s_2s_1s_0$]=[01110011] となる．他の CPP も同様に計算できる．

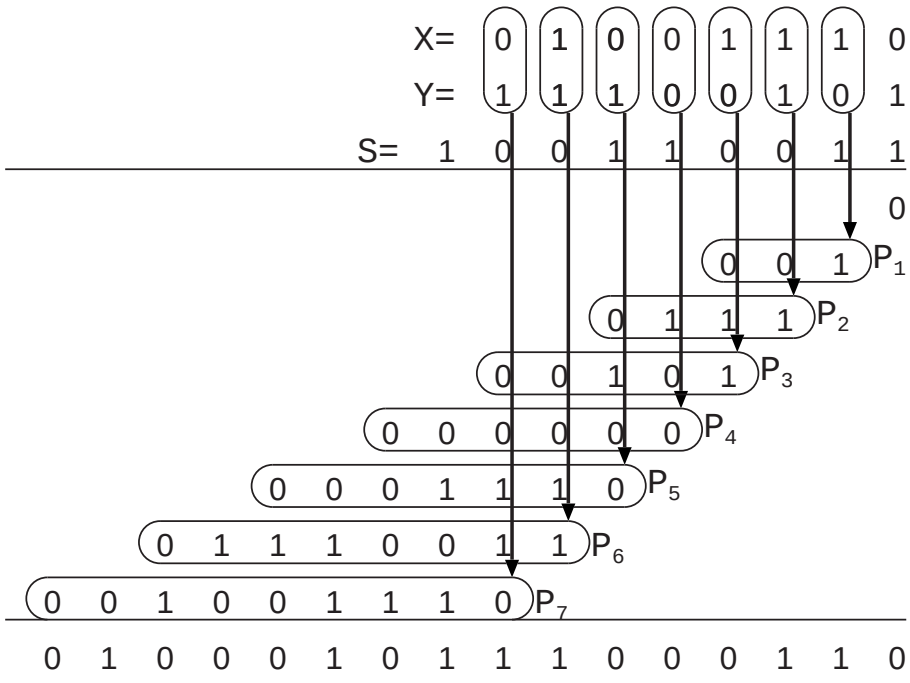


図 4.2: 8 ビット符号なし乗算の CPP

4.2.2　符号付き乗算

オペランドの和を利用した部分積生成法は少しの修正を加えることによって符号付き乗算に適用可能である．符号付き乗算では，X , Y は 2 の補数表現で表されるものとする．X，Y のビット表現をそれぞれ $[x_{n-1}x_{n-2}\cdots x_1x_0]$，$[y_{n-1}y_{n-2}\cdots y_1y_0]$ とすると，値は $-2^{n-1}x_{n-1}+\sum_{i=0}^{n-2}2^ix_i$，$-2^{n-1}y_{n-1}+\sum_{i=0}^{n-2}2^iy_i$ となる．$i \leq n-2$ に対しては，X_i，Y_i は符号なし整数であるとする．このとき，ビット表現 $[x_ix_{i-1}\cdots x_1x_0]$，$[y_iy_{i-1}\cdots y_1y_0]$ に対し，値は $\sum_{j=0}^{i}2^jx_j$，$\sum_{j=0}^{i}2^jy_j$ となる．S を $n+1$ ビット符号付き整数とし，2 の補数表現で表す．

符号付き乗算は以下の式で行われる．

$$X \times Y = \left(-2^{n-1}x_{n-1}+\sum_{i=0}^{n-2}2^ix_i\right)\left(-2^{n-1}y_{n-1}+\sum_{i=0}^{n-2}2^iy_i\right) \quad (4.11)$$

$$= X_{n-2}\times Y_{n-2}+2^{n-1}(2^{n-1}x_{n-1}y_{n-1}-x_{n-1}Y_{n-2}-y_{n-1}X_{n-2}) \quad (4.12)$$

x_0y_0 と P_1〜P_{n-2} に対応する CPP は符号なし乗算と同様に計算することができるため，$X_{n-2}\times Y_{n-2}$ も符号なし乗算と同様に計算することができる．P_{n-1} については符号なし乗算と異なる計算を行う．符号付き乗算における P_{n-1} は $2^{n-1}x_{n-1}y_{n-1}-x_{n-1}Y_{n-2}-y_{n-1}X_{n-2}$ と定義する．P_{n-1} の値は (x_{n-1},y_{n-1}) の組み合わせに従って，

4 個の候補から選択される．

$$P_{n-1} = \begin{cases} 0 & \texttt{if } (x_{n-1}, y_{n-1}) = (0,0) \\ -X_{n-2} & \texttt{if } (x_{n-1}, y_{n-1}) = (0,1) \\ -Y_{n-2} & \texttt{if } (x_{n-1}, y_{n-1}) = (1,0) \\ 2^{n-1} - (X_{n-2} + Y_{n-2}) & \texttt{if } (x_{n-1}, y_{n-1}) = (1,1). \end{cases} \tag{4.13}$$

P_{n-1} もオペランドの和 S を用いて効率良く計算することができる．$(x_{n-1}, y_{n-1}) = (1,1)$ の時，$X_{n-2} + Y_{n-2} = S_{n-1}$ と表現でき，S の表現の一部を用いる事ができる．ここで，$\overline{S_{n-1}}$ を S_{n-1} のビット否定とすると，$S_{n-1} = -2^n + \overline{S_{n-1}} + 1$ と計算することができる．$\overline{S_{n-1}}$ を n ビット符号なし整数として扱うと，$P_{n-1} = -2^n + 2^{n-1} + \overline{S_{n-1}} + 1$ となる．$P^*_{n-1} = P_{n-1} - 1$ とする．このとき，P^*_{n-1} は $n+1$ ビット 2 の補数表現で次のように表現できる．

$$P^*_{n-1} = \begin{cases} [\ 1 \quad 1 \quad 1 \quad 1 \quad \cdots \quad 1 \quad 1\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (0,0) \\ [\ 1 \quad 1 \quad \overline{x_{n-2}}\,\overline{x_{n-3}} \quad \cdots \quad \overline{x_1} \quad \overline{x_0}\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (0,1) \\ [\ 1 \quad 1 \quad \overline{y_{n-2}}\,\overline{y_{n-3}} \quad \cdots \quad \overline{y_1} \quad \overline{y_0}\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (1,0) \\ [s_{n-1}\, s_{n-1}\, \overline{s_{n-2}}\,\overline{s_{n-3}} \quad \cdots \quad \overline{s_1} \quad \overline{s_0}\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (1,1) \end{cases} \tag{4.14}$$

最終的に，符号付き乗算は以下のように行われる．

$$X \times Y = x_0 y_0 + \left(\sum_{i=1}^{n-2} 2^i P_i \right) + 2^{n-1} P_{n-1} \tag{4.15}$$

$$= x_0 y_0 + \left(\sum_{i=1}^{n-2} 2^i P_i \right) + 2^{n-1} P^*_{n-1} + 2^{n-1}. \tag{4.16}$$

CPP の総ビット数は $\frac{1}{2}n^2 + \frac{3}{2}n - 1$ ビットとなり，$x_0 y_0$ を加えると，$\frac{1}{2}n^2 + \frac{3}{2}n$ ビットとなる．

図 4.3 に $X = \texttt{[01001110]}$，$Y = \texttt{[11100101]}$ に対する 8 ビット符号付き乗算の例を示す．X，Y は 2 の補数表現で表される．$x_0 y_0$ および P_1 から P_6 は図 4.2 に示した符号なし乗算と同じである．この例では，$(x_7, y_7) = (0,1)$ であるため，P^*_7 は $[11\overline{x_6}\ \overline{x_5} \cdots \overline{x_1}\ \overline{x_0}] = [110110001]$ と表現される．

オペランドの和を利用した部分積生成法は，X と Y の乗算と加算を同時に行うため，$X \times Y$ と $X + Y$ が並列に出力される．一対のオペランドに対する和と積が必要とされるようなアプリケーションでは，提案した部分積生成法がより効率的に使用されると考えられる．

