多倍長乗算回路の構成と評価に関する研究

矢崎 俊志

電気通信大学電気通信学研究科
博士 (工学) の学位申請論文

2007 年 3 月

多倍長乗算回路の構成と評価に関する研究

博士論文審査委員会

主査: 尾内 理紀夫 教授
委員: 野下 浩平 教授
委員: 加古 孝 教授
委員: 山本 野人 教授
委員: 小林 聡 助教授
委員: 阿部 公輝 助教授

2007 年 3 月

A Study on Organization and Evaluation of Multi-digit Multiplier

Syunji YAZAKI

Abstract

This thesis describes a study done on hardware organization and evaluation of multipliers for multi-digit integers with large bit lengths which exceed those of general-propose processors.

Multi-digit arithmetic is widely used for various applications in recent years, including numerical calculation, chaos arithmetic, primality testing. Multi-digit arithmetic requires much computation time especially for multiplication frequently used in many applications, leading to forming a bottleneck in systems. Many sophisticated multiplication algorithms faster than the traditional schoolbook algorithm with complexity of $O(n^2)$ have been proposed, among which Karatsuba 2-way, 3-way, 4-way, and 5-way methods of $O(n^{1.58})$, $O(n^{1.465})$, $O(n^{1.404})$, and $O(n^{1.365})$ respectively, modulo arithmetic of $O(n^{1.63})$, and Fast Fourier Transform (FFT) method of $O(n \log n \log \log n)$ are well known, where n stands for number of multiplication digits. Although FFT method is the fastest with respect to computational complexity, its actual performance is worse than that of Karatsuba method for calculation of hundreds to tens of thousands bit multiplication due to the overheads caused by complex-number operations. Karatsuba method is frequently employed for many applications except for those requiring millions digit multiplication. Many software implementations of these algorithms are available and several studies on hardware implementations of multi-digit multiplication over Galois field (GF) have been reported. However, studies on multi-digit multiplication for integers are rare.

The goal of this study is to implement multi-digit multiplication by hardware for integers with very large bit length using FFT algorithm and for integers with medium bit length using Karatsuba method, and evaluate their costs and

performances compared with software and other hardware implementations.

For hardware implementation of FFT multiplier, optimum data representations assuring precision to produce correct products are defined based on experimental error analysis using observations that maximum error occurs when multiplying maximum values. Implementation results show that hardware FFT multipliers with optimized data representations can reduce the area by 60% and critical path delay by 26% compared with those with IEEE754 64 bit floating-point representation. A version of optimized FFT multipliers using HITACHI CMOS 0.18μm technology was found to perform multiplication of 2^5 to 2^{13} digit hexadecimal numbers 19.7 to 34.3 times (25.7 times in average) faster than software FFT multipliers at an area cost of 9.05mm^2. The hardware implementation of FFT multipliers was found to realize 35.7 times better performance than software Karatsuba multiplier with an area cost of 16.1mm^2 for 2^{21} digit hexadecimal multiplication where the software FFT and Karatsuba multipliers give the same performance. The feasibility of implementing FFT multiplication by hardware was shown by the result that an FFT multiplier for two digit hexadecimal numbers can be fabricated on a custom VLSI chip with the size of 2.8mm square.

When implementing Karatsuba algorithm by hardware, there are two alternatives: iterative and recursive approaches. In the iterative approach the algorithm is realized by sequential circuits. The architecture is called iterative Karatsuba multiplier (IKM). In the recursive approach the algorithm is realized by combinational circuits. The architecture is called recursive Karatsuba multiplier (RKM).

The implementation of RKM architecture using 0.18μm process was found to have less area cost than that of standard Wallace Tree Multiplier (WTM) for bit lengths larger than 2^9. The area cost of 2^9 bit RKM was 30mm^2. The critical path delay of RKM is always larger than that of WTM. Therefore we should use WTM instead of RKM for fundamental combinational multiplier used in IKM in order to have better performance cost ratio.

The implementation of IKM architecture was carried out with the number of recursive applications of Karatsuba algorithm varied. The designs are denoted by R1IKM, R2IKM, and R3IKM referring to the cases when the algorithm is recursively applied one, two, and three times, respectively. For each design the performance, area cost, and power consumption were evaluated by varying the length of fundamental combinational multiplier from 4 to 128 bits. The results show that the IKM designs with 32, 64, and 128 bit fundamental combinational

multiplier are 5, 10, and 30 times faster than software Karatsuba implementation (exflib), respectively. The area cost of R3IKM with 128 bit fundamental multiplier was found to be 10.9mm^2 which is the largest among IKM designs. The energy consumption of the same design was found to be 1/600 of that consumed by general-purpose processor which executes the software version.

Combining the results obtained by the study we found that the performance of FFT multiplication exceeds that of Karatsuba multiplication for digit lengths larger than 2^{23} bits (2^{21} hexadecimal digits). This holds both for software and hardware implementations. The performance of hardware implementation is 30 times better than that of software implementation for 2^{23} bit multiplication. This holds both for FFT and Karatsuba multiplication methods. The area costs of 2^{23} bit FFT multiplier and IKM were 16mm^2 and 2^{12}mm^2, respectively. External memory was not taken into account for evaluating the area costs of FFT multiplier.

To conclude, this study revealed tradeoffs of hardware and software implementations of FFT and Karatsuba multiplications for wide range of applications with respect to performance and area costs. The results obtained by this study will help system designers for applications requiring multi-digit multiplication to select design alternatives including ASIC realization. The study should contribute to research and development of multi-digit arithmetic implementations and to promoting their use in various applications.

多倍長乗算回路の構成と評価に関する研究

矢崎 俊志

概要

本論文は，一般的な汎用プロセッサのビット長を大きく上回る多倍長数の乗算をハードウェアで実現する方法について述べる．

多倍長数の演算は高精度の数値計算や素数判定，カオス計算，暗号計算など，多様なアプリケーションに利用されている．多倍長演算はその性質から，多くの時間を必要とする．特に，頻繁に利用される乗算は演算のボトルネックとなり得る．多倍長乗算においては $O(n^2)$ の筆算式乗算よりも効率よく乗算を行う様々なアルゴリズムが存在する．代表的なものとして，$O(n^{1.58})$ の Karatsuba 2-way 法，$O(n^{1.465})$ の Karatsuba 3-way 法，$O(n^{1.404})$ の Karatsuba 4-way 法，$O(n^{1.365})$ の Karatsuba 5-way 法，$O(n^{1.63})$ の 法算法，$O(n \log n \log \log n)$ の 高速フーリエ変換 (Fast Fourier Transform, FFT) 法がある．これらの中で，オーダの上で最も高速なのは FFT 法である．しかし，FFT 法は，複素数演算のオーバヘッドが大きく数百から数万ビットの演算における実質的な性能は Karatsuba 法よりも低い．このことから，現在もっとも利用されているアルゴリズムは Karatsuba 法である．ただし，数百万桁の乗算においては FFT 法の方が高速である．現在，これらの多倍長乗算はソフトウェアによって実現されている．一方，多倍長乗算のハードウェア実装に関する研究としては，ガロア体上の乗算を行うものが多く報告されているが，整数の乗算に関するものはごくわずかである．特に FFT 乗算のハードウェア実装に関する研究は知られていない．

本研究の目的は，FFT 法を用いた比較的大きな桁数の多倍長乗算と Karatsuba 法を用いた比較的小さな桁数の多倍長乗算をそれぞれハードウェア実装し，ソフトウェアとの比較を行うことでその性能やコストを明らかにすることである．

FFT 法のハードウェア実装においては，最大値どうしの乗算が最大の誤差をあたえることに着目し，乗算に必要な精度を求め，それを保証するデータ長で FFT 乗算器を CMOS 0.18μm テクノロジを用いて構成した．その結果，16 進数 2^{13} 桁の乗算において，IEEE754 の 64 ビット浮動小数点表現を用いた場合と比較して面積

を 60%，最大遅延時間を 26% 削減した FFT 乗算器を実装することができた．さらに最適なパイプライン化を行った結果，同世代のテクロノジで設計された Pentium 4 1.7GHz 上で実行した FFT 乗算と比較して 16 進数 2^5 から 2^{13} の範囲で，19.7 倍から 34.3 倍，平均で 25.7 倍の性能を実現することができた．この時の面積は 9.05mm^2 であった．また，FFT 乗算と Karatsuba 乗算の性能が逆転する 16 進数 2^{21} 桁の乗算においては 35 倍の性能を実現した．この時の面積は 16.1mm^2 であった．実際に，16 進数 2 桁の FFT 乗算器を 2.8mm 角のカスタムチップで試作し，その結果から，より大きな桁の FFT 乗算器も現実に実装可能であることを示した．

Karatsuba 乗算器の実装においては 2 つの設計選択肢として組み合わせ回路で行う RKM (Recursive Karatsuba Multiplier) と順序回路で行う IKM (Iterative Karatsuba Multiplier) を構成した．CMOS 0.18μm テクノロジを用いてこれらを実装した結果，2^9 ビット以上の乗算において RKM の面積は標準的な乗算回路である Wallace Tree 乗算器 (Wallace Tree Multiplier, WTM) より小さくなることがわかった．2^9 ビットにおける面積は 30mm^2 であった．また，最大遅延時間は常に WTM の方が短かった．このことから，性能コスト比において RKM より WTM の方が優れていることがわかった．したがって，IKM で用いる基本乗算器としては WTM を用いる方が良い．IKM に関しては，再帰の回数をそれぞれ 1，2，3 とした R1IKM，R2IKM，R3IKM を実装した．さらにそれぞれについて，基本乗算器のビット長 (基本ビット長) を 4 から 128 ビットとする IKM を実装し，その性能，面積，電力を評価した．その結果，基本ビット長を 32，64，128 ビットにした IKM は，ソフトウェア実装 exflib と比較してそれぞれ約 5，10，30 倍の性能を実現できることを示した．この時，最も大きい面積は，基本ビット長を 128 ビットにした R3IKM の 10.9mm^2 であった．同じ R3IKM について消費エネルギーを評価したところ汎用プロセッサと比較して 1/600 であることがわかった．

全体を通して，ハードウェアとソフトウェアいずれにおいても，FFT 乗算と Karatsuba 乗算は 2 進 2^{23} 桁 (16 進数 2^{21}) 付近で性能が逆転することがわかった．2 進 2^{23} 桁においてハードウェア実装とソフトウェア実装の性能比はいずれのアルゴリズムについても約 30 倍であった．またこのとき，面積はそれぞれ 2^{12}mm^2 と 16mm^2 であった．ただし，FFT 乗算器の面積に外部メモリは含まれていない．

これらの結果から，FFT 法と Karatsuba 法の両ハードウェア実装おいて，パラメータに応じた性能コスト比の変化と適用範囲が明らかになった．本論文は，広い桁範囲における多倍長乗算のハードウェア化に関する詳細な研究結果を述べた唯一のものであり，多倍長乗算を用いたアプリケーションやシステムの実現において有益な指標となる．また，多倍長演算に関する実装技術の研究や開発，およびアプリケーションシステムの利用促進に大きく寄与すると考えられる．

目次

図目次

表目次

第 1 章

はじめに

本研究は，一般的な汎用プロセッサのビット長を大きく上回る多倍長数の乗算をハードウェアで実現する方法に関するものである．

本章では，本研究の背景，目的，関連研究を述べる．本章は次のように構成されている．

第 1.1 節　本研究で扱う多倍長数とその演算の特徴について概略を述べる．

第 1.2 節　多倍長演算の中でも特に重要な乗算について，アルゴリズムとソフトウェア実装を述べる．

第 1.3 節　技術的背景と一般的な視点から見たハードウェア実装の意義について述べる．

第 1.4 節　本研究の目的を述べる．

第 1.5 節　ハードウェア多倍長乗算器の関連研究を述べる．

第 1.6 節　本研究の目的と意義をまとめ，本論文の章構成を示す．

1.1　多倍長数と演算

多倍長数とは一般的なプロセッサの 1 語長を大きく上回るビット長で表現される数である．多倍長数として整数と実数が考えられるが，実数は計算機上では本質的に整数と同様に扱うことができる．したがって，ここでは対象を整数に限定する．

以前から，多倍長の四則演算 ($+$，$-$，$\times$，$/$) は高精度の数値計算 [1] に使われてきた．今日では素数判定 [2]，カオス計算 [3, 4, 5]，暗号計算 [6] など，多様なアプリケーションに利用されはじめている．例えば，カオスを応用した画像処理システム [7] や通信システム [8] では，演算中の丸め誤差がシステム全体に大きな影響を与えることが知られている．このような場合，多倍長演算を用いることで誤差の影響を軽

減することができる．

多倍長演算を汎用プロセッサで行う場合，演算はプロセッサの 1 語長を超える．このとき，図 1.1 に示す加算の例のように，桁上げなどが語をまたぐため，ただの加減算であっても大きなオーバヘッドが発生する．乗除算についてはさらに多くの演算時間が必要となる．

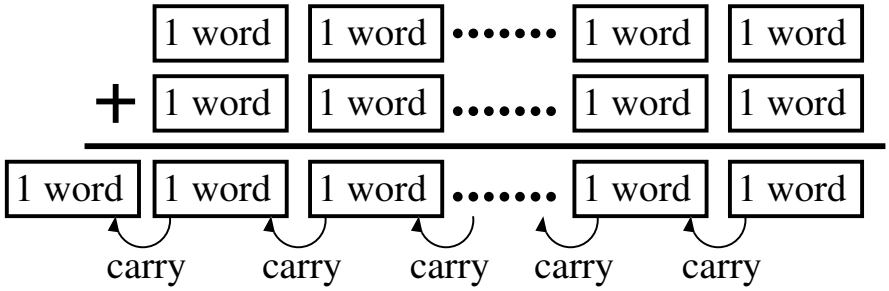

図 1.1　多倍長加算における桁上げの様子

多倍長演算を用いる多くのアプリケーションにおいて，乗算が繰り返し用いられる機会は多い．一方，除算に関しては，演算時間は乗算にくらべて長いが，使用される機会は乗算ほど多くない．また，除算は乗算を繰り返し行うことで実現することができるため乗算の高速化は除算の高速化にもつながる．よって，多倍長演算の中でも特に重要な乗算を高速化することで，多倍長演算を利用したアプリケーションの多くを高速化することができる．以上の理由から，本研究では多倍長の乗算に着目する．

1.2　多倍長乗算アルゴリズムとソフトウェア実装

前節で述べたように，多倍長乗算は多くの計算時間を必要とする．そのため，効率よい演算を行うために，様々なアルゴリズムが開発されてきた [9]．

1963 年には Karatsuba らによって，整数の平方を効率良く求めるアルゴリズムが提案された．これを乗算に応用したものが Karatsuba 法 [10] として知られている．Karatsuba 法は整数の乗数と被乗数をそれぞれ半分に分割し，上位部分と下位部分を係数とする 2 次多項式を作る．この 2 個の 2 次多項式から積となる 4 次多項式の係数を決定する．ソフトウェアにおいては，1 語長になるまで再帰的に分割を行うことで，乗算の演算時間を $O(n^{1.58})$ $(\log_2 3 \approx 1.58)$ まで削減することができる．この方法は 2-way 法とも呼ばれている．この方法をさらに拡張し，分割を半分ではなく 1/3，1/4，1/5 と行う方法もある．これらの方法は 3-way 法，4-way 法，5-way 法と呼ばれ，それぞれ演算時間は $O(n^{1.465})$ $(\log_3 5 \approx 1.465)$，$O(n^{1.404})$ $(\log_4 7 \approx 1.404)$，$O(n^{1.365})$ $(\log_5 9 \approx 1.365)$ となる．この方法は分割数を増やす