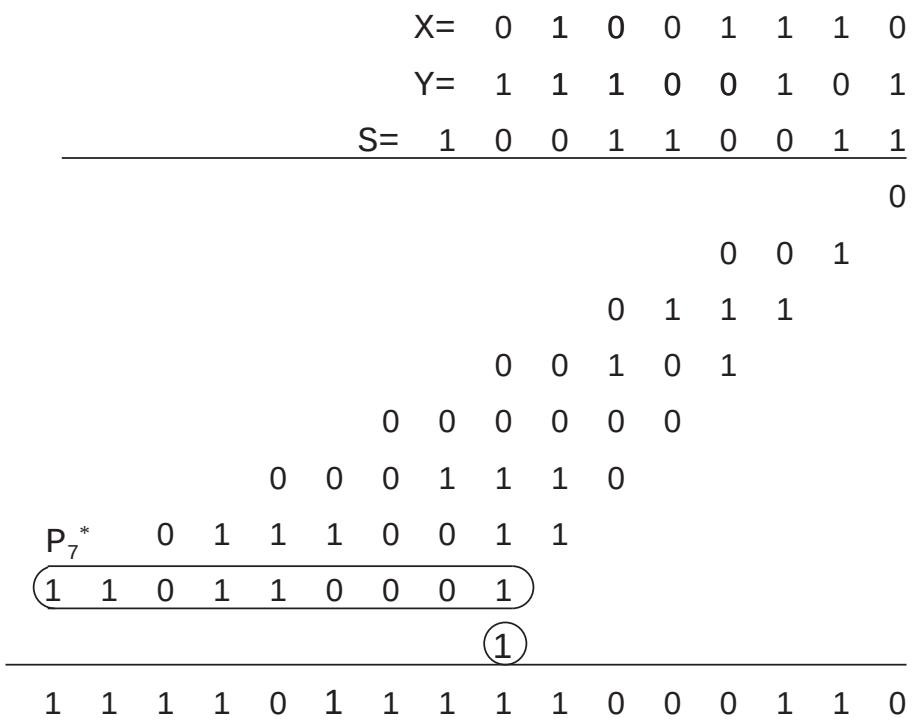


図 4.3: 符号付き乗算の例

4.3　CPP を用いた小面積乗算器

本節では，オペランドの和を利用した部分席生成法に基づく並列乗算器の構成について述べる．本論文では，オペランドの和を用いた部分積生成法を採用した並列乗算器を CPP 乗算器と呼ぶ．図 4.4 に CPP を用いた 8 ビット符号なし乗算器のブロック図を示す．CPP 乗算器はオペランド加算器 (Operand Adder)，オペランド符号化回路 (Operand Recoder)，CPP 生成器 (CPP Generator)，CPP 加算器 (CPP Compressor)，最終加算器 (Final Adder) で構成される．オペランド加算器はオペランドの和 S を計算するための桁上げ伝搬加算器である．オペランド符号化回路はオペランドの各ビットに対して $x_i \wedge y_i$，$\overline{x_i} \wedge y_i$，$x_i \wedge \overline{y_i}$ を計算する．CPP 生成器は X，Y，S と符号化されたオペランドの各ビットから CPP を生成する．CPP 生成器は CPP の各ビットを生成する $\frac{1}{2}n^2 + \frac{3}{2}n - 1$ 個のセレクタセルで構成される．各セレクタセルは (x_j, y_j, s_j) と $(x_i \wedge y_i, \overline{x_i} \wedge y_i, x_i \wedge \overline{y_i})$ から CPP ビット $p_{i,j}$ を計算する．CPP 加算器は桁上げ保存加算器を用いて CPP を 2 数まで加算する．CPP 加算器の構成としては，配列型，Wallace 木型等の既存の構成が利用できる．最後に CPP 加算器から得られた 2 数を最終加算器で加算し，X と Y の積が得られる．

CPP 生成器を構成するセレクタセルの詳細について述べる．i 番目の CPP P_i の下位から j ビット目を $p_{i,j}$ とする．符号なし乗算では，セレクタセルは次の式で CPP ビットを計算する．

$$p_{i,j} = ((\overline{x_i} \wedge y_i) \wedge x_j) \vee ((x_i \wedge \overline{y_i}) \wedge y_j) \vee ((x_i \wedge y_i) \wedge s_j) \tag{4.17}$$

$$p_{i,i} = (x_i \wedge y_i) \wedge \overline{s_i} \tag{4.18}$$

$$p_{i,i+1} = (x_i \wedge y_i) \wedge s_i \tag{4.19}$$

図 4.4 では，$p_{i,j}$，$p_{i,i}$，$p_{i,i+1}$ を計算するセルをそれぞれ白色，灰色，黒色で示している．符号付き乗算の場合は，P^*_{n-1} に対して変更が必要となる．P^*_{n-1} の下位 j ビット目を $p^*_{n-1,j}$ とする．P^*_{n-1} は次の式で計算される．

$$\begin{aligned} p^*_{n-1,j} &= \overline{((\overline{x_{n-1}} \wedge y_{n-1}) \wedge x_j)} \\ &\quad \wedge \overline{((x_{n-1} \wedge \overline{y_{n-1}}) \wedge y_j)} \wedge \overline{((x_{n-1} \wedge y_{n-1}) \wedge s_j)} \end{aligned} \tag{4.20}$$

$$p^*_{n-1,n-1} = s_{n-1} \vee \overline{(x_{n-1} \wedge y_{n-1})} \tag{4.21}$$

$$p^*_{n-1,n} = p_{n-1,n-1} \tag{4.22}$$

CPP 乗算器の遅延時間について述べる．基本的な部分積生成回路や基数 4 の Booth の方法による部分積生成は，乗算のオペランドのビット長によらず一定時間で部分積生成が完了する．一方，CPP の生成には桁上げ伝搬加算を行う必要がある．桁上げ伝搬加算の遅延時間はオペランドのビット長に依存するため，CPP の生成時間もオペランドのビット長に依存する．一般的にオペランドの和 S を構成するビットは，最下位ビットから最上位ビットに向かって順番に計算が完了する．S を構成するビットは $s_0, s_1, \cdots, s_n$ の順に計算が完了する．各 P_i の下位から j ビット目の $p_{i,j}$ の値は，s_j に依存して決定される．そのため，P_i を構成する各ビットの値は $p_{i,0}, p_{i,1}, \cdots, p_{i,i+1}$ の順に決定される．図 4.2 と図 4.3 の例では，各 CPP の右側から左側に向かって順番に値が確定する．一方，配列型の CPP 加算器を用いた場合，図の上側から下側に向かって加算が実行される．このままの回路構成では，CPP を構成する各ビットの値が確定する順序と CPP が加算される順序が一致しないため，無駄な遅延時間が生じると考えられる．

そこで，値の確定する順序と加算が実行される順序が一致するように CPP の配置を変更し，オペランド加算による乗算器の遅延時間の増加を最小限に抑える．CPP を斜めに配置し，CPP ビットの値が図の上側から下側に向かって順に確定していくようにする．符号なし乗算と符号付き乗算に対する CPP の配置を図 4.5，図

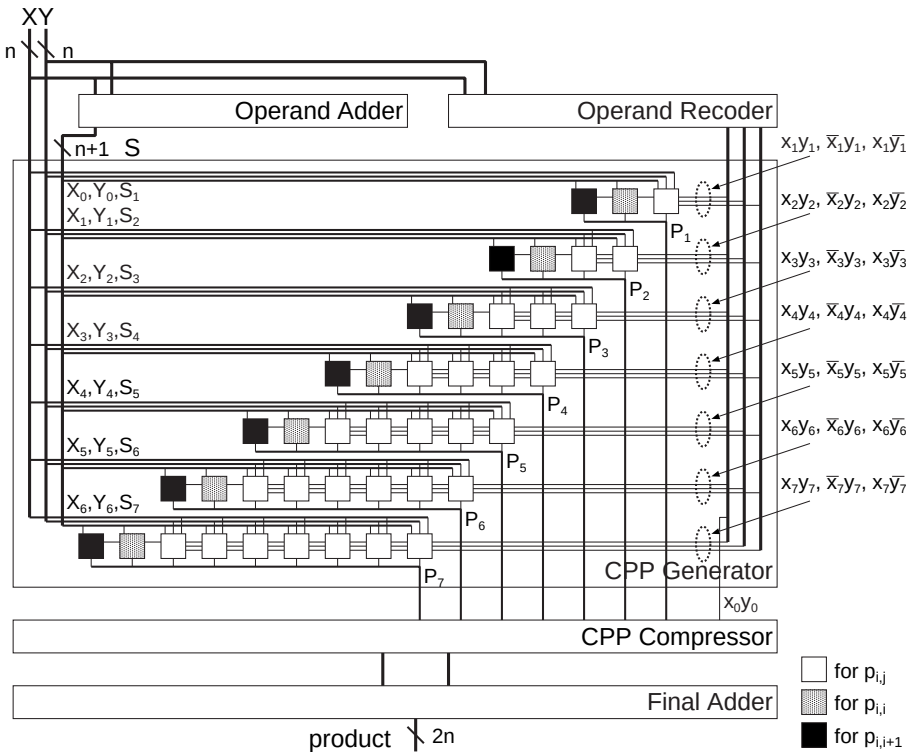


図 4.4: 8 ビット符号なし CPP 乗算器のブロック図

4.6 に示す．CPP を斜めに配置した場合，各 CPP の下位から j ビット目はすべて図の j 行目に配置される．すべての CPP の下位から j ビット目は S の下位から j ビット目 s_j に依存するため，同じ行に配置されたすべての CPP ビットは同時に確定する．配列型 CPP 加算器は図の上側から下側に向かって加算していくため，CPP の値が確定する順序と加算が行われる順序が一致することになる．この配置を採用することによって，オペランド加算による遅延時間の増加を最小限に抑えることができると考えられる．乗算器全体の遅延時間としては，配列型 CPP 加算器を採用した CPP 乗算器と配列型乗算器は同程度になると予想される．

CPP の配置変更による遅延時間の改善について示す．CPP を水平に配置した場合と斜めに配置した場合について乗算器を設計し，面積を最小化する制約条件と遅延時間を最小化する制約条件のもとで最適化を行った．CPP を水平に配置する構成を **horizontal**，斜めに配置する構成を **diagonal** と呼ぶ．詳細な設計，最適化条件は 4.5 節と同一条件であるため，ここでは省略する．**horizontal**，**diagonal** の面積制約下での回路面積と遅延時間を表 4.1 に示す．符号なし乗算器では，ROHM0.18μm，STARC90nm プロセス用セルライブラリを用いた場合に対して，**diagonal** の構成の方がそれぞれ 13%，20%高速であった．また，**diagonal** の方が回路面積につい

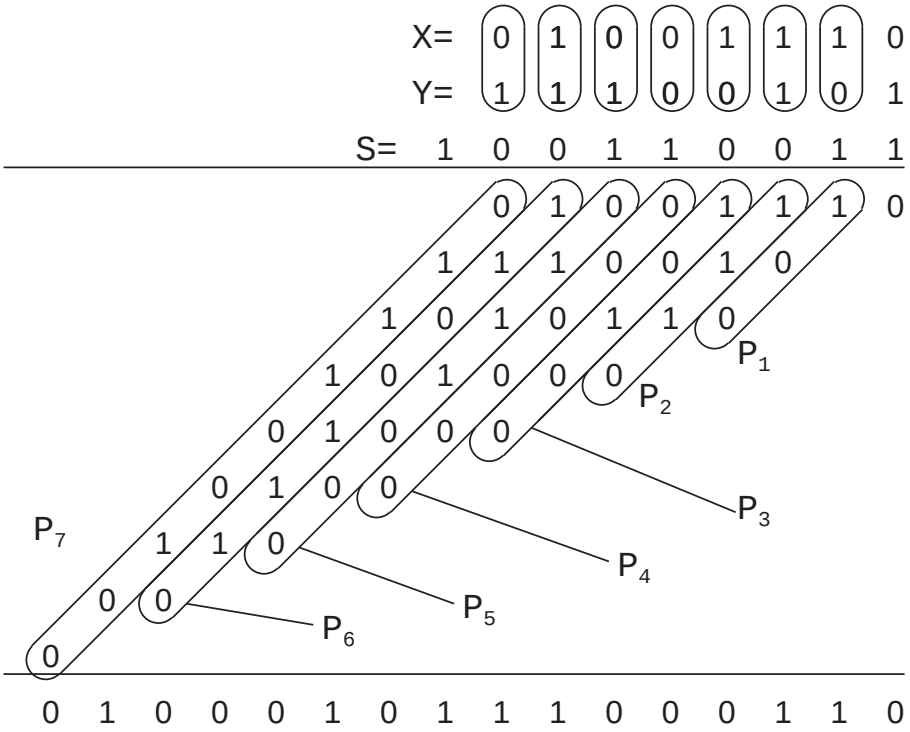


図 4.5: 符号なし乗算の CPP 再配置

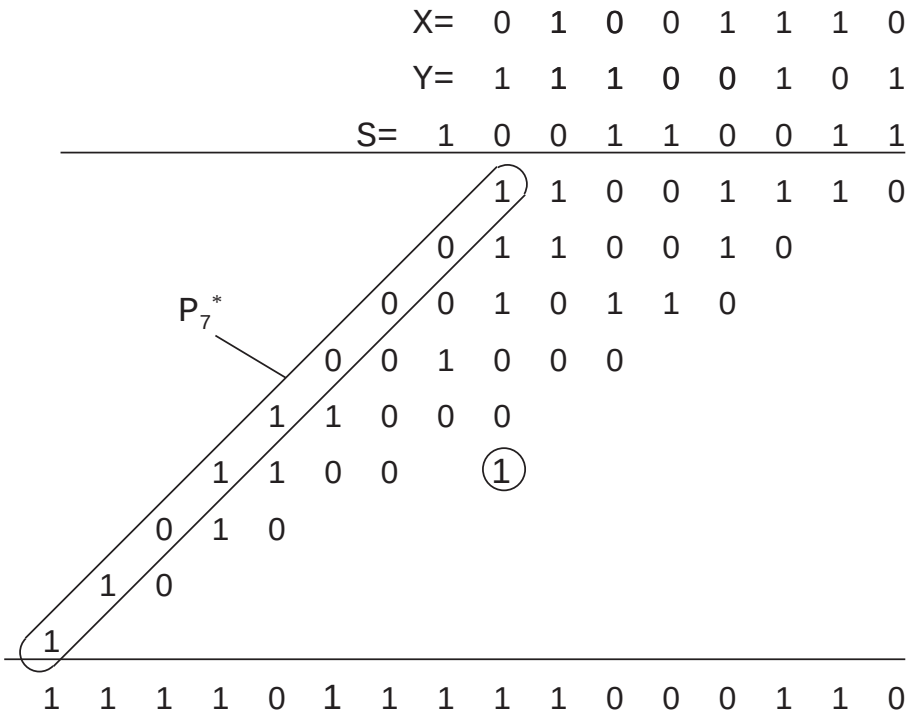


図 4.6: 符号付き乗算の CPP 再配置

表 4.1: CPP の配置方法による符号なし CPP 乗算器の比較

		Rohm 0.18 μm		STARC 90 nm	
		面積	遅延	面積	遅延
16 ビット	horizontal	20070.1	13.08	4592.3	7.04
	diagonal	19459.4	11.35	4039.6	5.63
32 ビット	horizontal	73484.7	25.76	16895.9	14.17
	diagonal	72165.1	22.20	15351.0	10.21
64 ビット	horizontal	280449.5	49.99	64765.3	28.46
	diagonal	279052.3	43.07	59750.9	20.89

てもわずかに小さくなった．符号付き乗算器についてもほぼ同じ傾向が見られた．これ以降，本章では CPP を斜めに配置する構成を採用する．

4.4　CPP 乗算器の高速化

本節では，4.3 節で述べた乗算器を高速化する手法について述べる．CPP を算出するためには，オペランドの和 S を計算する必要がある．一般的に桁上げ伝搬加算は下位ビットから上位ビットに向かって行われるため，CPP の上位ビットが出力されるまでに時間が掛かる．本節の手法では，オペランド加算を複数に分割し，オペランド加算を並列化することによって CPP 生成を高速化する．本節では，例としてオペランド加算を 2 分割した場合について述べる．

X，Y は n ビット符号なし整数とし，簡単のために n は偶数であると仮定する．X，Y の下位 $\frac{1}{2}n$ ビット $(X_L,\ Y_L)$，上位 $\frac{1}{2}n$ ビット $(X_H,\ Y_H)$ の値と表現を次のように定義する．

$$X_L = \sum_{j=0}^{\frac{1}{2}n-1} 2^j x_j\ ,\ [x_{\frac{1}{2}n-1} x_{\frac{n}{2}-2} \cdots x_1 x_0] \tag{4.23}$$

$$Y_L = \sum_{j=0}^{\frac{1}{2}n-1} 2^j y_j\ ,\ [y_{\frac{1}{2}n-1} y_{\frac{n}{2}-2} \cdots y_1 y_0] \tag{4.24}$$

$$X_H = \sum_{j=\frac{1}{2}n}^{n-1} 2^j x_j\ ,\ [x_{n-1} x_{n-2} \cdots x_{\frac{1}{2}n+1} x_{\frac{n}{2}}] \tag{4.25}$$

$$Y_H = \sum_{j=\frac{1}{2}n}^{n-1} 2^j y_j\ ,\ [y_{n-1} y_{n-2} \cdots y_{\frac{1}{2}n+1} y_{\frac{n}{2}}]. \tag{4.26}$$

X，Y の下位 $\frac{1}{2}n$ ビット同士，上位 $\frac{1}{2}n$ ビット同士の和 S_L，S_H を次のように定義する．

$$S_L = X_L + Y_L \ , \ [l_{\frac{1}{2}n} l_{\frac{n}{2}-1} \cdots l_1 l_0] \tag{4.27}$$

$$S_H = X_H + Y_H \ , \ [h_{\frac{1}{2}n} h_{\frac{n}{2}-1} \cdots h_1 h_0] \tag{4.28}$$

S_L，S_H はそれぞれ $\frac{1}{2}n+1$ ビット符号なし整数とみなす．S_L，S_H と S の間には $S = S_L + 2^{\frac{1}{2}n} S_H$ の関係が成立している．

CPP の各ビットは S の各ビットの値に依存して決定される．ここでは，S の代わりに S_L，S_H を用いて CPP を生成する．S_L，S_H を用いると，P_i に対応する CPP として，2 つの $\frac{1}{2}n+1$ ビットの CPP，P_{Li}，P_{Hi} が生成される．P_i は $i \geq 1$ に対して $i+2$ ビットである．P_{Li} は S_L に依存しており，$\frac{1}{2} > i \geq 1$ に対して $i+2$ ビット，$n > i \geq \frac{1}{2}n$ に対して $\frac{1}{2}n+1$ ビットとなる．P_{Hi} は S_H に依存しており，$n > i \geq \frac{1}{2}n$ に対して $i-2$ ビットとなる．P_i と P_{Li}，P_{Hi} の間には $P_i = P_{Li} + 2^{\frac{1}{2}n} P_{Hi}$ の関係が成立する．オペランド加算を 2 つに分割した場合の CPP を図 4.7 に示す．破線の上側が P_{Li}，下側が P_{Hi} である．S_L，S_H は並列に計算できるため，P_{Hi}，P_{Li} は並列に生成できる．また，CPP 加算も P_{Li}，P_{Hi} に対応する 2 つの部分に分割して並列に実行できる．

本節の高速化手法を用いると，CPP の総ビット数が $\frac{1}{2}n$ ビット増加する．オペランドの和 S は $n+1$ ビットであるが，オペランド加算を 2 つに分割したことによりオペランドの和の総ビット数は $n+2$ ビットとなる．これは S_L の最上位ビット $l_{\frac{1}{2}n}$ が現れるためである．CPP にも $l_{\frac{1}{2}n}$ に対応するビットが追加される．CPP に追加されるビット数は $\frac{1}{2}n$ ビットであり，図 4.7 では網掛けで表示されている．

オペランド加算が $\frac{1}{2}n$ ビットずつ，2 つに分割される場合には，より効率良く乗算を実行することができる．分割後のオペランドの和はそれぞれ $\frac{1}{2}n+1$ ビットであるため，各 CPP も $\frac{1}{2}n+1$ ビット以下となる．このとき，符号なし乗算では CPP 全体を $\lfloor \frac{1}{2}n \rfloor + 1$ 行で配置することが可能である．図 4.8 にオペランド加算を $\frac{1}{2}n$ ビットずつに分割した場合の CPP の配置を示す．符号付き乗算の場合には $\lfloor \frac{1}{2}n \rfloor + 2$ 行で配置可能である．S_L，S_H は並列に下位ビットから上位ビットに向かって計算されるため，CPP においても P_{Li}，P_{Hi}，それぞれの下位ビットから上位ビットに向かって並列に生成される．図 4.8 の配置の場合には，上側の行から下側の行に向かって CPP が生成されることとなる．4.3 節における議論と同様, 配列型 CPP 加算器を用いた場合には，乗算全体の遅延時間に与える影響を最小限に抑えることが可能である．この場合，乗算全体の遅延時間は CPP 加算器と最終加算器の遅延