程，演算時間のオーダを小さくすることができるが，細かい分割に伴って，多項式の係数を決定する処理が複雑になるためトレードオフが存在する．一般には，2-way 法 と 3-way 法がよく用いられる．

これをさらに応用したものに Toom-Cook 法 [11, 12] がある．この方法は，1963 年に Toom によって基本的な考え方が提案され，1966 年に Cook により計算機での利用法が示された．この方法は，乗算桁数が大きい時ほど分割数を増やし，さらに，積を表す多項式の係数を決定するとき，降べきの差分を利用した決定法を用いる．この方法は本質的に Karatsuba 法と同じであり，まとめて Karatsuba 法と呼ばれることもある．

1966 年には Schönhage らによって，高速な法演算を用いることで演算時間を $O(n^{1.63})$ ($\log_3 6 \approx 1.63$) とする乗算法が提案された [13]．また，1971 年に同じく Schönhage らによって高速フーリエ変換 (Fast Fourier Transform，FFT) を用いた FFT 乗算法が提案された [14]．この方法は FFT を用いることで，演算時間を $O(n \log n \log\log n)$ まで削減できることが知られている．

また，100 ビット程度の乗算には，$O(n^2)$ の単純な乗算が用いられることもある．

前述の乗算法の中で，オーダの上で最も高速なのは FFT 法である．しかし，FFT 法は，オーダの上では最速であるが，実数の演算を必要とするため，FFT 自体のオーバーヘッドが大きい．例えば，数百から数万ビットの演算における実質的な性能は Karatsuba 法よりも低い [15]．しかし，円周率の計算に用いられるような数百万ビットの演算では，オーバヘッドが相対的に小さくなるため，FFT 乗算は他の演算法と比較して高速である．このように，実質的な性能を考えた場合，アプリケーションに必要な乗算桁数に応じて適切なアルゴリズムが用いられる．これを表 1.1 にまとめる．表には法演算による乗算が示されているが，Karatsuba 法と比較して性能

表 1.1　多倍長乗算アルゴリズムの比較

適用桁数	アルゴリズム	計算量
100 ビット程度	筆算式乗算	$O(n^2)$
～数十万ビット	法演算による乗算	$O(n^{1.63})$
	Karatsuba 乗算 (2-way)	$O(n^{1.58})$
	Karatsuba 乗算 (3-way)	$O(n^{1.465})$
	Karatsuba 乗算 (4-way)	$O(n^{1.404})$
	Karatsuba 乗算 (5-way)	$O(n^{1.365})$
～数百万ビット	FFT 乗算	$O(n \log n \log\log n)$

が低く，アルゴリズムが複雑であることからあまり利用されていない．

現在，多倍長乗算はソフトウェアによって実現されている．代表的なものとして Karatsuba 法の実装である exflib[16] や，GNU MP[17] などが知られている．また，FFT 法を用いた乗算は，既存の FFT ライブラリを用いて実現することができる．高速な FFT ライブラリとしては FFTW[18] が知られている．これらの多倍長乗算ライブラリや FFT ライブラリは，実行するアーキテクチャに応じた最適化が行われるなど，様々な工夫がされており，現在のデファクト・スタンダードとして利用されている．

1.3　ハードウェア実装とその意義

本節では，はじめに技術的背景を述べたあと，2 つの視点 (ソフトウェアとの協調，計算量とコスト) からハードウェア実装とその意義について述べる．

1.3.1　汎用プロセッサと専用ハードウェアに関する技術的背景

ここではまず，ソフトウェアを実行する汎用プロセッサと専用ハードウェアの係わりについて述べる．汎用プロセッサは，1970 年代前半に登場して以来，世の中の様々な分野で利用されるようになり，より高い処理能力が要求されていった．それに伴い，図 1.2 に示すように，より長いビット長を持つ高性能な汎用プロセッサが開発されてきた．近年では数世代前のスーパコンピュータに匹敵する処理能力を持つ 32

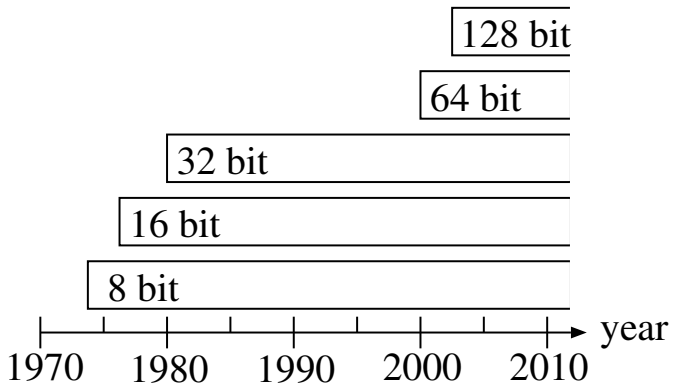

図 1.2　汎用プロセッサにおけるビット長の変遷

ビットから 64 ビットのプロセッサが登場している．特にゲーム機などをはじめとするマルチメディアに特化した製品には 128 ビットプロセッサが用いられるようにまでなった．

一方，時代と共に世の中で必要とされるアプリケーションは変化し，それに伴って，汎用プロセッサを含んだ複雑な計算機システムが多く設計され，社会の中でより重要な役割を担うようになってきた．計算機システムが登場した当時，その主な用途は比較的単純な数値計算やトランザクション処理であったが，近年では専門家の研究道具や，社会の重要なサービスを提供するサーバとしても利用さている．また，モバイル機器など，多くの組み込み機器には，比較的小さな 8 ビットや 16 ビットの汎用プロセッサが用いられている．したがって，現在の計算機システム設計においては，高性能，高い安全性，低実装コスト (面積)，低消費電力がより重要となっている．

高い性能をもつシステムを構築する 1 つの方法として，ある特定の演算に特化した VLSI (Very Large Scale Integrated circuit) を用いる方法がある．このように，特定の処理に特化した専用の VLSI は ASIC (Application Specific Integrated Circuit) と呼ばれている．ASIC をシステムに利用する方法としては，大きく分けて 2 つの方法がある．

1 つ目は，図 1.3 のように，汎用プロセッサの代わりに専用ハードウェアを用いる方法である．汎用プロセッサは大量に生産されるため安価である一方，汎用であるた

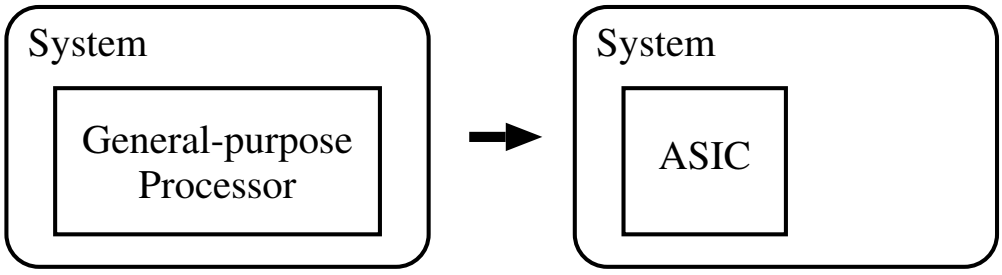

図 1.3　ASIC による汎用プロセッサの置き換え

めチップサイズが大きく，加えて要求に応じたカスタマイズが難しい．したがって，アプリケーションによっては適切な汎用プロセッサを探すのがむずかしい場合がある．このような場合，汎用プロセッサの代わりに，高価ではあるが高性能かつ低面積である ASIC を用いることで最適なシステムを設計することができる可能性がある．また，小型の ASIC を用いることで，システムの消費電力を削減することも可能である．具体的な例として，ネットワーク機器のスイッチング，ルーティング，プロトコルなどの専用ハードウェア化があげられる．

2 つ目は，図 1.4 に示すように，既存の汎用プロセッサと ASIC を組み合わせて用いることで，より高性能なシステムを実現するという方法である．このような構成は，汎用プロセッサ上のソフトウェアと ASIC が協調することにより比較的低コストで高性能なシステムを構築することができる．半面，汎用プロセッサと

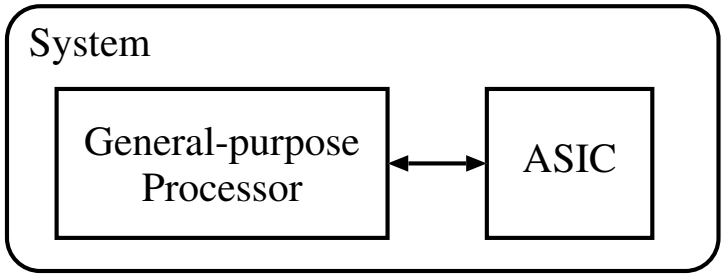

図 1.4　汎用プロセッサと ASIC の協調

ASIC を効率良く使い分けるための工夫が必要である．例として，パーソナル・コンピュータ (Personal Computer, PC) のグラフィック処理における GPU (Graphics Processing Unit) がある．近年の PC の中にはマルチメディア機能を充実させるためにグラフィックス性能をセールスポイントにしているものもあり，このような場合には画像処理を専用のチップで行う方法が有効である．

1.3.2　ハードウェアとソフトウェアの協調の視点から

計算機システムの設計において，設計するシステムに対する要求は様々である．もし，高性能なシステムを実現したいのであれば，システムの大部分をハードウェアで実現すればよい．また，低コストを実現したいのであれば，汎用プロセッサを用いて，ソフトウェアで実現すればよい．しかし，多くのシステムはこのような極端な要求に基づいて設計されるのではなく，性能コスト比が最適になるように設計されることが望ましい．そのためには，システムの構成要素を適切に区分し，特に処理のボトルネックとなる部分のみをハードウェアで実現するなどの工夫が必要である．この様子を図 1.5 に示す．本研究では，このボトルネックとして，多倍長の乗算を取り上げている．このように，ソフトウェアとハードウェアを組み合わせることで，システム全体のスループットを比較的低コストで向上させることができる．このような設計手法は，ソフトウェアとハードウェアの協調設計 (Software / Hardware Co-design) として知られており，今日の計算機システムではこのような手法が採用されている例は少なくない．

実際のシステム設計においてこのような協調設計を行うためには，ボトルネックを適切に判断し，許容されたバジェット (budjet) 内で効率良くハードウェア部分を構成する方法の検討が必要になる．

近年，ソフトウェアの設計やその実装は機能ブロックごとに行われることが多く，開発の初期段階でシステムのボトルネックを特定することは難しくない．しかし，

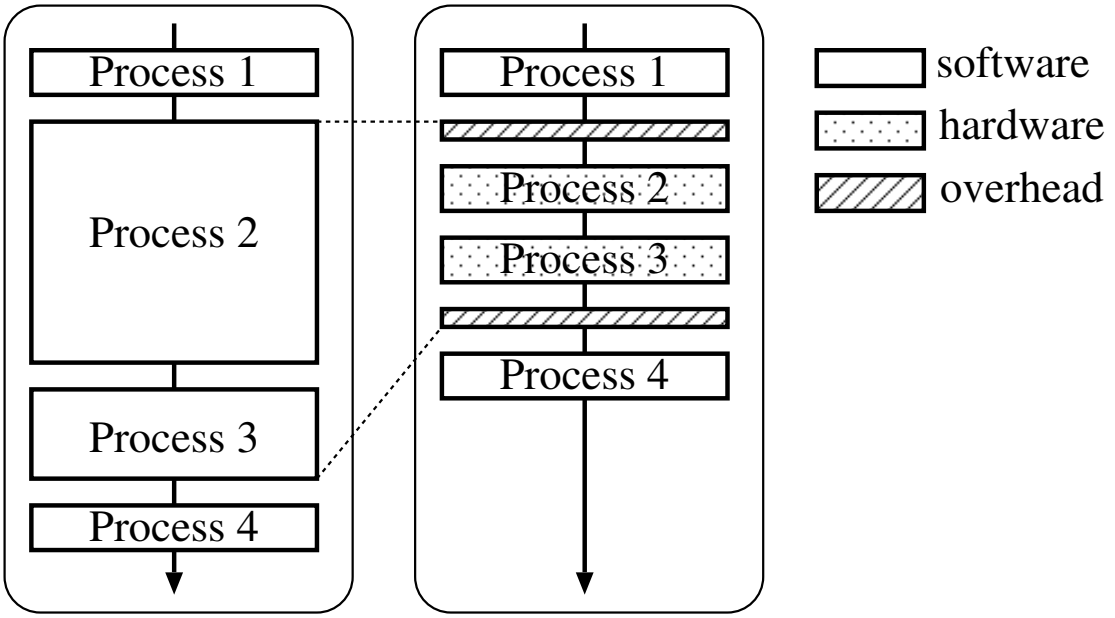

図 1.5　ソフトウェアとハードウェアの協調

ハードウェア部分の構成方法についてはその選択が多岐にわたる場合が多い．よって，様々なアルゴリズムやハードウェア・アーキテクチャに対して多くの視点から検討を行い，適切なものを選ぶためには非常に多くの時間を必要とする．

このような要求に対して，多くのアプリケーションで有用な機能や演算をハードウェア化し，その結果を他の実装と比較，評価する研究は，学術的，産業的に有益であると考えられる．

1.3.3　計算量とコストの視点から

本節では，計算量から見たソフトウェア実装とハードウェア実装の違いについて議論し，この視点から多倍長乗算ハードウェア実装の意義について述べる．

ソフトウェアによるアルゴリズムの実装とは，あらかじめ与えられた演算器資源を繰り返し用いて，逐次的な演算を実行することである．一方，ハードウェアによる実装では，演算器資源は固定されず設計者が自由に設定できるため，1 つの演算器を繰り返し用いる逐次的な処理の他に，演算器を複数用いて並列的な処理を行うことができる．演算器の数を増やして並列的な演算を行った場合，計算量は演算時間と回路面積の両方に現れる．

これに関して，乗算を例にして具体的に考える．n 桁の乗算を最も単純な筆算式の乗算アルゴリズムを用いてソフトウェアで実装し，固定ビット長の乗算器を持った汎用プロセッサで実行すると，$O(n^2)$ の計算時間を要する．これに対して，加算器を複

数用意して組み合わせ回路で実現した n 桁の Wallace Tree 乗算器 [19] は n 個の部分積を木構造で累算するため，加算器 $\log n$ 段分の遅延時間を要する．したがって，演算時間は $O(\log n)$ である．しかしこの時，加算器を増やした分，面積は $O(n^2)$ となる．このように，逐次的に演算を行うソフトウェアでは，計算量が時間に反映されるのに対して，並列的な演算が可能なハードウェアにおいては，特に組み合わせ回路を用いた場合に，計算量が面積コストに反映される．このことから，ハードウェア実装とは，計算資源を追加することで計算量を面積コストに反映し，その分，演算を高速化する実装方法であるという見方もできる．なお，基本的な加算器と乗算器の構成は，それぞれ付録 A と B で述べる．

ソフトウェアにおける工夫は，与えられた資源を有効に用いるという点では合理的である．一方，ハードウェアによる実装は先に述べたように新たな計算資源を追加し，計算量を面積コストに反映させることで演算の高速化を行うというアプローチである．多倍長演算の概念が考え出された 1960 年代や 70 年代は計算機システムにおいて，ハードウェア資源は非常に貴重なものであった．また，計算機が広く一般に普及した現代とは違い，計算機自体が高価であったため，ある特定の演算を高速化するためだけに新たなハードウェアを追加することはシステムを構成するために許容されたバジェットの多くを消費してしまうことになり，性能コスト比が悪く，あまり良い方法ではなかった．しかし，近年，計算機が世の中に浸透するにしたがって，計算機システム全体のコストが劇的に下がったため，新たなハードウェア追加に対するコスト的な障壁は減少している．加えて，実装技術の進歩により，比較的安価にハードウェアを設計することができるようになった．特に，回路をプログラムすることが可能な FPGA (Field Programmable Gate Array) の登場により，個人レベルでも実用的な専用ハードウェアの開発が可能になった．