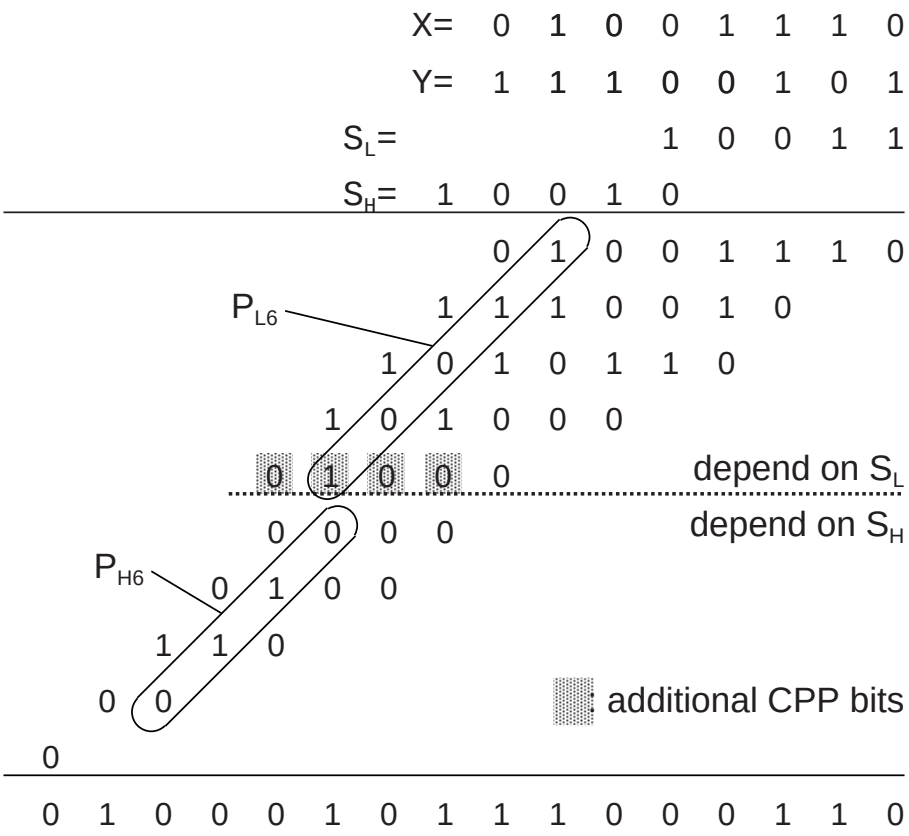


図 4.7: オペランド加算を 2 分割した場合の CPP

時間に左右されると考えられる．本節の高速化手法を用いた場合の乗算器全体の遅延時間は，基数 4 の Booth の方法を用いた配列型乗算器と同程度になると考えられる．

本節では，オペランド加算を 2 つに分割した場合について述べたが，分割は任意の数，任意の箇所で行うことができる．分割数を増やした場合，乗算器の遅延時間は短く，追加の CPP ビット数が増えるため回路面積は増加する傾向となると考えられる．

4.5　評価

4.5.1　評価環境

CPP 乗算器を Rohm 0.18 μm，メタル 5 層 CMOS プロセス向けセルライブラリ，STARC 90 nm，メタル 6 層 CMOS プロセス向けセルライブラリを用いて設計，評価した．これらのセルライブラリは東京大学大規模集積システム設計教育研究セ

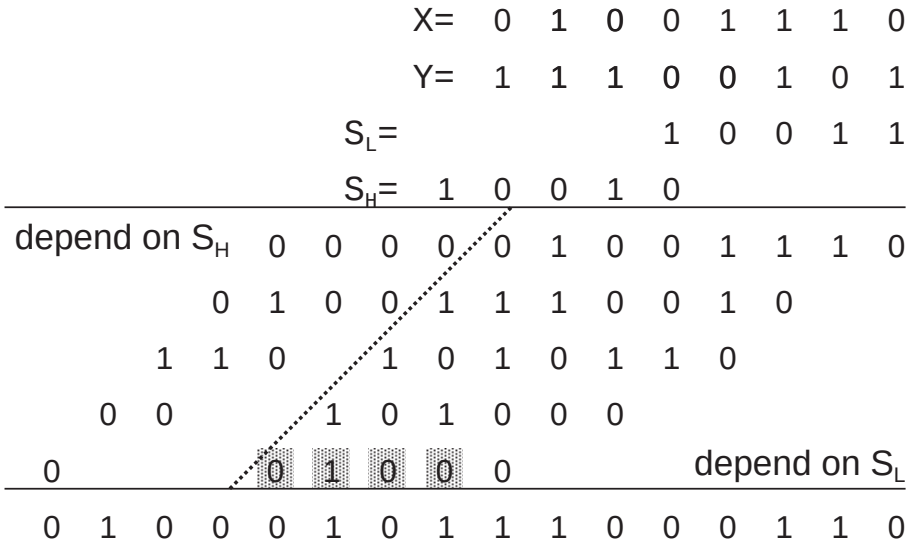


図 4.8: $\lfloor \frac{1}{2}n \rfloor + 1$ 行に配置した CPP

ンターを通して提供されたものを使用した．シノプシス社の Design Compiler を用いて論理合成，最適化を行った．0.18 μm プロセス，90 nm プロセスに対してそれぞれケイデンス社の Soc Encounter，シノプシス社の Astro を用いて配置配線設計を行った．乗算器全体の回路面積に対する素子面積の割合は 95%以上であった．

乗算器の面積と遅延について，CPP 乗算器と既存の乗算器を比較した．入力のビット長は 16 ビット，32 ビット，64 ビットとした．CPP 乗算器は次の 2 つの構成をとる．

CPP+array： CPP 加算器として，配列型の構成を用いる．オペランド加算は分割しない．

CPP+div2+array： CPP 加算器として，配列型の構成を用いる．オペランド加算を同じサイズの 2 つに分割する．

比較対象として次の 2 つの乗算器を用いる．

array： 基本的な部分積生成法を用いた配列型乗算器．

array+Booth： 基数 4 の Booth の方法を用いた配列型乗算器．

4.5.2　オペランドの和を用いた部分積生成法の評価

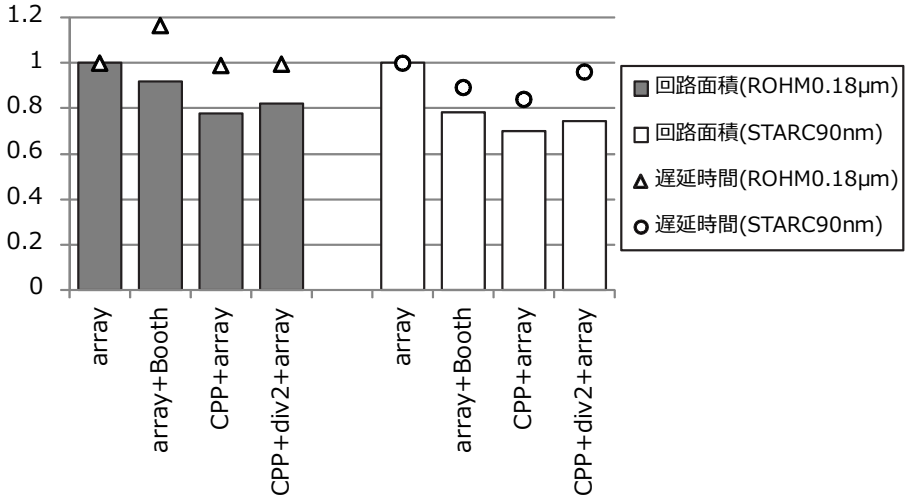
図 4.9 に面積最適化を行った場合の乗算器の回路面積と遅延時間を示す．表 4.2 に面積最適化を行った場合の乗算器の回路面積と遅延時間を示す．最適化は面積

が最小化されるように条件を設定し，遅延時間の制約条件は設定しない．最適化の結果，乗算器に含まれるすべての桁上げ伝搬加算器は順次桁上げ加算器の構成をとった．図 4.10 に，面積，遅延の両方に対して最適化条件を設定した場合の回路面積と遅延時間を示す．この場合の最適化条件としては，遅延時間，回路面積ともに最小化する設定とし，より遅延時間を優先して最適化を行うように設定した．ただし，遅延時間を強く最適化すると回路面積が爆発的に大きくなるため，遅延時間に対する制約を僅かにゆるめて回路面積が大きくなり過ぎないようにする．この最適化条件下では，加算器としてシノプシス社の DesignWare IP ライブラリから DW01_add を用いた．それぞれ，配列型乗算器の回路面積と遅延時間を 1 として正規化し，棒グラフで回路面積、プロットで遅延時間を表す．それぞれの詳細な評価値は表 4.2，表 4.3 に示す．

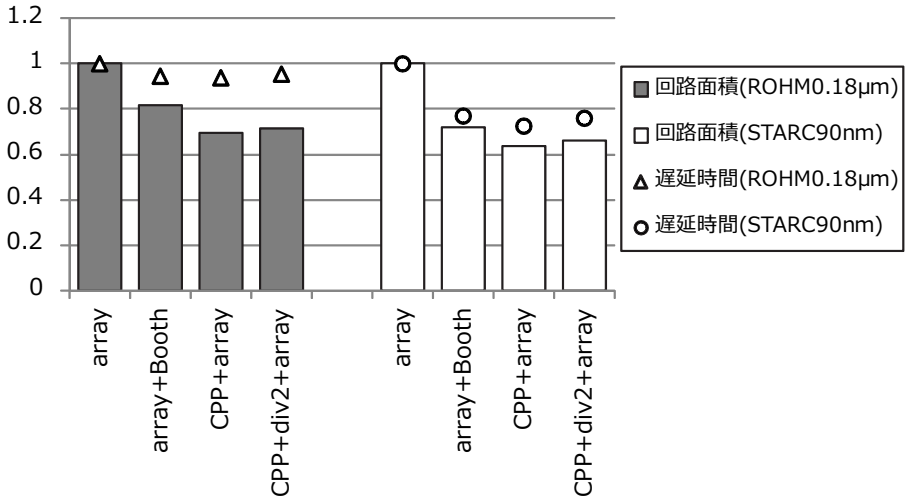
面積制約下では，CPP 乗算器は既存の乗算器よりも小面積となった．**CPP+array** の面積は **array** と比較して，16 ビット，32 ビット，64 ビットに対してそれぞれ約 20%，30%，35%小さくなった．**CPP+array** を **array+Booth** と比較した場合，約 10%小さくなった．遅延・面積制約下では，CPP 乗算器は基数 4 の Booth の方法を用いた配列型乗算器よりも小面積かつわずかに遅延時間が大きくなった．**CPP+div2+array** は **array+Booth** に対し，0.18 μm プロセス，90nm プロセスでそれぞれ 22.7%，12.2%小さくなった．一方，遅延時間はわずかに大きくなった．これらの結果から，提案手法は面積の小さい乗算器を構成するためには有用であるといえる．また，CPP 乗算器は基数 4 の Booth の手法を用いた乗算器を置き換えることが可能な性能を有している．4.3 節と 4.4 節では，**CPP+array** と **CPP+div2+array** の遅延時間はそれぞれ **array** と **array+Booth** と同等であると予測した．この予測に対する実験結果について述べる．**CPP+array** は，どの設計条件においても **array** に対して非常に近い遅延時間を示している．また，**CPP+div2+array** と **array+Booth** の遅延時間の差は 10%以内に収まっている．

4.5.3　CPP 乗算器の高速化手法の評価

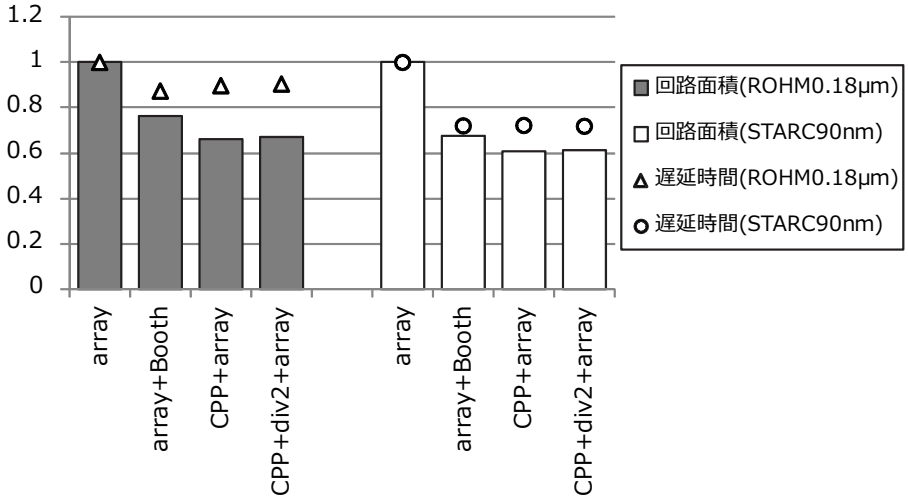
4.4 節の高速化手法の評価について述べる．図 4.9 に示すとおり，面積制約下では高速化手法は効果がないことがわかった．面積制約下では最終加算器の構成は，小面積で遅延時間が大きい順次桁上げ加算器となることが予想される．そのため，最終加算器が乗算器全体のクリティカルパスとなり，高速化手法によるオペランド加算及び CPP 加算の高速化を隠蔽する結果となったと考えられる．一方，図 4.10



(a) 16 ビット CPP 乗算器の評価

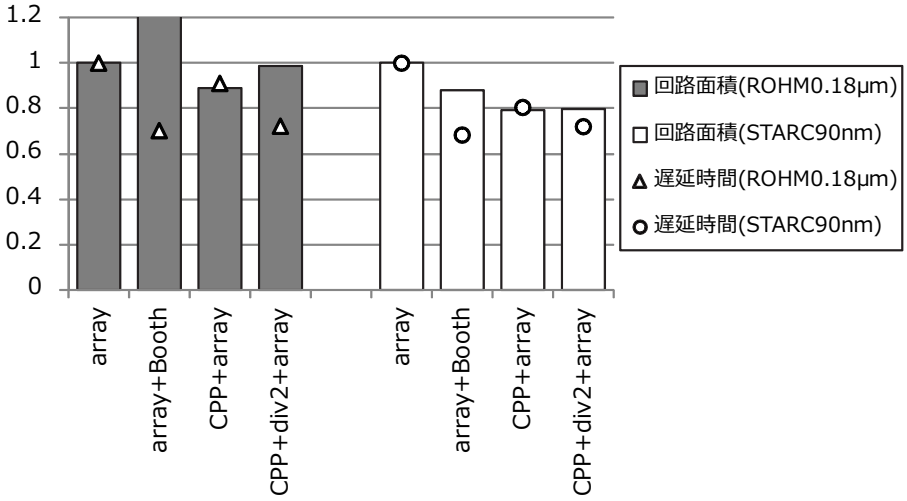


(b) 32 ビット CPP 乗算器の評価

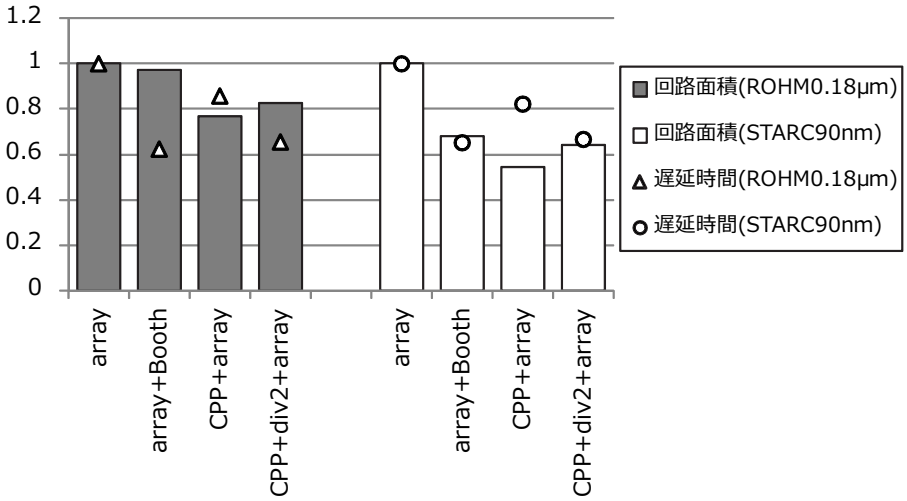


(c) 64 ビット CPP 乗算器の評価

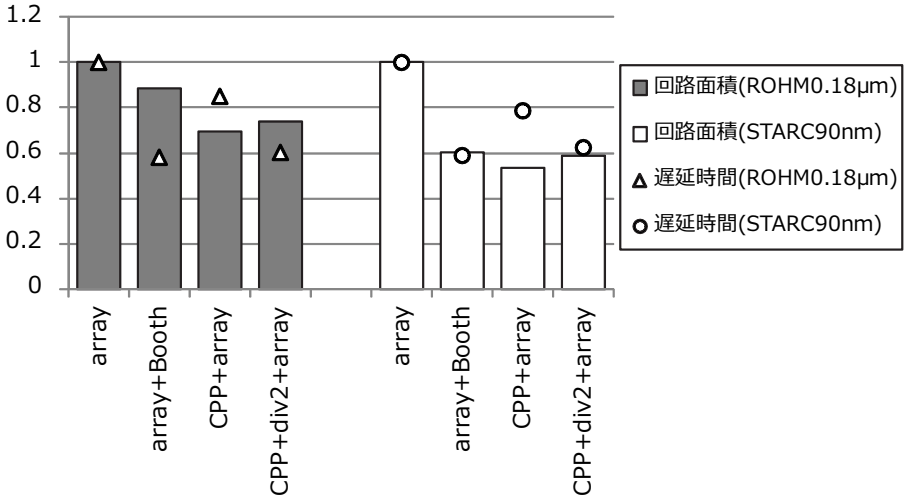
図 4.9: 面積制約下で最適化した CPP 乗算器の評価



(a) 16 ビット CPP 乗算器の評価



(b) 32 ビット CPP 乗算器の評価



(c) 64 ビット CPP 乗算器の評価

図 4.10: 面積・遅延制約下で最適化した CPP 乗算器の評価

表 4.2: 面積制約下で最適化した CPP 乗算器の評価

			Rohm 0.18 μm		STARC 90 nm	
			面積 (μm^2)	遅延 (ns)	面積 (μm^2)	遅延 (ns)
符号なし	16 ビット	array	24946.8	11.47	5790.6	6.70
		array+Booth	22902.3	13.37	4522.5	5.98
		CPP+array	19459.4	11.35	4039.6	5.63
		CPP+div2+array	20470.9	11.42	4311.2	6.44
	32 ビット	array	103775.1	23.67	24127.1	14.09
		array+Booth	84552.3	22.37	17362.1	10.84
		CPP+array	72165.1	22.20	15351.0	10.21
		CPP+div2+array	74397.7	22.58	15911.3	10.70
	64 ビット	array	423104.6	48.00	98448.4	28.95
		array+Booth	322622.9	41.93	66727.6	20.84
		CPP+array	279052.3	43.07	59750.9	20.89
		CPP+div2+array	283645.9	43.41	60298.5	20.79
符号付き	16 ビット	array	24753.1	11.61	5748.2	6.73
		array+Booth	21618.5	13.08	4251.6	5.59
		CPP+array	19615.9	11.40	4057.2	5.62
		CPP+div2+array	21652.1	11.42	4384.6	5.92
	32 ビット	array	103362.3	23.83	24037.2	14.21
		array+Booth	81821.4	23.36	16651.1	10.77
		CPP+array	72483.0	22.39	15385.6	10.65
		CPP+div2+array	76746.4	22.24	16052.4	10.86
	64 ビット	array	422256.8	48.18	98264.0	28.90
		array+Booth	317325.6	41.25	65437.1	20.25
		CPP+array	279697.0	43.20	59750.9	20.94
		CPP+div2+array	288414.8	43.61	61196.7	22.24