1.4　本研究の目的

これまでに述べてきた背景において，今後より多くのアプリケーションに利用され，より高い性能が要求されるであろう多倍長乗算をハードウェアで実装するという選択肢は，より現実的なものとなっている．また，小さな桁から大きな桁までの広い範囲の乗算を可能にするためにも，標準的な多倍長乗算に用いられている Karatsuba 法と，それ大きく上回る桁数の乗算に用いられている FFT 法の両アルゴリズムに関する検討が必要である．

そこで本研究では，FFT 法と Karatsuba 法に注目し，それらをハードウェアで実装することを目的とする．実装にあたり，最適化したハードウェアを構成し，これらの性能とコストを求める．また，他の実装との比較においてこれらを評価する．

FFT 法を対象とした理由として，前述の意義に加え，FFT ハードウェアがすでに世の中で DSP (Digital Signal Processing) などに利用されている点があげられる．これにより，既存の FFT ハードウェアが存在する環境においては，ハードウェア資源の追加が少なくて済むと考えられる．また，Karatsuba 法を対象とした理由として，他の乗算法に比べてアルゴリズムが単純である点があげられる．ハードウェアの構成には様々な面でソフトウェアよりも強い制約がある．したがって，アルゴリズムは可能な限り規則的かつ単純であることが望ましい．Karatsuba アルゴリズムの詳細は後述するが，比較的単純であり，かつ規則的な再帰によって性能コスト比を変えることができる．

1.5　ハードウェア多倍長乗算器の関連研究

本節では，ハードウェアで実現された多倍長乗算器に関する研究を紹介する．図 1.6 に既に報告されているハードウェア多倍長乗算器の分類をまとめる．

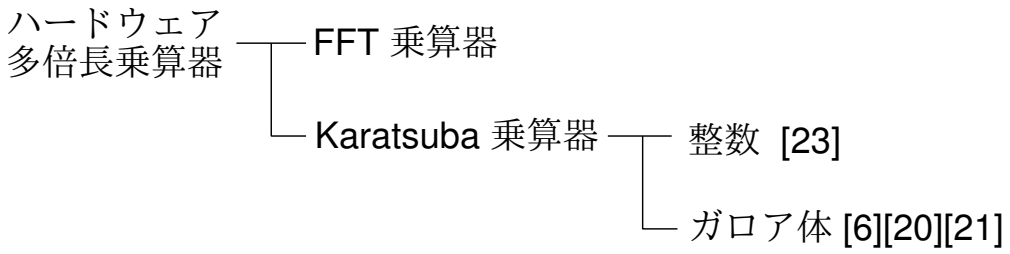

図 1.6　ハードウェアによる多倍長乗算器の分類と報告例

現在までに FFT アルゴリズムをハードウェア実装した例は多いが，FFT を多倍長乗算に利用する目的でハードウェア実装した研究は知られていない．

Karatsuba 乗算器には，整数の乗算を行うものと，ガロア体 (Galois Field，GF) 上で乗算を行うものの 2 種類ある．ガロア体とは有限個の要素からなる体 (Field) である．したがって，そこで定義された演算は四則演算に対して閉じており，要素を 2 進数値にマッピングすることでハードウェアによる演算器を容易に構成することができる．一般に用いられているガロア体は 2 個の要素からなるガロア体 $GF(2)$ の要素を 1 ビットの値として 0 と 1 にマッピングする．さらに，$GF(2)$ 上で定義される既約多項式を用いてガロア拡大体 $GF(2^w)$ を定義する．ただし，w は自然数である．このガロア拡大体 $GF(2^w)$ の要素は w ビットの 2 進数値にマッピングすることができる．単にガロア体と言った場合は，このガロア拡大体を意味することがある．このように定義したガロア体上で，加算と乗算を定義すると，加算は排他的論

理和，乗算は論理積で定義することができ，よって，ハードウェアによる演算器を容易に構成することができる．このガロア体上の Karatsuba 乗算器に関する研究は多数報告されている．

文献 [20] では，Karatsuba アルゴリズムを再帰的に適用する際，1 度目の適用では 2 分割した乗数と被乗数を，2 回目の適用では 3 分割している．このように，分割数を混合にすることで，演算コストをさらに削減した Hybrid Karatsuba 乗算器を設計した．文献 [21] では，Hybrid Karatsuba 乗算器をさらに発展させた Ordered Karatsuba 乗算器と Padded Karatsuba 乗算器が設計されている．Ordered Karatsuba 乗算器は，文献 [22] の結果を引用して分割数の組み合わせを最適化した Hybrid Karatsuba 乗算器である．Padded Karatsuba 乗算器は，乗数と被乗数のビット長を，2 や 3 などの小さな素数の積で表すことができるビット長にパディング (Padding) した上で，再帰的な分割を行う Karatsuba 乗算器である．文献 [6] では，Karatsuba アルゴリズムの再帰部分をループ展開し，順序回路で乗算器を構成することで面積と消費電力を削減した Iterative Karatsuba 乗算器を設計した．これらの乗算器は，ガロア体上の乗算を対象としているため，整数乗算器とは演算の定義が根本的に異なる．よって，これらの Karatsuba 乗算器と整数の Karatsuba 乗算器を性能やコストの面で単純に比較することはできない．しかし，ハードウェア設計において参考にできる部分はある．

整数 Karatsuba 乗算器に関しては，2 分割の Karatsuba アルゴリズムを 1 度だけ適用して 32 ビットの 整数 Karatsuba 乗算を実装した例が報告されている [23]．これらを表 1.2 にまとめる．本研究では，実装例の少ない整数の乗算を対象としている．

表 1.2　ハードウェア Karatsuba 乗算器に関する先行研究

数体系	著者	アルゴリズム	ビット長	分割数
ガロア体	Grabbe ら [20]	Hybrid Karatsuba	233	2, 3
	Cheng ら [21]	Ordered Karatsuba Padded Karatsuba	113	2, 3 2 分割を 6 回
	Dyka ら [6]	Iterative Karatsuba	128，64，32	2
整数	柴岡ら [23]	Karatsuba	32	2

本論文は，現時点において，FFT 乗算器のハードウェア構成法について実装と評価を行った唯一の例であり，Karatsuba 乗算器の構成法について，文献 [23] で示されているものよりさらに詳細な結果を示すものである．

1.6 まとめと本論文の構成

これまでの議論から，本研究の目的と意義をまとめる．本研究では，多くの分野への応用が期待できる整数の乗算に関して，FFT 法を用いる比較的大きな桁数の多倍長乗算から Karatsuba 法を用いる比較的小さな桁数の多倍長乗算について，それぞれをハードウェアで実装し，その性能やコストを明らかにする．さらにこれらを既存のソフトウェア実装やハードウェア実装と比較する．これは，今まで明らかでなかったハードウェアによる整数多倍長乗算に関する新しい知見となり，将来，重要性が増すであろう高速な多倍長乗算を用いたアプリケーションをシステムとして実装する際の有益な指標になると考える．

本論文は次のように構成されている．

第 2 章 一般的なハードウェア設計手法および評価手法の概略を述べ，さらに，本研究における実験環境と評価手法を示す．

第 3 章 FFT 法を用いたハードウェア多倍長乗算器を設計し，性能とコストを評価する．また，ソフトウェア FFT 乗算と性能を比較し，その結果を示す．

第 4 章 Karatsuba 法を用いたハードウェア乗算器を設計し，性能とコストを評価する．また，ソフトウェア Karatsuba 乗算と性能を比較し，その結果を示す．

第 5 章 第 3, 4 章までの結果を踏まえ，FFT 乗算器と Karatsuba 乗算器の関連をまとめ，全体的な視点から比較を行う．また，その結果に基づいて，桁数に応じた多倍長乗算器の構成法を示す．

第 6 章 本論文をまとめ，多倍長演算に関する研究の今後について述べる．

第 2 章

ハードウェア設計方法

本章では，本論文におけるハードウェア設計の流れや評価の方法とその妥当性を理解するため，一般的なハードウェア設計手法および評価手法の概略について述べる．さらに，本研究における実験環境と評価手法を示す．

2.1 一般的なハードウェア設計手法および評価手法

現代のハードウェア設計はおおむねトップ・ダウンの設計が基本であり，図 2.1 に示すように，その設計フローは大まかに 5 つのレベルに分けられる．各レベルは設

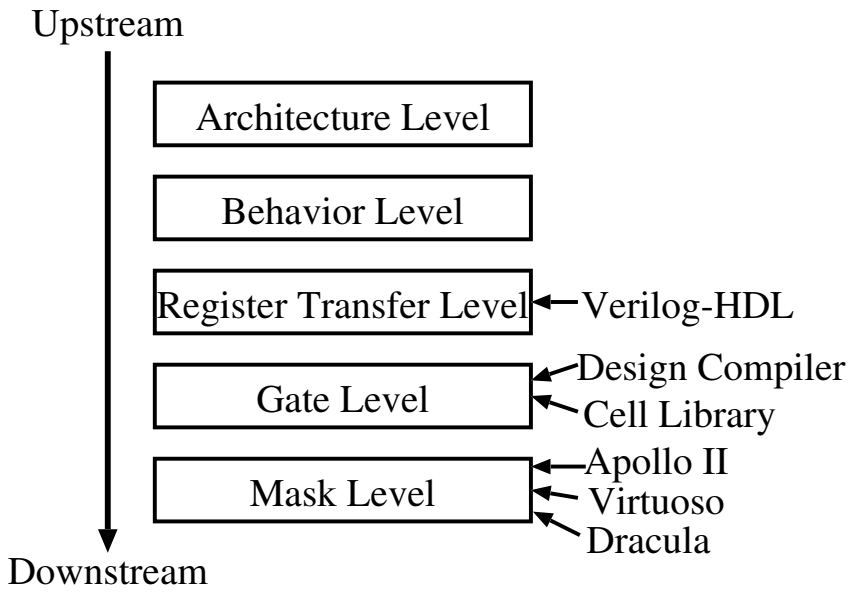

図 2.1 ハードウェア設計の流れ

計の上流 (Upstream) 工程から下流 (Downstream) 工程に向かって順番に，

- アーキテクチャ・レベル (Architecture Level)
- ビヘイビア・レベル (Behavior Level)
- レジスタ・トランスファ・レベル (Register Transfer Level)
- ゲート・レベル (Gate Level)
- マスク・レベル (Mask Level)

と呼ばれる．図にはそのレベルにおいて実装や評価に用いられるツールも示した．各レベルにおける設計の詳細は後述する．

設計は基本的に下流であるほど多くの時間を必要とする傾向にある．しかし，設計の内容がより具体的になるため，下流で行われる評価ほど，正確であると言える．上流での評価は精密さの面では下流での評価に及ばないが，大きな問題を初期段階から低コストで発見できるという点では重要である．次にそれぞれのレベルで行われる設計の内容と主だった評価方法を述べる．

2.1.1 **アーキテクチャ・レベル** (Architecture Level)

目的に則したアルゴリズムの候補を挙げ，計算量やソフトウェア実装による予備実験結果をもとに適切なものを選択する．

2.1.2 **ビヘイビア・レベル** (Behavior Level)

選択したアルゴリズムをどのような構成で実装するか検討する．設計工程や機能ブロックを明確にするためにモジュール分けを行い，これから設計するハードウェアが全体としてどのように動作するのかを確認する．この段階で回路設計を行い，回路図を作成する．

近年では，従来，配線基板上に構成していた複数の LSI を 1 つのチップに集約する SoC (System on a Chip) による設計が盛んである．このような大規模回路の構成には System C をはじめとしたシステム設計用の言語が用いられることがある．これらは C++ などの高級プログラミング言語の拡張として提供されており，効率良く大規模システムの設計や検証を行うことができる．

2.1.3 **レジスタ・トランスファ・レベル** (Register Transfer Level)

各モジュールに必要な，演算器やレジスタ等の記憶素子を定義し，各演算器やレジスタに対して信号がどのようにやりとりされるのかを記述する．記述にはハードウェア記述言語 (Hardware Description Language，HDL) が用いられる．代表的な

HDL として，Verilog-HDL や VHDL がある．これらは細かな違いはあるものの，どちらも実際の設計で同じくらい頻繁に用いられている．他のレベルで用いる設計ツールによっては，どちらかにしか対応していないものもあるため，ツールによって HDL を選択することもある．なお，上記のシステム記述言語も HDL に含まれる．

このレベルの設計において，あらかじめ実装され，テストされている演算器などをライブラリ化したものを用いることがある．これらは IP (Intellectual Property) 部品と呼ばれ，ソフトウェアのライブラリと同様に様々なメーカや団体から提供されている．信頼性の高い IP を用いることにより，実装やテストの手間を省き，効率の良い設計を行うことができる．

2.1.4 **ゲート・レベル** (Gate Level)

AND や OR などの論理ゲート・レベルでの設計を行う．このレベルにおいて，目的の回路をゲート単位で最初から設計することは現在ではほとんどなく，多くの場合は，HDL のソースファイルから半自動で論理ゲート・レベルの回路図を生成する．この作業を論理合成と呼ぶ．論理合成に用いられるツールの多くは，合成の条件を指定することが可能である．論理合成の結果得られた回路図をネットリスト (Netlist) と呼ぶ．ネットリストにはゲート間の配線が記述されているが，物理的な配置は考慮されていない．したがって，配線による遅延は考慮されていない．

ゲート・レベルの設計に用いる AND や OR などのゲートはプロセス・ルールやテクノロジによって構造が異なる．したがって，論理合成時には，このゲートをライブラリ化したセル・ライブラリ (Cell Library) と呼ばれるものを用いる．セル・ライブラリには，各ゲートの遅延時間，面積，キャパシタンスなどの情報が含まれる．よって，このセル・ライブラリを用いて合成されたネットリストから大まかな遅延時間，面積，消費電力を見積もることができる．多くの設計においてこのレベルで回路の善し悪しを評価する．

FPGA への実装を目的とした研究では，面積の代わりに，FPGA 内部において基本論理を実現するための素子である LE (Logic Element) を用いた評価を行っているものもある．

2.1.5 **マスク・レベル** (Mask Level)

このレベルでは，ゲート・レベル設計で作成されたネットリストを元に，半導体のマスク・パターン (Mask Pattern) を設計する．マスク・パターンの作成を行うためには，ネットリストに含まれる各ゲートの物理的な配置が決定していなければならな

い．よって，このレベルで各ゲートの物理的な配置を行い，ゲート間の配線やクロック・ツリー (Clock Tree) の作成を行う．そのため，このレベルで回路を評価することで，作成されるチップにより近い性能や面積を見積もることができる．しかし，配置配線やクロック・ツリーの生成には論理合成より多くの時間が必要となる．この作業もほとんどの場合，自動配置配線ツールを用いて半自動で行う．

作成したマスク・パターンは半導体回路の製造に使われる．カスタムチップを試作する場合にもこのマスク・パターンが必要である．よってこの時点で，マスク・パターンはデザイン・ルール違反がないことを厳密にチェックされる．このチェック作業を DRC (Design Rule Check) と呼ぶ．

2.2　本研究における実験環境と評価手法

ここでは，第 2.1 節で述べた一般的な設計手法を踏まえ，本研究における設計環境と手法，および評価手法を示す．

2.2.1　設計環境と手法

本研究における設計環境と手法は次のとおりである．

アーキテクチャ・レベルにおいては，これに関して，既に第 1.3.3 節で FFT 法や Karatsuba 法の検討を行った．

ビヘイビア・レベルにおいては，設計するシステムの規模がそれほど大きくないため，システム設計言語を用いずに，手作業による設計を行う．

レジスタ・トランスファ・レベルにおいては，他の設計レベルにおいて用いるツールに合わせ，HDL として Verilog-HDL [24] を用いる．また，信頼性のある設計を行うため，加算器や乗算器には IP 部品を用いる．これらの IP 部品は，Synopsys 社 [25] から提供されている Design Ware と呼ばれるライブラリに含まれている．

ゲート・レベルにおいては論理合成ツールとして, Synopsys 社 の Design Compiler を用いる．セル・ライブラリは VDEC (VLSI Design and Education Center)[26] で作成された CMOS 0.18μm と CMOS 0.35μm テクノロジで設計されたものを用いる．

マスク・レベルの設計においては配置配線ツールとして Synopsys 社の Apollo II と Cadence 社 [27] の Virtuoso を用いる．また，DRC には Cadence 社の Dracula を用いる．

2.2.2 評価手法

本研究における評価は，主にゲート・レベルによるものである．Verilog-HDL で記述した回路を Design Compiler による論理合成でネットリストにする．得られたネットリストとセル・ライブラリに登録されたパラメタから，回路面積，遅延時間，電力を見積もることができる．ただし，前述のように，配線による遅延や面積増加は考慮されていない．このようなゲート・レベルにおける評価は多くのハードウェアに関する研究で行われており，実績のある評価手法であるといえる．また，必要な場合は，マスク・レベル設計によるチップの試作とその評価も行う．

第 3 章

高速フーリエ変換 (Fast Fourier Transform, FFT) 法によるハードウェア多倍長乗算器

本章では FFT 法によるハードウェア多倍長乗算器の実装とその評価について述べる．本章の構成は次の通りである．

第 3.1 節 FFT 法による乗算アルゴリズムのハードウェア実装について，その意義を述べる．

第 3.2 節 FFT 乗算アルゴリズムを説明する．

第 3.3 節 最も基本的な FFT アルゴリズムである Cooley Tukey FFT について説明し，過去に提案されたその他の FFT アルゴリズムを紹介する．

第 3.4 節 FFT 乗算と誤差の関係について述べ，実装コスト最適化のために必要な最小限の計算精度を求める．

第 3.5 節 FFT 乗算器の設計を述べる．

第 3.6 節 実装した FFT 乗算器の評価を行う．

第 3.7 節 FFT 乗算器のカスタムチップを試作し，その結果を述べる．

第 3.8 節 本章の内容をまとめる．

3.1 FFT 法による乗算のハードウェア実装の意義

FFT 法による乗算は桁数 n に対して $O(n \cdot \log n \cdot \log\log n)$ の計算量で乗算を行うことができる．これをハードウェアで実装することで，高速な演算を実現する．

本章で述べる FFT 乗算器は固定された演算器を繰り返し用いて乗算を実現する点

において，ソフトウェアと同じアプローチで実装する．したがって，計算量は計算時間に表れ $O(n \cdot \log n \cdot \log\log n)$ となる．しかし，専用ハードウェアであるため，ソフトウェアと比べて繰り返しの制御が単純になり，最適なスケジューリング，内部演算の並列化，パイプライン化が行いやすくなる．これにより，ソフトウェアよりも高速な乗算を実現することができると考えられる．これが FFT 乗算をハードウェア実装する大きな意義の 1 つである．

FFT 乗算の欠点として計算の複雑さがあげられる．これは FFT の演算の複雑さに起因する．FFT では浮動小数点演算などの実数演算が必要であるため，演算のオーバヘッドが大きくハードウェア化の際，面積コストが大きくなる傾向にある．また，実数演算によって誤差の問題も発生するため，FFT を乗算に用いる際は特に精度に注意する必要がある．本章では FFT 乗算の演算中に発生する誤差を見積もることで，乗算桁数に応じて必要となる最低限の精度を明らかにする．この精度で演算を行うことで最適な FFT 乗算器を構成する．この工夫により，ハードウェア化による面積の増加を最小限に抑えた FFT 乗算器を実現することができる．さらに，カスタムチップを試作することで乗算器が現実的なサイズのチップに実装可能であることを示す．このように，本章で述べる FFT 乗算ハードウェア実装のもう 1 つの意義は，FFT 乗算に必要な最小限のハードウェアコストを評価することである．

FFT は以前から DSP などで頻繁に利用されてきたアルゴリズムである．よって，FFT そのものやそのハードウェア実装に対しては多くの工夫がなされてきた．FFT 乗算における処理の大半は FFT であるため，これらの工夫を生かした実装が可能である．FFT を乗算に用いることの大きな利点はここにある．場合によっては，システムに元から組み込まれている FFT ハードウェアをそのまま流用することで，わずかなハードウェアの追加で高速な多倍長乗算器を実現することもできる．本来，FFT 乗算は非常に大きな桁においてのみ有効であったが，このハードウェアの追加量が小さければ，比較的小さな桁においても安価なハードウェア多倍長乗算器としての利用も考えられる．FFT 乗算に必要なハードウェアコストの評価はこの点においても意義がある．

3.2　FFT 乗算アルゴリズム

本節では FFT 乗算アルゴリズムを説明する．N 個の複素数からなるベクトル $x = (x_i)_{i=0,1,\ldots,N-1}$，$y = (y_i)_{i=0,1,\ldots,N-1}$ に対する離散フーリエ変換 (Discrete Fourier Transform, DFT) $X = (X_i)_{i=0,1,\ldots,N-1}$，$Y = (Y_i)_{i=0,1,\ldots,N-1}$ は，それ

ぞれ式 (3.1)，(3.2) で表される．

$$X_i = \sum_{k=0}^{N-1} x_k W_N^{ik} \tag{3.1}$$

$$Y_i = \sum_{k=0}^{N-1} y_k W_N^{ik} \tag{3.2}$$

式中の W_N^{ik} は回転子と呼ばれ，次式で定義される．ここに，j は虚数単位を表す．

$$\begin{aligned} W_N^{ik} &= e^{-j2\pi ik/N} \\ &= \cos(\frac{-2\pi ik}{N}) + j\ \sin(\frac{-2\pi ik}{N}) \end{aligned} \tag{3.3}$$

ここで，X_i と Y_i の積を H_i とする．すなわち，

$$H_i = X_i \times Y_i. \tag{3.4}$$

式 (3.4) に 式 (3.1)，(3.2) を代入すると次式を得る．ただし，ここでは W_N^{ik} を W^{ik} と略記する．

$$\begin{aligned} H_i &= X_i \times Y_i \\ &= \sum_{k=0}^{N-1} x_k W^{ik} \times \sum_{k=0}^{N-1} y_k W^{ik} \end{aligned} \tag{3.5}$$

$$\begin{aligned} &= \left\{ W^{i\times 0} x_0 + W^{i\times 1} x_1 + \cdots + W^{i\times (N-1)} x_{N-1} \right\} \\ &\times \left\{ W^{i\times 0} y_0 + W^{i\times 1} y_1 + \cdots + W^{i\times (N-1)} y_{N-1} \right\} \end{aligned} \tag{3.6}$$

$$\begin{aligned} &= W^{i\times 0} \left\{ x_0 y_0 + x_1 y_{N-1} + \cdots + x_{N-1} y_1 \right\} \\ &+ W^{i\times 1} \left\{ x_0 y_1 + x_1 y_0 + x_2 y_{N-1} + \cdots + x_{N-1} y_2 \right\} \\ &+ W^{i\times 2} \left\{ x_0 y_2 + x_1 y_1 + x_2 y_0 + x_3 y_{N-1} + \cdots + x_{N-1} y_3 \right\} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ &+ W^{i\times (N-2)} \left\{ x_0 y_{N-2} + x_1 y_{N-3} + \cdots + x_{N-1} y_{N-1} \right\} \\ &+ W^{i\times (N-1)} \left\{ x_0 y_{N-1} + x_1 y_{N-2} + \cdots + x_{N-1} y_0 \right\} \end{aligned} \tag{3.7}$$

$$= \sum_{l=0}^{N-1} \left\{ \sum_{k=0}^{N-1} x_k \times y_{(\ (l-k)\ \bmod\ N)} \right\} W^{il} \tag{3.8}$$

したがって，$H = (H_i)_{i=0,1,\ldots,N-1}$ は，

$$h_i = \sum_{k=0}^{N-1} x_k \times y_{(\ (i-k)\bmod N\ )} \tag{3.9}$$

で定義されたベクトル $h = (h_i)_{i=0,1,\ldots,N-1}$ の DFT に等しい．すなわち，ベクトル h はベクトル H の逆離散フーリエ変換 (Inverse Discrete Fourier Transform, IDFT) である．

ここで $N = 2n$ とおき，x と y の後半をすべて 0 にするような条件

$$x_i = y_i = 0 \qquad (i = n, n+1, \ldots, 2n-1)$$

を与える．すると，h は

$$\begin{array}{lcl}
h_0 & = & x_0y_0 \\
h_1 & = & x_0y_1 + x_1y_0 \\
 & & \vdots \\
h_{n-2} & = & x_0y_{n-2} + x_1y_{n-3} + \cdots + x_{n-2}y_0 \\
h_{n-1} & = & x_0y_{n-1} + x_1y_{n-2} + \cdots + x_{n-2}y_1 + x_{n-1}y_0 \\
h_n & = & \quad + x_1y_{n-1} + \cdots + x_{n-2}y_2 + x_{n-1}y_1 \\
 & & \vdots \\
h_{2n-3} & = & x_{n-2}y_{n-1} + x_{n-1}y_{n-2} \\
h_{2n-2} & = & x_{n-1}y_{n-1} \\
h_{2n-1} & = & 0
\end{array}$$

となり，これは

$$x = \sum_{i=0}^{N-1} x_i r^i \tag{3.10}$$

$$y = \sum_{i=0}^{N-1} y_i r^i \tag{3.11}$$

としたときの積 $x \cdot y$ を表している．ここに r は基数である．

この乗算法に用いられる DFT と IDFT は，それぞれ $2n$ 桁のベクトルと，サイズが $2n \times 2n$ となる回転子のテーブルを行列として掛けるという計算を行うため，$(2n)^2$ 回の乗算が必要になる．また，式 (3.9) に示したように，2 つのベクトルの同じ項を掛け合わせる計算が必要であり，ここで $2n$ 回の乗算が行われる．よって，n 桁の乗算を行うために，乗数と被乗数に対する 2 回の DFT，1 回の IDFT，1 回の項ごとの乗算を合計して $3((2n)^2) + 2n$ に比例する計算量が必要となる．これは通常の筆算の計算量 $O(n^2)$ より大きい．ここで DFT と IDFT に FFT アルゴリズムを用いることでフーリエ変換の計算量を $O(n^2)$ から $O(n \log n)$ に削減することが

できるため，先に述べた乗算の計算量を $3((2n)^2)+2n$ から $3(2n\log 2n)+2n$ に比例する程度まで削減することができる．

一般に，DFT と IDFT は複素数に対する演算であるが，DFT を利用した乗算の最終的な演算結果は，誤差が十分小さければ実ベクトルとなる．なぜならば，実ベクトルの DFT 結果は常に複素共役対称な複素ベクトルとなり，複素共役対称な複素ベクトルの IDFT 結果は常に実ベクトルになるからである．

FFT 乗算アルゴリズムの基本は以上に述べた通りであるが，これを計算機上で行うためには，他にいくつかの手続きが必要になる．これらも含めた処理フローを図 3.1 にまとめる．まず，r 進数 n 桁の乗数と被乗数を，n 個の項を持つベク

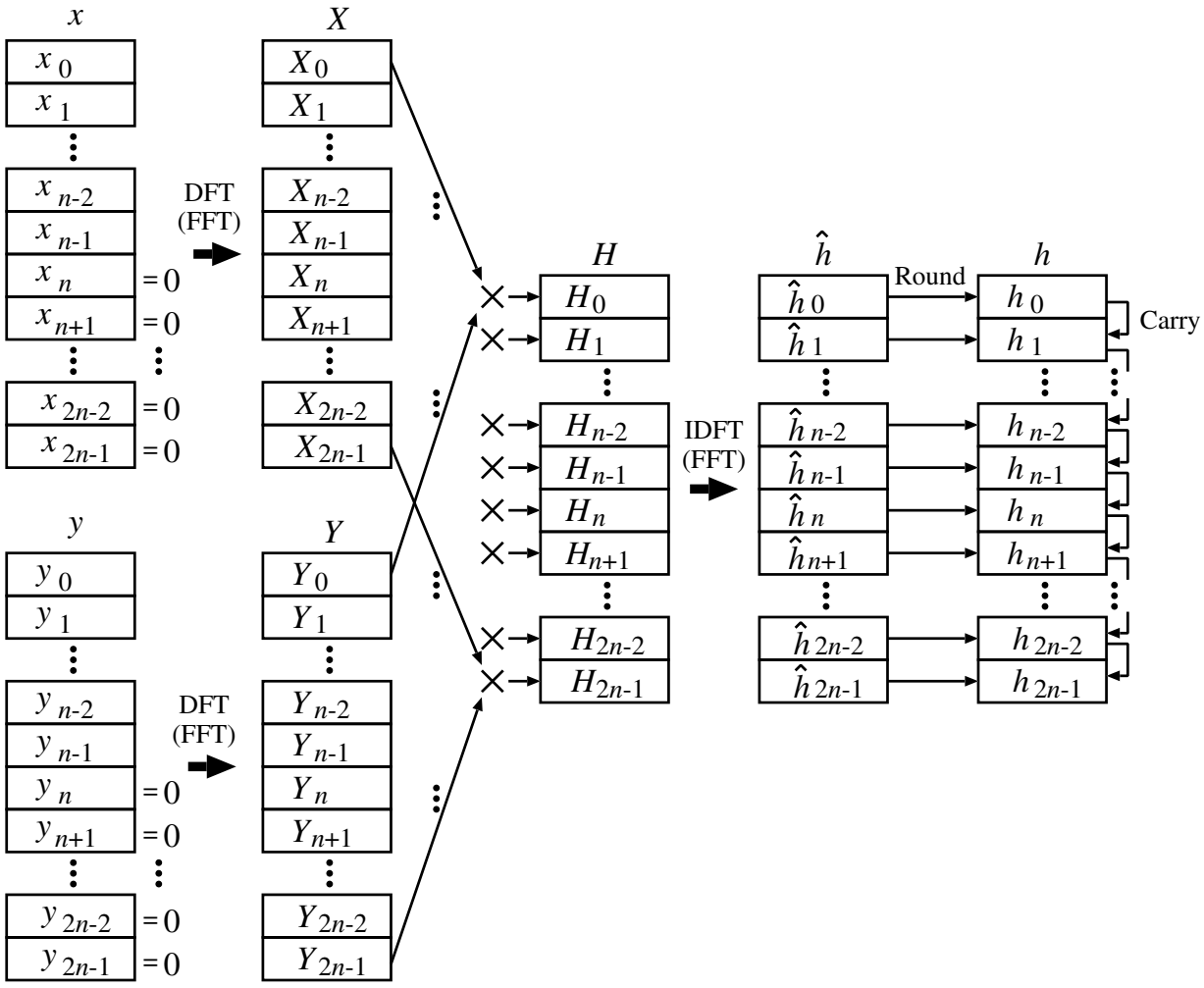

図 3.1 計算機における FFT 乗算の流れ

トルとみなす．それぞれの上位桁に n 個の 0 を繋げ，$2n$ 個の項を持つベクトル $x=(x_i)_{i=0,1,\ldots,2n-1}$，$y=(y_i)_{i=0,1,\ldots,2n-1}$ を作る．それぞれに対して DFT を行