表 4.3: 面積・遅延制約下で最適化した CPP 乗算器の評価

			Rohm 0.18 μm		STARC 90 nm	
			面積 (μm^2)	遅延 (ns)	面積 (μm^2)	遅延 (ns)
符号なし	16 ビット	array	27478.9	8.15	12954.8	2.71
		array+Booth	35103.0	5.72	11379.9	1.85
		CPP+array	24423.4	7.42	10245.3	2.18
		CPP+div2+array	27122.5	5.88	10358.2	1.95
	32 ビット	array	108445.4	15.73	51755.8	5.45
		array+Booth	105590.0	9.80	35140.3	3.55
		CPP+array	83170.4	13.49	28034.9	4.48
		CPP+div2+array	89573.8	10.30	33225.3	3.63
	64 ビット	array	432733.4	31.22	192510.3	11.43
		array+Booth	383888.0	18.14	115831.3	6.73
		CPP+array	301362.8	26.54	102784.1	8.99
		CPP+div2+array	319258.6	18.80	112801.5	7.13
符号付き	16 ビット	array	27262.8	8.29	12859.6	2.61
		array+Booth	35041.5	6.42	12294.4	1.90
		CPP+array	24865.6	7.14	9796.6	2.26
		CPP+div2+array	27214.4	6.37	10790.7	2.08
	32 ビット	array	108288.2	15.88	51882.8	5.44
		array+Booth	107904.9	10.73	33841.3	3.53
		CPP+array	86848.3	13.13	30380.3	4.35
		CPP+div2+array	91805.6	10.48	32262.9	3.70
	64 ビット	array	432169.9	30.53	186657.3	11.55
		array+Booth	384976.5	19.35	124665.4	6.63
		CPP+array	309527.1	26.01	116498.1	9.34
		CPP+div2+array	320856.1	19.91	115926.6	7.48

に示す面積・遅延制約下での最適化結果においては，高速化手法は有効であり，どの条件においても **CPP+div2+array** が **CPP+array** よりも高速となっている．面積・遅延制約下においては，最終加算器の構成は順次桁上げ加算器よりも高速な構成となると予想される．この場合，配列型 CPP 加算器の遅延時間が乗算器全体の遅延時間に対して支配的になると考えられる．そのため，CPP 加算器の高速化によって，乗算器全体の遅延時間が改善されたと考えられる．

4.5.4　Wallace 木を用いた構成の比較

並列乗算器の採用にあたって，遅延時間は欠くことのできない重要な要素である．本章は並列乗算器の小面積化を目的としているが，ここでは簡単に遅延時間についての議論を行う．高速並列乗算器では，部分積加算器として Wallace 木が広く用いられている．CPP 乗算器を構成する CPP 加算器において Wallace 木を用いた場合の実験結果を示す．

CPP+Wallace： CPP 加算器に Wallace 木を用いた乗算器．オペランド加算の分割は行わない．

Wallace： 基数 4 の Booth の手法を用いず，部分積加算器に Wallace 木を用いた乗算器．

Wallace+Booth： 基数 4 の Booth の手法を用い，部分積加算器に Wallace 木を用いた乗算器．

表 4.4，表 4.5 に Wallace 木を用いた乗算器の面積と遅延時間を示す．表 4.4 に面積制約下で最適化した場合の回路面積と遅延時間を示す．表 4.2 の配列型の部分積加算器を用いた場合と同じ傾向を示した．表 4.5 に面積・遅延制約下で最適化した場合の回路面積と遅延時間を示す．**CPP+Wallace** は **Wallace**，**Wallace+Booth** よりも小面積であるが，遅延時間が大きくなった．**Wallace**，**Wallace+Booth** ではオペランドの長さに依存せずに部分積生成が完了する．一方，**CPP+Wallace** で用いている CPP 生成器では桁上げ伝搬加算を行なっており，オペランドの長さに依存した時間がかかる．そのため，**CPP+Wallace** の方が遅延時間が大きくなったと考えられる．

表 4.4: 面積制約下で最適化した Wallace 木を用いた CPP 乗算器の評価

			Rohm 0.18 μm		STARC 90 nm	
			面積 (μm^2)	遅延 (ns)	面積 (μm^2)	遅延 (ns)
符号なし	16 ビット	Wallace	25085.9	10.18	5823.8	4.69
		Wallace+Booth	22814.1	13.05	4743.7	5.94
		CPP+Wallace	19650.0	11.44	4292.9	5.50
	32 ビット	Wallace	103911.6	20.73	24160.9	9.27
		Wallace+Booth	84359.8	24.59	17320.2	10.37
		CPP+Wallace	72357.5	22.66	15981.8	10.38
	64 ビット	Wallace	423241.2	42.59	98482.8	18.85
		Wallace+Booth	322250.3	46.03	67516.7	19.52
		CPP+Wallace	279246.2	44.34	61542.4	21.06
符号付き	16 ビット	Wallace	24892.2	10.30	5780.0	4.68
		Wallace+Booth	20806.7	12.69	4366.3	5.18
		CPP+Wallace	19765.6	11.40	4110.1	5.27
	32 ビット	Wallace	103502.0	20.62	24069.5	9.37
		Wallace+Booth	80054.6	23.80	17244.9	10.09
		CPP+Wallace	72632.6	22.82	15946.6	10.36
	64 ビット	Wallace	422398.8	42.28	98295.3	18.79
		Wallace+Booth	313588.5	46.51	65279.3	19.73
		CPP+Wallace	279849.0	44.70	61542.4	20.84

表 4.5: 面積・遅延制約下で最適化した Wallace 木を用いた CPP 乗算器の評価

			Rohm 0.18 μm		STARC 90 nm	
			面積 (μm^2)	遅延 (ns)	面積 (μm^2)	遅延 (ns)
符号なし	16 ビット	Wallace	32432.9	4.30	12356.5	1.45
		Wallace+Booth	38168.0	4.18	9824.1	1.46
		CPP+Wallace	29893.8	4.96	7413.0	1.83
	32 ビット	Wallace	121921.2	5.45	41037.7	2.41
		Wallace+Booth	119321.4	5.50	34931.4	2.26
		CPP+Wallace	98790.6	7.04	27151.5	2.76
	64 ビット	Wallace	468251.3	7.35	150714.8	4.02
		Wallace+Booth	411641.9	7.48	112519.9	3.58
		CPP+Wallace	345247.5	10.32	105950.1	4.06
符号付き	16 ビット	Wallace	32674.6	4.07	10386.4	1.43
		Wallace+Booth	41975.1	4.04	10877.5	1.65
		CPP+Wallace	31506.8	4.72	7878.0	1.89
	32 ビット	Wallace	123510.4	5.64	38745.9	2.27
		Wallace+Booth	120589.5	5.40	32112.6	2.19
		CPP+Wallace	101488.9	6.81	25293.6	2.71
	64 ビット	Wallace	470246.1	7.73	151475.4	3.59
		Wallace+Booth	413382.9	7.65	107321.8	3.46
		CPP+Wallace	341909.2	9.72	99666.0	4.20

4.6　まとめ

本章では並列乗算で用いられる新しい部分積生成法を示し，それに基づく並列乗算器を提案した．オペランドの和を利用した部分積生成法はCPP と呼ぶ部分積を導入することにより，乗算全体で部分積のビット数を従来の部分積から約半分に削減した．CPP はオペランドの和を利用することにより効率良く生成することが可能である．本章ではCPP を用いた乗算器を高速化する手法も提案した．高速化手法は，オペランド加算を複数に分割し，並列に実行する．これにより，CPP 生成，CPP 加算を並列に行うことができるようになり，乗算全体の高速化に成功した．実験結果より，CPP を用いた乗算器の有効性を示すことができた．CPP を用いた乗算器は，小面積な乗算器として用いられてきた基数 4 の Booth の手法を用いた配列型乗算器よりも優れた性能を示しており，これを置き換えることが期待される．