い，変換結果 $X = (X_i)_{i=0,1,\ldots,2n-1}$，$Y = (Y_i)_{i=0,1,\ldots,2n-1}$ を得る．この X，Y を項ごとに乗算することによって，積の DFT である $H = (H_i)_{i=0,1,\ldots,2n-1}$ を得る．これを IDFT すると，積が求まる．この DFT と IDFT は FFT アルゴリズムで行う．ただし，計算機上での実数演算により，積は誤差を含んだ $\hat{h} = (\hat{h}_i)_{i=0,1,\ldots,2n-1}$ で得られる．誤差を取り除き，値を整数にするために各項の小数点以下 1 桁目を四捨五入 (2 進数値の小数点以下 1 桁目を 0 捨 1 入) することによって整数の積 h を得る (Rounding)．r 進数どうしの乗算を行った場合の第 i 項の値 h_i が，$h_i > r-1$ ならば，桁上げ (Carry) が発生したということなので，$h_i - (r-1)$ を h_{i+1} に加算する．これを h_0 から h_{2n-1} まで順に行うことによって，桁上げを伝搬させる．以上の手順で，FFT による乗算を行うことができる．

上述の FFT 乗算アルゴリズムの中で，項ごとの乗算を行う部分以外の計算量は，FFT の計算量によるため $O(n \log n)$ である．ただし，基数 r が非常に大きな値である場合，項ごとの積をとる際にも再帰的に FFT 乗算アルゴリズムを適用する必要がある．項ごとの積が定数時間と見なせるほど小規模になるまで FFT 乗算アルゴリズムを再帰的に適用すると，最終的な計算量は $O(n \log n \log\log n)$ となる．本研究においては，乗算桁数によらず基数 r を 16 に固定し，項ごとの乗算に FFT 乗算を再帰的に適用しない．したがって，計算資源によって乗算桁数が限定されるが，計算量は $O(n \log n)$ となる．

FFT 乗算には，FFT を上述の様に複素数で行う方法の他に，整数で行うもの [14] や，FFT に剰余理論を取り込んだ手法 (Fast Modulo Transformation, FMT) によるもの [28] が知られており，計算量はいずれも同じである．FMT による乗算法は他の方法と比較してメモリの使用量を削減できる．一方，複素数 FFT による乗算法は，信号処理などで用いられている FFT を利用することができる．本論文では，既存のハードウェア設計を有効活用できる点を考慮し，複素数 FFT による乗算法を採用する．

3.3　FFT アルゴリズム

3.3.1　Cooley Tukey アルゴリズム

FFT は 1965 年に J.W.Cooley と J.W.Tukey によって発表された算法 [29] で，以来，様々な用途に利用されている [30]．この Cooley Tukey による FFT のアルゴリズムを簡単に述べる．まず，式 (3.12) のような N 個の複素数を変換する DFT を

考える．

$$X_i = \sum_{k=0}^{N-1} x_k W_N^{ik} \tag{3.12}$$

$$W_N^{ik} = e^{\frac{-2\pi ikj}{N}} = \cos(\frac{2\pi ik}{N}) - j\sin(\frac{2\pi ik}{N}) \tag{3.13}$$

この DFT を単純に計算すると，N 次元ベクトルと $N \times N$ 行列の乗算になるので，$O(N^2)$ の計算時間が必要である．

ここで W_N^i を f とおき，多項式 $p(f)$ を用いて式 (3.12) を次のように表現する．

$$p(f) = x_0 + x_1 f + x_2 f^2 + \cdots + x_{N-1} f^{N-1} \tag{3.14}$$

すなわち， $X_i = p(W_N^i)$ とする．ここで $p(f)$ の項を偶数と奇数に分け，2 つの多項式 $P_e(f)$，$P_o(f)$ を作る．すると，式 (3.14) は次のように変形することができる．

$$\begin{aligned} p(f) &= x_0 + x_1 f + x_2 f^2 + \cdots + x_{N-1} f^{N-1} && (3.15)\\ &= x_0 + x_2 f^2 + \cdots + x_{N-2} f^{N-2} + f(x_1 + x_3 f^2 + \cdots + x_{N-1} f^{N-2}) && (3.16)\\ &= p_e(f^2) + f p_o(f^2) && (3.17) \end{aligned}$$

今，$f^2 = W_N^{2i} = W_{N/2}^i$ であるので，$p(W_N^i)$ を $0 \le i \le N-1$ について行う計算は，N 回の加算と，$W^0 = 1$ の場合を除いた $N-1$ 回の乗算によって，$p_e(W_{N/2}^i)$ と $p_o(W_{N/2}^i)$ を $0 \le i \le N/2-1$ について行う計算に置き換えることができる．

ここで，$\psi(N)$ を N 点を変換する DFT に必要な演算回数とする．すると，$\psi(N)$ を行うために 2 回の $\psi(N/2)$，N 回の加算，$N-1$ 回の乗算が必要であるため

$$\psi(N) = 2\psi(N/2) + 2N - 1 \tag{3.18}$$

となる．これを $N = 2^k$ とおいて $\psi(1) = 0$ が現れるまで再帰的に計算すると

$$\psi(2^k) = N(2\log_2 N - 1) + 1 \tag{3.19}$$

となる．以上の説明は文献 [31] に基づく．このアルゴリズムは，多項式を 2 個に分割することが繰り返されることから基数 2 の FFT と呼ばれ，Radix-2 FFT と表記される．Radix-2 FFT の長さは 2 の冪である必要がある．

DFT において，式 (3.12) の右辺を周波数領域，左辺を時間領域と呼ぶ，これはそもそもフーリエ変換が任意の時間的に連続な波形を周波数ごとに分割する目的で考案されたことに由来する．上記の FFT の説明では，多項式を添字が偶数の場合と奇数の場合で分割したが，多項式の上位 $N/2$ 項と下位 $N/2$ 項に分割する方法もある．先に述べた説明は偶数と奇数に分割した．このような分割方法は，時間間引き型

(Decimation in time，DIT) とよばれる．一方，上位と下位で分割する方法を，周波数間引き型 (Decimation in Frequency，DIF) と呼ぶ．FFT には順変換と逆変換があるが，順変換に一方を用いると，逆変換には残りの一方が用いられるため，順変換の後に逆変換を行う場合には，順変換にどちらを選んでも本質的に同じである．

3.3.2　その他の FFT アルゴリズム

FFT には様々な種類がある [32, 33]．比較的単純な高速化手法として，高基数による高速化手法がある．これは，元の大きな DFT を 2 分割ではなく，4 分割や 8 分割することで，Radix-4 FFT や Radix-8 FFT を構成する方法である．分割数を増やすことで，再帰の回数を減らすことができるため高速な FFT を実現することができるが，FFT への入力の長さが 4 の冪，8 の冪となり，構成が複雑となるため実用性の面ではトレードオフが存在する．実用性の限界は Radix-4 FFT であると言われている [30]．

より複雑な方法の例として，複数の基数を組み合わせる Split-Radix FFT，多次元の DFT を一次元の DFT の直積で計算する Row-Column 法を用いた多次元の FFT (RCFFT)，一次元の FFT を多次元に拡張した VFFT (Vector-Radix FFT)，分解を互いに素な長さで行い RCFFT を適用した PFFFT (Prime Factor FFT)，さらにこれを改良した Winograd FFT などがあげられる．

3.3.3　FFT に用いられるデータ表現

FFT を計算機システム上で行う場合，データ表現として，主に浮動小数点表現，固定小数点表現，ブロック小数点表現が用いられている．本節では，これらのデータ表現について述べる．

浮動小数点表現

浮動小数点表現は現在でも多くの汎用プロセッサで採用されている実数表現である．データ表現の構成を図 3.2 に示す．この表現方法は，符号 (Sign)，指数部

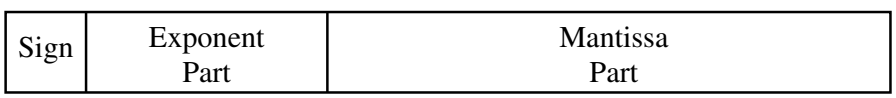

図 3.2　浮動小数点表現の構成

(Exponent Part)，仮数部 (Mantissa Part) で構成され，指数部の値を exp，仮数部

の値を $frac$，基数を r とすると，数値は $\pm frac \times r^{exp}$ で表現される．現在，汎用プロセッサなどで用いられている浮動小数点表現は，IEEE 754 で標準化されている．浮動小数点表現は，指数部で値の範囲，仮数部で精度を表す方式であり，指数部と仮数部の大きさを変える事で様々な演算に適用することができ，その応用範囲は広い．ただし，四則演算が複雑になる傾向にある．特に，加減算は 2 値の指数部の値を同じにしてから行うため，複雑な処理が必要となる．

固定小数点表現

固定小数点表現は，主に演算器の単純化を目的として用いられる実数表現である．構成を図 3.3 に示す．この表現は，小数点の位置を固定した単純な表現方法である．

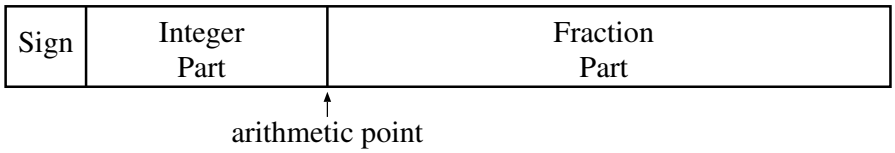

図 3.3 固定小数点表現の構成

大きな値を表現する場合，必要なビット長が長くなる傾向にあり，大小様々な範囲の値を扱う場合には無駄が多くなる．しかし，本質的に整数と同じように扱う事ができ，演算を単純な演算器で実現することができるため，小さな FFT ではこのデータ表現が用いられる場合がある．

ブロック浮動小数点表現

ブロック浮動小数点表現は，浮動小数点表現と固定小数点表現を組み合わせたものである．構成を図 3.4 に示す．この表現形式は，複数のデータで浮動小数点の指数部を共有したものである．したがって，個々の値が取り得る範囲の自由度は低くなるが，浮動小数点表現の性質をある程度維持しながらも，演算器を単純化することができる．

3.3.4 既存の FFT ハードウェア

ハードウェア FFT 乗算器を設計するにあたり，既存の FFT ハードウェアを用いることが可能である．そこでまず，既存の FFT ハードウェアの具体的な例を挙げる．

実装例の一つとして，FFT に用いられる実数演算を浮動小数点ではなく固定小数点表現や，指数部を複数のデータで共有するブロック小数点表現などで行い，演算

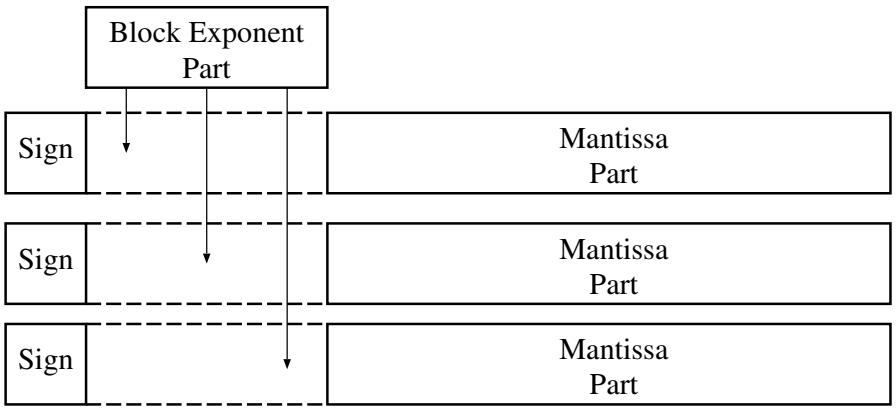

図 3.4　ブロック浮動小数点表現の構成

器の構造を単純化した実装がある．これにより，演算器の動作が高速化になり，実装コストも大幅に削減することができる．他にも，FFT ハードウェア中で使用されている乗算器を効率化したり，FFT の計算量を削減したりすることにより，高速かつ低コストな実装を行った例もある [34, 35]．また，ALTERA 社 [36] によって，Stratix デバイス用に最適化された FFT が IP 部品として提供されている．Stratix とは，ALTARA 社が開発した FPGA である．ALTERA 社が IP 部品として提供する FFT の仕様は文献 [37] で公開されている．これを表 3.1 にまとめる．この FFT

表 3.1　ALTERA 社 FFT IP の仕様

基数	2
データメモリ	Dual Port RAM(one read port one write port)
データ表現	IEEE754 32bit 浮動小数点

ハードウェアを用いて 1,024 個のデータを変換した場合の性能とコストを，表 3.2 に示す．表中の DSP ブロックとは，信号処理プロセッサで頻繁に使用される積和演算

表 3.2　1,024 個の変換におけるコストと性能

LE (Logic Element) 数	2,704
DSP ブロック数	2
メモリブロック	72Kbit
クロック数	10,630

を行うために用意された特別なハードウェア・マクロである．また，メモリブロックとは，同様に，メモリを構成するための特別なブロックである．共に Stratix 内部に特別な領域として確保されている．表のクロック数から，この回路を 50MHz で動作させると，変換にかかる時間は $(10,630/50) \times 10^{-6} = 0.2126$ms であることがわかる．

上に述べたような，既存の実装法や IP を用いて FFT 乗算器を実現することは可能である．しかし，上述の FFT 実装は，乗算に利用することを前提に設計されていないため，FFT 乗算に最適であるとは言えない．したがって，本研究では既存の FFT を用いずに，乗算用に最適化した FFT を独自に設計する．

3.4 FFT 乗算と誤差

FFT 乗算は，内部の演算を実数で行うため誤差が発生，拡大する．また，最終的な整数の積を得る際には実数を整数に丸める必要がある．整数への丸めは小数点以下 1 桁目を四捨五入することで行われる．よって，演算で発生した誤差が蓄積し，各桁の値を整数に丸める際に各桁における絶対誤差が絶対値で 0.5 を超えると，値が 1 つ隣の整数に丸められてしまう．したがって，FFT 乗算を行う際は，誤差の振舞に注意する必要がある．そこでまず，FFT 乗算における誤差の振る舞いを調べる．

3.4.1 FFT の誤差

FFT 乗算は，そのほとんどが FFT の演算で実現されているため，FFT における誤差解析が詳細に行われていれば，それを用いて FFT 乗算の誤差を解析することが可能であると考える．FFT における誤差解析の結果は過去に報告されている．これらを表 3.3 にまとめる．Weinstein ら [38] は，浮動小数点表現を用いた FFT のノイズ・シグナル比 (Noise to Signal Ratio，NSR) を統計的なモデルで定式化し，実験値との比較を行った．その結果，NSR は変換するデータ数の対数に対して線形であることを示した．Tran-Thong ら [39] は，DIT (時間間引き) 型と DIF (周波数間引き) 型 Radix-2 FFT と，DIT 型 Radix-3 FFT について，同様に NSR による誤差評価を行い，解析的な誤差見積もりと実際の誤差の比較を行った．その結果，DIT 型でも DIF 型でも誤差の振舞はほぼ同じであることを示した．また，DIT 型 Radix-3 FFT について，NSR は変換するデータ数の対数に対して単調増加であることを示した．Pitas ら [40] は，RCFFT，VFFT，PTFFT (Polynomial Transform FFT) に対して解析的な NSR を示した．Munson[41] らは，Prime Factor FFT の誤差を解析的に求め，シミュレーション結果との比較を行った．Ma[42] は，固定小

表 3.3　FFT の誤差に関する研究

文献	FFT の種類	データ表現	評価方法
Weinstein[38]	Radix-2 FFT	浮動小数点	NSR
Tran-Thong ら [39]	Radix-2 FFT Radix-3 FFT	浮動小数点	NSR
I. Pitas ら [40]	RCFFT VFFT PTFFT	浮動小数点	NSR
D. C. Munson ら [41]	Radix-2 FFT PFFFT	浮動小数点	NSR
Y. Ma[42]	Radix-2 FFT	固定小数点 ブロック浮動小数点	分散

数点表現とブロック小数点表現を用いて，Radix-2 FFT の誤差を解析的に示し，シミュレーション結果との比較を行った．

これらの研究結果は, FFT を DSP へ利用することを前提としたものである．よって，評価も NSR や分散といった統計的尺度に基づいている．一方，FFT を乗算に用いる場合は，統計的な誤差の振舞いよりも，その最大値が重要となる．したがって，上述の FFT の誤差に関する研究で得られた結果は，そのまま FFT 乗算の誤差解析に用いることはできない．これらの研究から FFT 乗算の誤差解析にも言えることは，変換するベクトルの長さが大きくなった時に誤差が拡大するという点と FFT のアルゴリズムによって誤差の振舞いが異なるという点である．FFT 乗算における誤差解析は FFT の誤差解析とは別の尺度で行う必要がある．

3.4.2　FFT 乗算における誤差の見積もり

FFT 乗算の誤差見積もりは，Henrici によって解析的に行われている [43]．これによると，マシンイプシロンが ϵ_M の計算機上で，桁数 n ，基数 r の FFT 乗算を行った場合，正しい積を得るには，式 (3.20) を満足する必要があるとされている．

$$\epsilon_M \leq \frac{1}{192n^2(2\log_2 n + 7)r^2} \tag{3.20}$$

この結果は，FFT 乗算における誤差伝搬の一次モデルに基づくものであり，誤差の最小の上界を表すものではない．

一方，平山により実験的な誤差解析も行われている [44]．平山は，Henrici が行った解析的な誤差の見積もりは最悪ケースであり，実際の計算では，誤差の条件はもっと緩いことを指摘し，数値実験でこれを確認している．IEEE754 の 64bit 表現を用いた場合を例にあげると，仮数部長は 52bit であるので，マシンイプシロンは $\epsilon_M \fallingdotseq 2.220446 \times 10^{-16}$ である．これを式 (3.20) で評価した場合，10 進数 77,091 桁まで計算可能であるという結果が得られるのに対し，実際には 100 万桁以上の演算が可能であることが示されている [44].

また，平山の実験では，IEEE754 や IBM 形式など，広く一般的に利用されている浮動小数点表現を用いて，基数 r と桁数 n を変化させながら FFT 乗算を行い，正しい積を得ることができる r と n の限界を測定している．この実験において，r 進数 n 桁で表現できるすべての値の組み合わせで正しい積が得られることを保証する必要があるが，演算する値は多倍長であるため，すべての組み合わせの誤差を測定することは現実的ではない．そこで，最大の誤差が発生する組み合わせで正しい結果を得ることで，すべての組合せで正しい積が得られることを保証している．この最大の誤差が発生する値の組み合わせに関して，平山は式 (3.20) を誤差の評価式とみなし，基数 r が大きいほど要求される精度が高くなるという点から，すべての桁が $r-1$ となるような値，すなわち，最大値どうしの乗算が最大の誤差を与えるとしている．

このことを確認するために実験を行った．実験は乗数 $u = (\overbrace{0\ldots0}^{n}\overbrace{a\ldots a}^{n})_{16}$，被乗数 $v = (\overbrace{0\ldots0}^{n}\overbrace{b\ldots b}^{n})_{16}$ の a，b を 0 から F まで変化させた値の組合せで FFT 乗算を行い誤差を測定するものである．測定は，各 i 桁目における真の値を R_i とし，計算による値を $R'_i + jI'_i$ とした場合，$\max(|R_i - R'_i|)$ $(i = 1, 2, \ldots, 2n)$ を調べるものである．この時，I'_i は本来 0 になるはずなので無視する．FFT には最も基本的な Cooley Tukey FFT[29]，データ表現には IEEE754 の倍精度 (double) を用いた．結果を図 3.5 に示す．グラフの x 軸は a，y 軸は b，z 軸は $u \times v$ における絶対誤差を示す．誤差の値は 10 進数値である．グラフは桁数 n が 1,024 桁と 2,048 桁の場合を示している．図から，$u = (0\ldots0\mathrm{F}\ldots\mathrm{F})_{16}$，$v = (0\ldots0\mathrm{F}\ldots\mathrm{F})_{16}$ の場合に最大の誤差が発生していることが確認できる．4,096 桁，8,192 桁についても測定を行った結果，同様の結果が得られた．また，前節で示した FFT の誤差に関する先行研究の結果を踏まえると，桁数が大きくなると，誤差は大きくなるが，振舞はアルゴリズムが同じであれば変わらないと考えることができる．以上より，FFT 乗算を行う場合は，最大値どうしの乗算で正しい結果が得られるような精度で演算を行えばよいことが確認できた．この結果は，数値実験により，Henrici による誤差伝搬の一次モデルよりも精密に誤差を見積もったものであり，最小の上界により近いと言える．

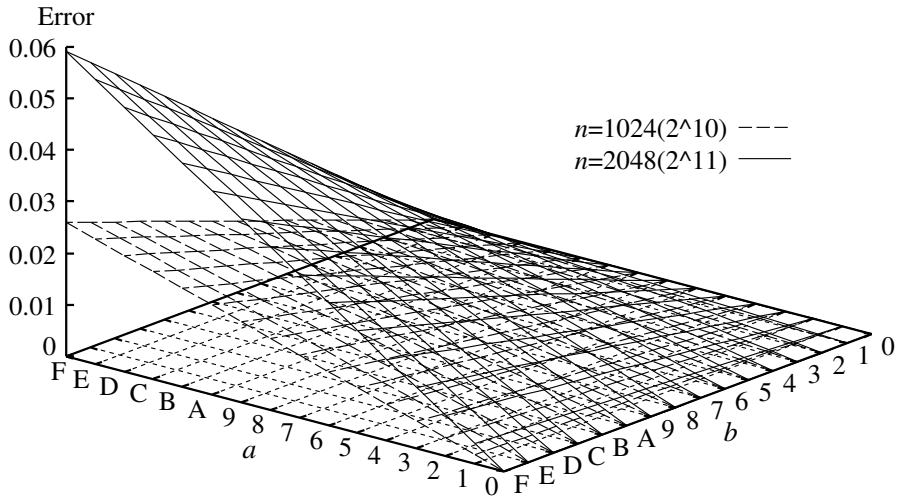

図 3.5　FFT 乗算において，$(0^n a^n)_{16}$ と $(0^n b^n)_{16}$ の乗算で発生する誤差

3.4.3　最小のデータ長

FFT 乗算では，ある程度大きな範囲の実数値を扱う．そのため，この実数を表現するために，浮動小数点表現を用いる必要がある．しかし，浮動小数点演算器をハードウェア実装した場合，一般に回路規模は大きくなりコストの増大と性能の低下につながる．そのため，浮動小数点演算器の規模を削減するような工夫が必要となる．

回路規模の削減法として効果的なのは，扱うデータのビット長を削減することである．そこで，浮動小数点表現の指数部，仮数部の長さを削減することを考える．指数部に関しては，演算中にオーバフローを起こさない最小の長さまで削減できる．一方，前述のように，FFT 乗算は演算の精度によっては正しい演算が行われないため，仮数部の最小ビット長は誤差解析によって必要最低限の精度を求め，それに基づいて決定する必要がある．演算に要求される精度は乗算桁数に依存する．言い換えると，乗算桁数が決まれば正しい演算を保証する最低限の精度が定まり，必要な仮数部長を見積もることができる．そこで，第 3.4.2 節の結果を利用し，乗算桁数に対して要求される最小仮数部長を見積もる．

まず，Henrici による解析的な誤差見積もりの式 (3.20) に基づいて最小の仮数部長を考える．データ表現の精度を表すマシンイプシロン ϵ_M は，仮数部長を f とすると，$\epsilon_M = 1/2^f$ と表すことができる．これを，式 (3.20) に適用すると，仮数部長 f，桁数 n，基数 r の関係が次式のように得られ，これから，n と r が与えられた場

合の仮数部長を求めることができる．

$$\frac{1}{2^f} = \frac{1}{192n^2(2\log_2 n + 7)r^2} \tag{3.21}$$

しかしながら，式 (3.21) で求められる仮数部長は最悪ケースの誤差計算に基づくものである．そこで，前節の誤差解析に基づいて，より最適な仮数部長を実験的に求めた．

実験は，ソフトウェアで仮数部と指数部のビット長を自由に変更できる浮動小数点表現を実装し，乗算桁数 n を変化させながら，n 桁で表現できる最大値どうしの乗算 (FF...FF×FF...FF) を行い，正しい結果が得られる最小の仮数部長を測定するものである．またこのとき，浮動小数点表現の仕様を以下のように定めた．

- 乗数と被乗数の基数 r は 16 とする．
- 仮数部には MSB の 1 を省略する正規化数を用いる．
- 非正規化数は扱わないものとする．

これらの仕様は後に述べる FFT 乗算器のハードウェア実装でも用いる．結果を図 3.6 に示す．グラフの横軸は桁数の対数，縦軸はビット長を表している．× 点は乗算

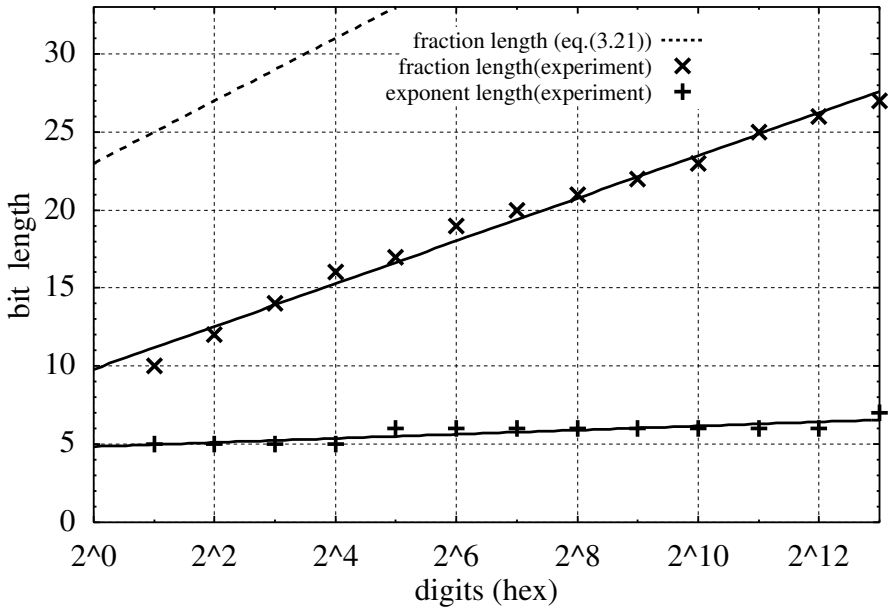

図 3.6 指数部と仮数部のビット長と乗算桁数の関係 ($r = 16$)

桁数に対する最小の仮数部長を実験により測定した結果を示し，+ 点は演算中にオーバフローしない最小指数部長を表している．また，点線は式 (3.21) を示している．

実験によって得られた仮数部長は，式 (3.21) から求めたものと比較してかなり短

いことが確認できる．また，16 進数 2^{13} 桁 (10 進数 9,831 桁) どうしの乗算を行う場合を例にすると，演算に必要なビット長は，指数部が 7 ビット，仮数部が 27 ビットである．したがって，例えば，この桁数の乗算を IEEE754 の 32 ビット表現 (指数部 8 ビット，仮数部 23 ビット) を用いて行うと，精度が足りず間違った結果を出力してしまう．また，64 ビット表現 (指数部 11 ビット，仮数部 52 ビット) を用いて行うと精度が余り，演算器の性能低下や面積増大をまねく．ここで示したように，演算に必要なビット長を測定した上で回路を設計することで，精度不足や回路規模増大などの問題を回避し，性能や面積を最適化した演算器を設計することができる．

3.5　FFT 乗算器の設計

3.5.1　FFT の設計

本実装で使用する FFT は，

1. ハードウェアによる実装が容易
2. アルゴリズムや設計した回路の解析が容易

という理由から，本章の冒頭で紹介した最も初期的な FFT である基数 2 の Cooley Tukey 型 [29] を用いる．以降，本論文で単に FFT と書いた場合は，この FFT を意味する．また，本設計では，順変換に DIT (時間間引き型) を用いる．

図 3.7 に 長さ 8 の DIT 型 FFT のデータフローを示す．このフロー図の導出は付録 C で述べる．演算は図の左から右へ行われる．図中の○で囲まれた数字を中心とする交差した流れは 1 回のバタフライ演算を表す．バタフライ演算は，FFT の基本演算であり，入力を p, q，出力を P, Q とすると，その演算内容は式 (3.22), (3.23) で表される．

$$P = p + qW^k \tag{3.22}$$

$$Q = p - qW^k \tag{3.23}$$

$$W^k = \cos(\frac{-2\pi k}{N}) + j\sin(\frac{-2\pi k}{N}) \tag{3.24}$$

回転子 W^k は，sin と cos の値として定義される．FFT のハードウェア実装において，この sin と cos の値を必要なタイミングでその都度計算すると，専用の大規模な演算器が必要になる．よって，一般的な FFT ハードウェアでは，これらの値はあらかじめ計算し，テーブルとしてメモリに格納しておく方法が用いられている．本実装においても同様の方法を用いる．