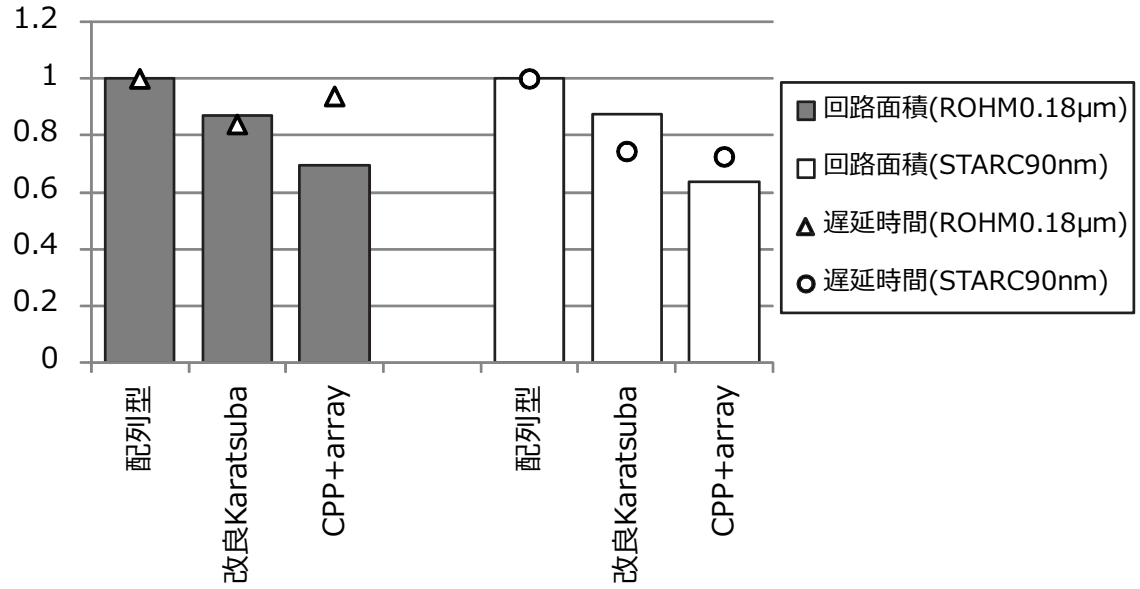
本章では，オペランド加算を二等分した場合に限定して議論を行った．CPP を用いた乗算器には，オペランド加算の分割数，分割位置，CPP 加算器の構成等，様々な構成が考えられる．今後，より効率の良い構成を発見することで更に並列乗算器の性能を向上することが可能であると考えられる．また，CPP を用いた乗算器はオペランドの和を利用しているため，乗算と加算を同時に行うことが可能である．オペランドの和と積が同時に必要になるようなアプリケーションでは，より効率的に乗算器を使用することが可能であると考えられる．

第5章　Karatsuba乗算器とCPP乗算器の比較

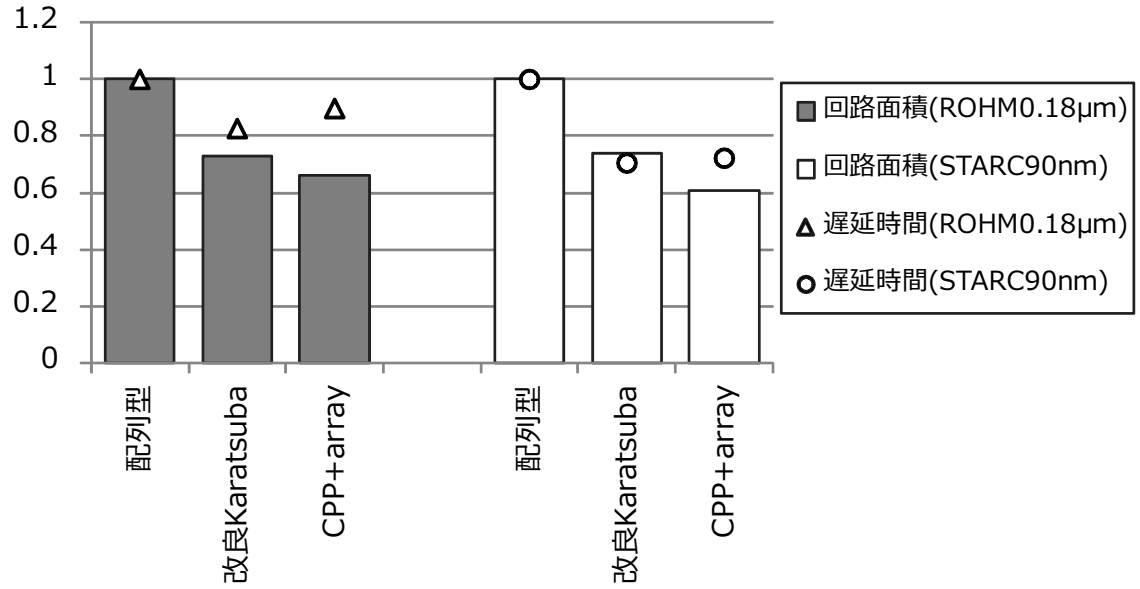
3章，4章では，それぞれKaratsubaアルゴリズムに基づく並列乗算器，オペランドの和を用いた部分積生成法に基づく並列乗算器を提案した．本章では，提案した2つの構成法を用いた並列乗算器を比較し，最も小面積な並列乗算器について考察する．

図5.1に符号なしの配列型乗算器とKaratsuba乗算器，CPP乗算器の比較を示す．棒グラフは回路面積，プロットは遅延時間を示す．Karatsuba乗算器は3章で示した改良Karatsuba乗算器，CPP乗算器は4章で示したCPP+arrayを用い，評価値はそれぞれ図3.8，図4.9から抜粋した．3章と4章の評価では細部の設計や論理合成に用いたツールのバージョン，最適化の設定等，細かい評価条件が異なるため，同じ条件での比較はできない．そこで，図5.1では配列型乗算器を1として正規化して比較を行なっている．本章では回路面積の評価が主な目的であるため，並列乗算器の構成は回路面積が最小になる構成を用いた．

32ビット，64ビットともにCPP乗算器の方が小面積であるが，64ビットではKaratsuba乗算器とCPP乗算器の回路面積の差が小さくなっている．64ビットKaratsuba乗算器ではKaratsubaアルゴリズムを2回適用した構成となっており，Karatsubaアルゴリズムの再帰的適用の効果によって面積の差が小さくなったと考えられる．さらに入力ビット幅が大きい場合には，Karatsubaアルゴリズムをさらに再帰的に適用することができる．Karatsubaアルゴリズムを再帰的に適用した場合，配列型乗算器やCPP乗算器よりも素子数のオーダーが小さくなるため，入力のビット幅が大きくなればなるほどKaratsuba乗算器の方が小面積になると考えられる．



(a) 32 ビット乗算器の評価



(b) 64 ビット乗算器の評価

図 5.1: Karatsuba 乗算器と CPP 乗算器の比較

第6章　結論

本論文では，回路面積の小さい並列乗算器を構成するための手法を示した．本論文の貢献により，既存の並列乗算器の構成法を用いるよりも小面積な並列乗算器を構成することが可能となった．

Karatsubaアルゴリズムに基づく小面積乗算器は，多倍長乗算でよく用いられるKaratusbaアルゴリズムを並列乗算器に適用した．Karatsubaアルゴリズムは，入力を上位と下位に分割し，計算を効率良く行うことにより，少ない演算量で乗算を行う．Karatsubaアルゴリズムは再帰的に適用できるため，入力サイズの大きい並列乗算器ではKaratsubaアルゴリズムを複数回にわたって適用できる．そのため，より入力サイズの大きい乗算器でより大きい回路面積の削減効果が得られた．入力のビット幅が大きい場合にはKaratsubaアルゴリズムを再帰的に適用することが有効である．$0.35\mu m$ から $90nm$ のプロセスにおいてKaratsubaアルゴリズムを再帰的に適用することが有効なビット幅を見積もった．14ビットから20ビットよりも小さいビット幅の場合はKaratsubaアルゴリズムを適用するよりも配列型の構成をとったほうが小面積となることがわかった．28ビットから44ビットよりも小さい場合にはKaratsubaアルゴリズムを1回適用した構成が小面積となり，それよりも大きい場合にはKaratusbaアルゴリズムを2回適用した方が小面積となることがわかった．

オペランドの和を利用した部分積生成法に基づく並列乗算器では，オペランドの和を利用することにより部分積のビット数を約半分に削減した．基数4のBoothの方法でも部分積のビット数を約半分に削減しているが，オペランドの和を利用した手法では部分積を生成するための回路をより小さい面積で構成でき，並列乗算器全体の面積も小さく抑えることが可能である．オペランドの加算の並列化による高速化手法を用いた場合は，遅延時間を最適化した場合にも少ない面積増加で高速化効果を得られることがわかった．オペランドの和を利用した並列乗算器は，乗算と加算を同時に行うことが可能である．さらに，Karatusbaアルゴリズムに基づく並列乗算器と組み合わせることにより，多彩な演算を行うことが可能である．算術演算回路を多数搭載し，演算の組合せを有効利用できるような環境において

は，単純な乗算器よりも高い演算能力を発揮することが期待できる．

従来の半導体製造プロセスでは配線を行うための層の数が十分でなく，配線が複雑になる場合には回路面積が大きくなる場合があった．本研究の貢献により，近年の半導体製造プロセスを用いた場合，十分な配線層数が提供されているため配線の複雑さによる回路面積の増加は起こりにくいことがわかった．より素子面積の少なくなる構成法を用いることによって回路面積の小さい並列乗算器が構成できることがわかった．他の演算器でも，多くの配線層数が使用できる設計において小面積な演算器を構成するためには，配線量が多くても少ないセル数となる回路の構成法を選択することが有効であると考えられる．

謝辞

本研究を遂行するにあたり，多くの皆様に御指導をいただきました．ここに心からの感謝の意を表します．

京都大学大学院情報学研究科の高木直史教授には，名古屋大学大学院情報科学研究科在任中より，研究の全般に渡り格別の御指導をいただきました．深く感謝いたします．京都大学大学院情報学研究科の高木一義准教授，名古屋大学大学院情報科学研究科の中村一博助教には日頃より研究内容について有益なご指摘，ご議論を頂き，心より感謝いたします．また，御指導いただいた高木研究室の先輩方をはじめ，議論していただいた学生の皆様に感謝いたします．

研究の後半におきましては，組込みシステムの研究に関して名古屋大学大学院情報科学研究科の高田広章教授，枝廣正人教授，本田晋也准教授，工学研究科の曾剛講師に丁寧な御指導をいただきました．深く感謝いたします．また，ご議論いただいた高田研究室および附属組込みシステム研究センターの皆様に感謝いたします．

最後に，長きに渡り私生活の面から支えてくれた妻美幸に心から感謝いたします．

参考文献

[1] 高木直史, 算術演算の VLSI アルゴリズム, コロナ社，2005.

[2] A. D. Booth, “A Signed Binary Multiplication Technique,” Quarterly Journal of Mechanics and Applied Mathematics, vol. 4, no. 2, pp. 236-240, 1951.

[3] O. McSorley, “High-Speed Arithmetic in Binary Computers,” *Proceedings of the IRE*, vol.49, pp. 67-91, 1961.

[4] R. Baugh and B.A. Wooley, “A Two’s Complement Parallel Array Multiplication Algorithm,” IEEE Trans. Computers, vol. 22, no. 12, pp. 1045-1059, 1973.

[5] C.S. Wallace, “A suggestion for a fast multiplier,” IEEE Transactions on Electronic Computers, vol.EC-13, pp. 14-17, 1964.

[6] L. Dadda, “On Parallel Digital Multipliers,” Alta Frequenza, vol. 45, pp. 574-580, 1976.

[7] W.K. Luk and .E. Vuillemin “Recursive Implementation of Optimal Time VLSI Integer Multipliers,” International Conference on VLSI 1983, pp. 155-168, Aug. 1983.

[8] N. Takagi, H. Yasuura and S. Yajima, “High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree,” *IEEE Trans. on Computers*, vol.C-34, no.9, pp. 789-796, 1985.

[9] M. Ito, D. Chinnery, K. Keutzer, “Low Power Multiplication Algorithm for Switching Activity Reduction through Operand Decomposition,” 2003 IEEE International Conference on Computer Design (ICCD’03), pp. 21-26, 2003.

[10] T. Matsunaga and Y. Matsunaga, "Timing-Constrained Area Minimization Algorithm for Parallel Prefix Adders," IEICE Trans. Fundamentals, vol.E90-A, no.12, pp. 2770-2777, 2007.

[11] A. Karatsuba and Yu. Ofman, "Multiplication of multidigit numbers on automata," Soviet Physics Doklady, vol.7, pp. 595-596, Jan. 1963.

[12] D. Knuth, "The Art of Computer Programming, vol.2:Seminumerical Algorithms," Third Edition, Addison-Wesley, 1997.

[13] Z. Dyka and P. Langendoerfer, "Area efficient hardware implementation of elliptic curve cryptography by iteratively applying Karatsuba's method," Proc. of the Design, Automation and Test in Europe Conference and Exhibition, vol.3, pp. 70-75, Munich, Germeny, Mar. 2005.

[14] S. Yazaki and K.Abe, "VLSI implementation of Karatsuba algorithm and its evaluation," Proc. The International Workshop on Modern Science and Technology 2006, pp. 378-383, Wuhan, China, May 2006.

[15] S. Yazaki and K.Abe, "VLSI design of iterative Karatsuba multiplier and its evaluation," Proc. The 4th IASTED International Conference on Circuits, Signals, and Systems, pp. 313-318, San Francisco, USA, Nov. 2006.

[16] P.L. Montgomery, "Five, six, and seven-term Karatsuba-like formulae," IEEE Transactions on Computers, vol.54-3, pp. 362-369, 2005.

[17] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann Publishers, 2003.

[18] I.-N. Chen and R. Willoner, "An 0() Parallel Multiplier with Bit-Sequential Input and Output," *IEEE Trans. on Computers*, vol.28, no.10, pp. 721-727, 1979.

[19] N. R. Strader and V. T. Rhyne, "A Canonical Bit-Sequential Multiplier," *IEEE Trans. on Computers*, vol.31, no.8, pp. 791-795, 1982.

[20] R. Gnanasekaran, "On a Bit-Serial Input and Bit-Serial Output Multiplier," *IEEE Trans. on Computers*, vol.32, no.9, pp. 878-880, 1983.

研究業績

学術論文

- 川島裕崇，柴岡雅之，高木直史，高木一義，”Karatsubaアルゴリズムに基づく小面積乗算器”，電子情報通信学会論文誌，vol. J91-A，no. 7，pp. 707-715，2008.

- Hirotaka Kawashima and Naofumi Takagi, “Partial Product Generation Utilizing the Sum of Operands for Reduced Area Parallel Multipliers,” IPSJ Transactions on System LSI Design Methodology, vol. 4, pp.131-139, 2011.

- 立松知紘, 高瀬英希, 曾剛, 川島裕崇, 冨山宏之, 高田広章, ”実行トレースを用いた組込みシステムにおけるタスク内DVFSのためのチェックポイント抽出,” 情報処理学会論文誌, vol. 52, no. 12, pp. 3729-3744, 2011年12月.

- Hirotaka Kawashima, Gang Zeng, Hideki Takase, Masato Edahiro and Hiroaki Takada, “Efficient Algorithms for Extracting Pareto-optimal Hardware Configurations in DEPS Framework,” IPSJ Transactions on System LSI Design Methodology, vol. 6, 2012(採録決定).

国際会議論文

- Hirotaka Kawashima and Naofumi Takagi, “Small Area Multipliers Utilizing the Sum of Operands,” The 15th Workshop on Synthesis And System Integration of Mixed Information technologies, pp. 189-194, Okinawa, Japan, Mar. 9-10, 2009.

- Hideki Takase, Gang Zeng, Lovic Gauthier, Hirotaka Kawashima, Noritoshi Atsumi, Tomohiro Tatematsu, Yoshitake Kobayashi, Shunitsu Kohara, Takenori Koshiro, Tohru Ishihara, Hiroyuki Tomiyama and Hiroaki Takada, “An Integrated Optimization Framework for Reducing the Energy Consumption of Embedded Real-Time Applications,” In Proc. of International Symposium on Low-Power Electronics and Design (ISLPED), pp. 271-276, Fukuoka, Japann, Aug. 2011.

- Hirotaka Kawashima, Gang Zeng, Hideki Takase, Masato Edahiro, Hiroaki Takada, “Checkpoint Selection for DEPS Framework Based on Quantitative Evaluation of DEPS Profile,” The 17th Workshop on Synthesis And System Integration of Mixed Information technologies, pp. 174-179, Beppu, Japan, Mar. 8-9, 2012.

研究会発表論文 (査読なし)

- 川島裕崇, 高木直史, 高木一義, “配線層数の乗算器の回路面積への影響について,” 電子情報通信学会技術報告, VLD2006-141, pp.7-11, 沖縄, Mar. 2007.
- 川島裕崇, 中村一博, 高木直史, 高木一義, “部分積加算における信号遷移回数の削減による配列型乗算器の低消費エネルギー化設計,” 電子情報通信学会技術報告, VLD2007-87, pp.31-36, 福岡, Nov. 2007.
- 川島裕崇, 高木直史, “オペランドの和を利用した小面積乗算器,” 電子情報通信学会技術報告, VLD2008-64, pp.25-30, 福岡, Nov. 2008.
- 川島裕崇, 高木直史, “オペランドの和を用いた並列乗算器の消費エネルギー評価,” 電子情報通信学会技術報告, VLD2009-66, pp.173-178, 高知, Dec. 2009.
- 川島裕崇, 中村一博, 高木一義, 高木直史, “セル遅延モデルを用いた算術演算回路の信号遷移回数見積もり手法,” 電子情報通信学会技術報告, VLD2009-124, pp.151-156, 沖縄, Mar. 2010.
- 川島裕崇, 曾剛, 渥美紀寿, 立松知紘, 高田広章, “DEPS フレームワークにおける最悪実行時間と平均消費エネルギーのタスク内解析手法,” 電子情報通信学会技術報告, VLD2010-119, pp.19-24, 沖縄, Mar. 2011.
- 川島裕崇, 曾剛, 渥美紀寿, 立松知紘, 高瀬英希, 高田広章, “DEPS プロファイルの評価法とそれを利用したチェックポイント選定,” 情報処理学会研究報告, 2011-SLDM-149(4), Mar. 2011.
- Hideki Takase, Gang Zeng, Hirotaka Kawashima, Noritoshi Atsumi, Tomohiro Tatematsu, Lovic Gauthier, Tohru Ishihara, Yoshitake Kobayashi, Shunitsu Kohara, Takenori Koshiro, Hiroyuki Tomiyama and Hiroaki Takada,

“An Energy Optimization Framework for Embedded Applications,” 情報処理学会研究報告, 2011-SLDM-149(3), Mar. 2011.

- 川島裕崇, “AUTOSAR OS 仕様ベースの RTOS の仕様検討および開発に関するコンソーシアム型共同研究,” 第 13 回組込みシステム技術に関するサマーワークショップ (SWEST13) 予稿集, pp. 25-28, 下呂, Sep. 2011.

大会等

- 川島裕崇, 高木直史, 高木一義, “Karatsuba 乗算器の設計と評価, 2006 年電子情報通信学会ソサイエティ大会講演論文集,” A-3-8, Sep. 2006.
- 川島裕崇, 高木直史, “Chen-Willoner アルゴリズムに基づく小面積並列乗算器について,” 2008 年電子情報通信学会ソサイエティ大会講演論文集, A-3-6, Sep. 2008.

受賞等

- 2007 年度名古屋大学学術奨励賞
- 平成 21 年度システム LSI 設計技術研究会優秀論文賞「オペランドの和を利用した小面積乗算器」，川島裕崇，高木直史
- 第 137 回 SLDM 研究会優秀発表学生賞「オペランドの和を利用した小面積乗算器」，川島裕崇，高木直史