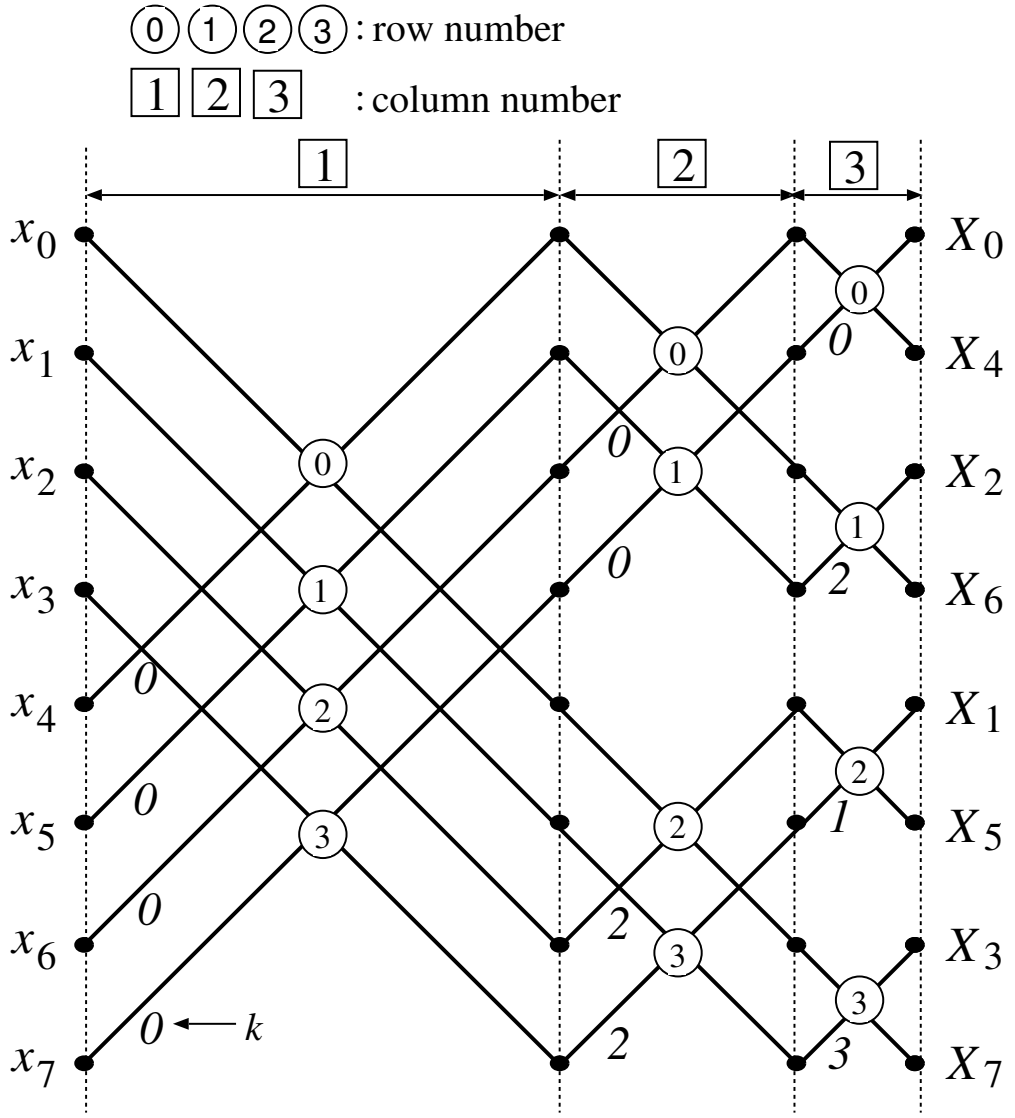

図 3.7 基数 2 の時間間引き型 FFT のデータフロー

FFT の逆変換は図 3.7 を，順変換とは逆に，右から左にたどったものとなる．したがって，逆変換におけるバタフライ演算は，式 (3.25)，(3.26) のようになる．

$$p = P + Q \tag{3.25}$$

$$q = (P - Q)W'^{k} \tag{3.26}$$

$$W'^{k} = \cos(\frac{-2\pi k}{N}) - j\sin(\frac{-2\pi k}{N}) \tag{3.27}$$

逆変換における回転子 W'^{k} は W^{k} の複素共役である．

3.5.2　データ表現

本実装では，IEEE754 の浮動小数点表現を基本としたデータ表現形式を採用した．このデータ表現形式は，非数と非正規化数に対応していないという点で，IEEE754 の仕様とは若干異なる．仕様を次にまとめる．

- 値 v は符号 $sign$，指数部 exp，仮数部 $frac$ からなる．
- 指数部と仮数部の基数は 2 とする．値 v は，$\pm frac \times 2^{exp}$ の形で表現される．
- 符号 $sign$ は 1 ビットで表現され，0 の時 $+$，1 の時 $-$ を表す．
- 指数部 exp はゲタばき表現で表され，n ビット長であるときのゲタは $2^n - 1$ とする．
- 仮数部 $frac$ は正規化とケチ表現を用いることで，$2 < frac \leq 1$ の範囲を表現する．
- 0 は，符号ビット以外がすべて 0 であるパターンで表現する．
- 非数 (Not a Number，NaN) と非正規化数 $(1 < |v| < 0)$ はサポートしない．

この表現で IEEE754 と同じ符号ビット，指数部 11bit，仮数部 52bit という構成をとると，$-1.999\cdots \times 2^{-1023} < v < 1.999\cdots \times 2^{1024}$ の値を表現する事ができる．またこのとき，マシン・イプシロン ϵ_M は 2^{-52} となる．このような精度を持つデータ表現形式を用いて FFT 乗算を行った場合，IEEE754 の 64 ビット浮動小数点表現を用いた場合と同様に，約 100 万桁の演算が可能である．

3.5.3　メモリアーキテクチャ

FFT 乗算器の各演算器をパイプライン化すると，あるタイミングにおいて，同じメモリに対する書き込みと読み出しが同時に起こり得る．このハザードを解決するために，Memory モジュールの設計において，次の 2 つの方法を検討した．1 つ目は，通常の Single Port RAM (SPRAM) を用いて読み出しと書き込みをそれぞれ異なるクロックで行う方法である．もう 1 つは，読み出しと書き込みを同一のクロックで行うことができるように，Read 用の Port と Write 用の Port を別々に設けた Dual Port RAM (DPRAM) を用いる方法である．メモリの構造としては後者の方が理想であるが，Port あたりのコストは大きい．そこで，これらの方法を選択するにあたり，この SPRAM と DPRAM のコストと性能のトレードオフを調べるための実験を行った．実験は，1 語 64 ビットとして，8 語を格納できる SPRAM と DPRAM を Verilog-HDL で記述し，論理合成ツール Design Compiler で遅延時間

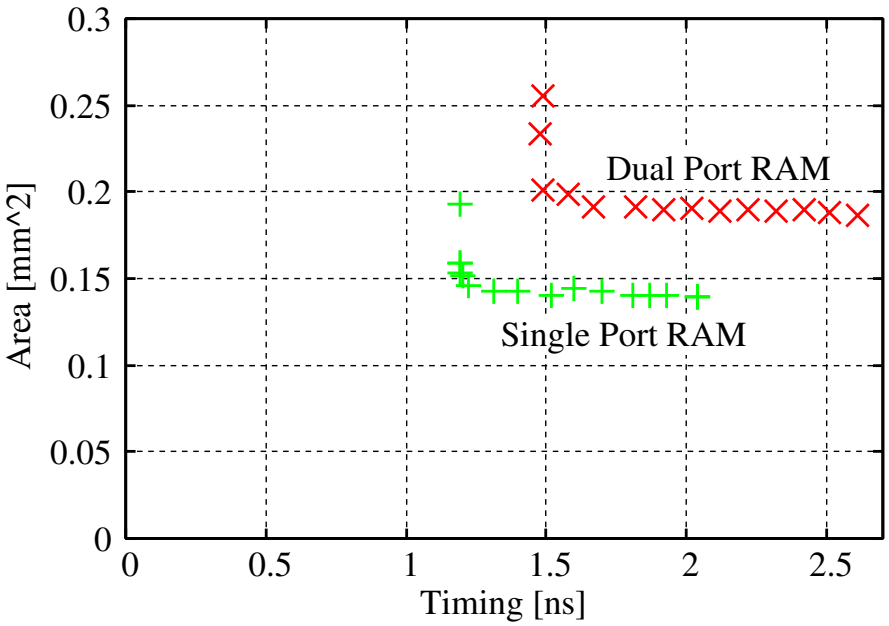

図 3.8 RAM と DPRAM の面積と性能

の制約を変えて合成した結果から，コストと性能のトレードオフを調べるというものである．実験環境を表 3.4 に示す．比較の結果を図 3.8 に示す．グラフは，x 軸が

表 3.4 SPRAM と DPRAM の比較実験環境

記述言語	Verilog-HDL
Simulator	Verilog-XL 04.10.001-p
論理合成	Design Compiler 2003.3
Technology	日立製作所 CMOS 0.18μm

遅延時間，y 軸が面積を表している．グラフより，SPRAM の面積が約 0.14mm^2，DPRAM の面積が約 0.19mm^2 で安定している様子がわかる．これから，DPRAM の面積は，SPRAM の約 1.4 倍大きいことがわかる．また，より原点に近い点の遅延時間は，DPRAM が 1.5 ns，SPRAM が 1.2ns となっている．したがって，DPRAM の遅延時間は，SPRAM の約 1.2 倍であることが分かる．これらの結果を踏まえると，効率の良い DPRAM を使用した場合であっても遅延時間とコストの増加は許容範囲であると言える．この考察から，本実装では DPRAM を採用する．

3.5.4　FFT 乗算器の構成

FFT 乗算を行うために必要となるモジュール構成を図 3.9 に示す．図中の実線矢

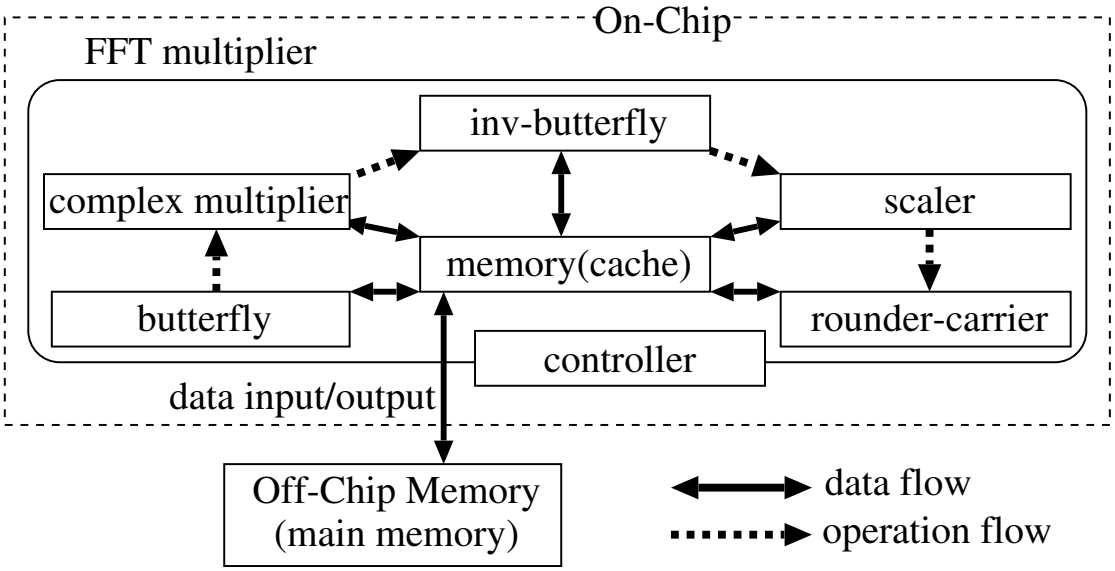

図 3.9　FFT 乗算器の構成

印はデータの流れを示している．演算は，FFT 乗算アルゴリズムにしたがい，図中の破線矢印が示す順番で行われる．全体として，各モジュールが 1 個のメモリを共有し，メモリからデータを読み出し，演算をして再びメモリに書き戻すという動作の繰り返しで演算が行われる．

演算器として，項ごとの乗算に使用する複素数乗算器 (complex multiplier)，順・逆 FFT を行うための順変換用バタフライ演算器 (butterfly) と逆変換用バタフライ演算器 (inv-butterfly)，浮動小数点表現の指数部の減算を行い，小数点を左シフトすることによって 2 のべき乗の除算を実現するシフタ (scaler)，実数から整数への丸めと桁上げの処理を行う丸め桁上げ器 (rounder-carrier) が用意されている．また，メモリ (memory) はチップ内部に用意されたキャッシュの役割を担うものであり，演算中の中間値や回転子を記憶する．回転子はあらかじめ計算し格納しておく．コントローラ (controller) はメモリアドレスの計算やデータの流れのコントロールを行う．各モジュールの詳細な説明は後で述べる．

浮動小数点加減算器 (fpaddsub)

fpaddsub モジュールは，浮動小数点加減算を行うモジュールである．浮動小数点の加減算は次の手順で行われる．

1. 入力値を符号，指数部，仮数部に分割する．
2. 加算を行うか，減算を行うかを決定する．
3. 指数部を比較し，小さい方の値の仮数部をシフトして桁を合わせる．
4. 加減算を行う．
5. 演算結果の正規化を行い，符号，指数部，仮数部を決定する．

浮動小数点乗算器 (fpmul)

fpmul モジュールは，浮動小数点乗算を次の手順で行うモジュールである．

1. 入力値を符号，指数部，仮数部に分割する．
2. 符号を決定する．
3. 仮数部の乗算を行う．
4. 仮数部の積をシフトし，正規化する．
5. 仮数部を正規化する際のシフト数を考慮して指数部を計算する．

複素数乗算器 (complex multiplier)

complex multiplier モジュールは，複素数 $s_r + s_i j$，$t_r + t_i j$ が与えられた時，式 (3.28) にしたがって複素数乗算を行う．ただし，j は虚数単位とする．一般に乗算器より加減算器の方がハードウェアの実装コストが低いため，式 (3.28) に示す式変形を行い加減算の回数を増やすかわりに乗算の数を削減している．

$$\begin{aligned} p_r + jp_i = (s_r + js_i)(t_r + jt_i) &= s_r t_r - s_i t_i + j(s_r t_i + s_i t_r) \\ &= s_r t_r - s_i t_i + j((s_r + s_i)(t_r + t_i) - (s_r t_r + s_i t_i)) \end{aligned} \tag{3.28}$$

式から，complex multiplier モジュールには fpaddsub モジュールが 5 個と fpmul モジュールが 3 個必要であることがわかる．モジュールのハードウェア構成を図 3.10 に示す．

バタフライ演算器 (butterfly， inv-butterfly)

butterfly モジュールは FFT の順変換に，inv-butterfly モジュールは逆変換に用いられるバタフライ演算器である．バタフライ演算の内容は FFT の種類によって

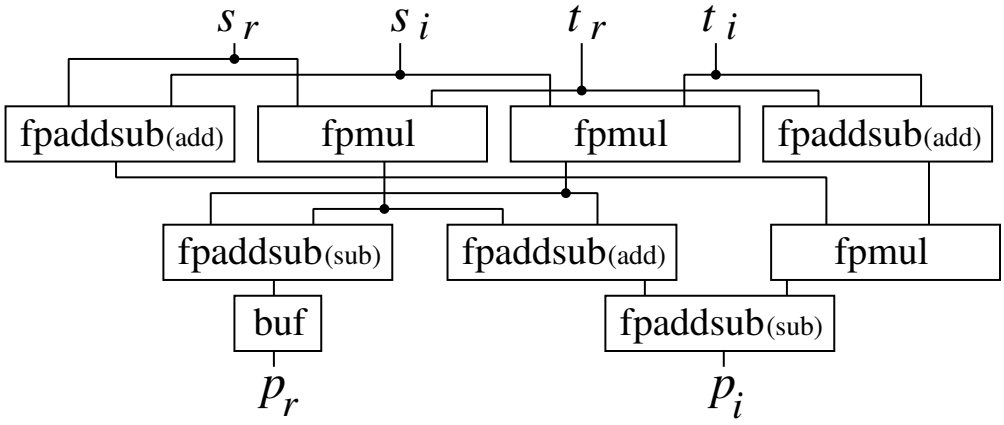

図 3.10　complex multiplier モジュールの構成

変わる．前述のとおり，本実装では Cooley-Tukey FFT を用いる．このときのバタフライ演算は，変換前の値を x，y，変換後の値を X，Y とすると，$X = x + yW$，$Y = x - yW$ で表される．ただし，x，y，X，Y，W は複素数である．W は FFT における回転子を表す．逆変換のバタフライ演算は，変換前の値を X'，Y'，変換後の値を x'，y' とすると，$x' = X' + Y'$，$y' = (X' - Y')W'$ で表される．x'，y'，X'，Y' も複素数である．W' は FFT における回転子であり W と複素共役である．butterfly と inv-butterfly には complex multiplier モジュール 1 個と fpaddsub モジュール 4 個がそれぞれ必要である．それぞれの演算器のハードウェア構成を図 3.11 と図 3.12 に示す．

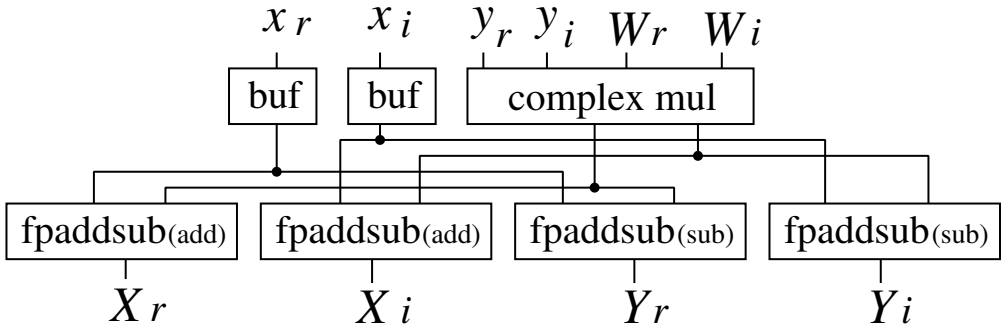

図 3.11　butterfly モジュールの構成

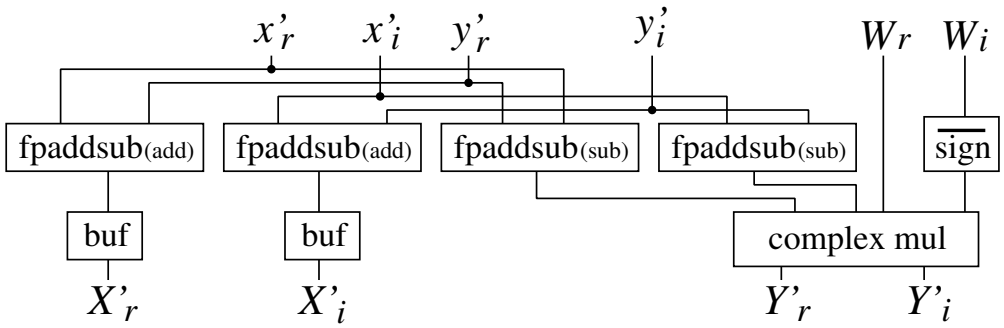

図 3.12 inv-butterfly モジュールの構成

スケーラ (scaler)

DFT は次式で定義される．

$$X_i = \sum_{k=0}^{N-1} x_k W_N^{ik} \tag{3.29}$$

この時，ベクトル X を x に変換する IDFT は

$$x_i = \frac{1}{N} \sum_{k=0}^{N-1} x_k W_N^{-ik} \tag{3.30}$$

で定義される．よって，逆変換値は N による除算が必要である．しかし，$N = 2^k$ (k は正整数) なので，実際に除算を行う必要はなく，浮動小数点表現の指数部から k を減ずればよい．scaler はこの操作を行うモジュールである．

また，ここで扱う値は実数ベクトルをフーリエ変換し，それを再び逆変換したものである．よって，DFT の対称性から，必ず実数ベクトルである．しかし，DFT の演算は複素数 $R'_i + jI'_i$ で行われるため，第 3.4.2 節でも述べたように，計算機の実数演算で発生した誤差によって虚数部 I'_i に 0 でない値が現れる場合がある．この時の I'_i は誤差そのものであるので，以降は虚数部 I'_i をすべて無視して R'_i だけの実数ベクトルとして扱ってよい．

よってこの scaler モジュールは R'_i の指数部を減ずるための単純な減算器 1 個で実現することができる．

丸め桁上げ器 (rounder-carrier)

rounder-carrier モジュールは，スケーリング後，実数ベクトルで表現された積の各桁を丸めにより整数に変換した後，桁上げを行う演算器である．浮動小数点表現の

指数部と仮数部の基数は 2 なので，小数点以下 1 桁目を 0 捨 1 入することで整数に丸めることができる．これにより，絶対誤差が 10 進数で ±0.5 を超えない限りは正しい結果を得ることができる．その後，r 進数の桁上げを行う．この手順を次にまとめる．

1. 入力値を受け取る．
2. 下位桁からの桁上げと入力値を加算する．(入力が最下位桁である場合は桁上げとして 0 を加算する．)
3. 加算された値の小数点以下 1 桁目を 0 捨 1 入し，整数に丸める．
4. 得られた整数から上位へ桁上げする値を計算し，保存する．もし，整数が桁に収まる値ならば，桁上げは 0 とする．
5. 桁の値を計算する．
6. 2 に戻り，最上位桁まで繰り返す．

この演算には，各桁の値を丸めて各桁の整数値と桁上がりを計算するためのマルチプレクサと加算器が 1 個ずつ，下位桁からの桁上げを加算するための浮動小数点加算器が 1 個必要である．rounder-carrier モジュールの構成を図 3.13 に示す．

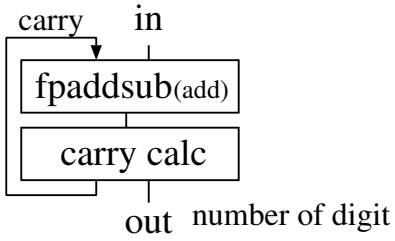

図 3.13　rounder-carrier モジュールの構成

メモリ (memory)

多倍長演算を行う場合，演算の対象となるデータを格納する記憶領域は巨大なものとなる．したがって，データ全体はチップ外部の大規模な記憶装置に格納し，演算に必要なデータのみをチップ内の記憶装置にキャッシュする必要がある．memory モジュールはこのキャッシュとして機能する．また，FFT を行うためには回転子 W^k を計算する必要がある．これをハードウェアで行うと多くのコストが必要となるため，W^k はあらかじめ計算し，テーブルとしてメモリに書き込んでおく必要がある．このテーブルも，全体はチップの外に持ち，必要な部分だけを memory モジュールに読み込む．

memory モジュールの構成を図 3.14 に示す．乗数 (Multiplier)，被乗数 (Multi-

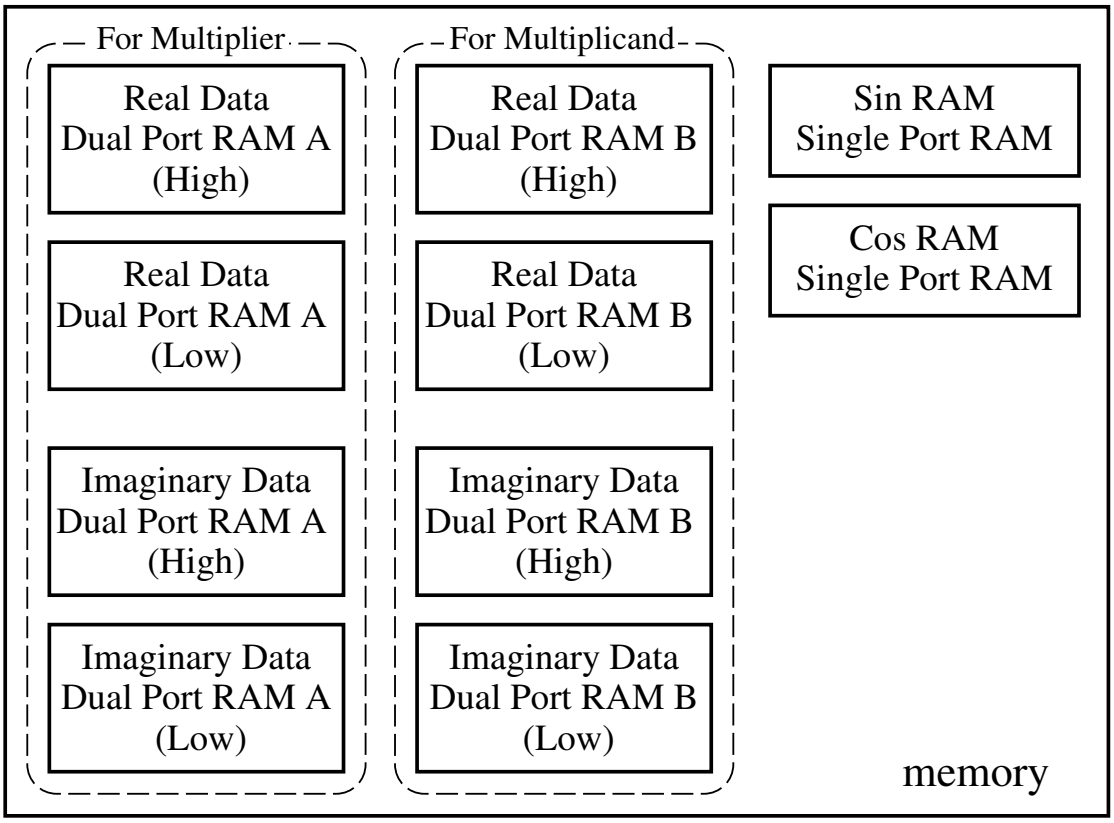

図 3.14　memory モジュールの構成

plicand) の実数部 (Real Data) と虚数部 (Imaginary Data) 用にそれぞれ専用の DPRAM を用意する．また，バタフライ演算の際には，2 個の値 x と y を同じタイミングでメモリから読み出し，演算結果の X, Y を同じタイミングで書き戻す必要がある．そのため，x の読み出しや X の書き込みを行うメモリを High メモリ，y の読み出しや Y の書き込みを行うメモリを Low メモリとして別々に用意することでメモリアクセスに関する構造的なハザードを回避し，効率の良いメモリアクセスを実現している．また，memory モジュール内には，回転子を sin と cos の定数値として記憶しておく RAM を用意する．演算中，この RAM に対しては読み出ししか行わないため，SPRAM を用いる．

コントローラ (controller)

controller モジュールは，各モジュールや memory モジュールへ適切なタイミングで制御信号を送信することで FFT 乗算器を制御する．また，memory モジュールに対する書き込みや読み出しアドレスの計算も行う．

ここでは順・逆 FFT におけるアドレス計算方法について述べる．順方向の FFT を行うためには，図 3.15 の流れで演算を行う必要がある．これは図 3.7 で示した FFT のデータフローに，読み書きするメモリの種類とそのアドレスを図示したものである．斜線部領域における中間値は High メモリに対して読み書きされ，点々部領

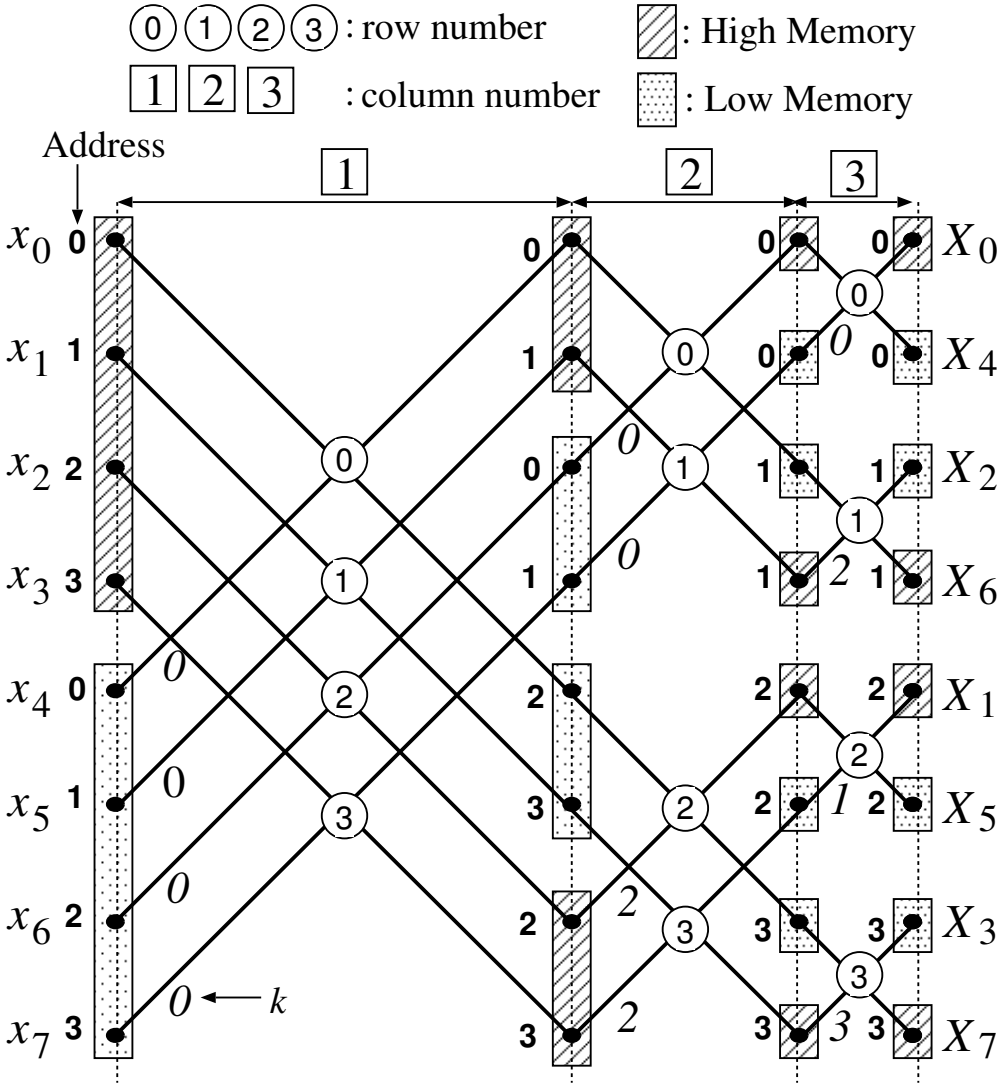

図 3.15　FFT におけるメモリアドレス指定

域における中間値は Low メモリに対して読み書きされる．また，斜線部領域と点々部領域の左側にある数字はメモリのアドレスを表している．

さらに，各段の上に書かれている□で囲まれた数字とバタフライ演算上に○で囲まれた数字は，それぞれ段数 $column$ と各段におけるバタフライ演算の番号 row を表している．以降，$column$ 段の row 番目のバタフライ演算を，$(row, column)$ 番目のバタフライ演算と表現する．

1 回のバタフライ演算に必要な値は式 (3.22) で示したように p, q, P, Q, W^k の 5 つである．これをフロー図にすると，図 3.16 のようになる．よって，1 回のバタフ

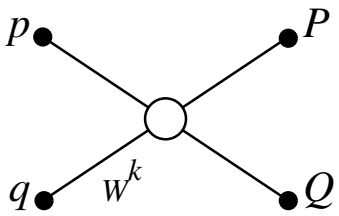

図 3.16 バタフライ演算のフロー図

ライ演算を行うためには，p, q, W^k を読み出すメモリと，P, Q を書き込むメモリそれぞれの種類とアドレスを求める必要がある．

次に，N 個の入力に対する FFT において，$(row, column)$ 番目のバタフライ演算に必要な値を保持しているメモリと，そのアドレスを決定する方法について説明する．p, q の値を読み出すメモリは，p を High メモリから，q を Low メモリから読み出すという状態を初期状態とすると，各段において，図 3.17 が示すように，バタフライ演算を $N/2^{column}$ 回行うごとに，High と Low を入れ換えるという操作で求めることができる．P, Q を書き込むメモリは同様に各段において，バタフライ演算を $N/2^{column+1}$ 回行うごとに High，と Low を入れ換える．読み出しアドレスは，

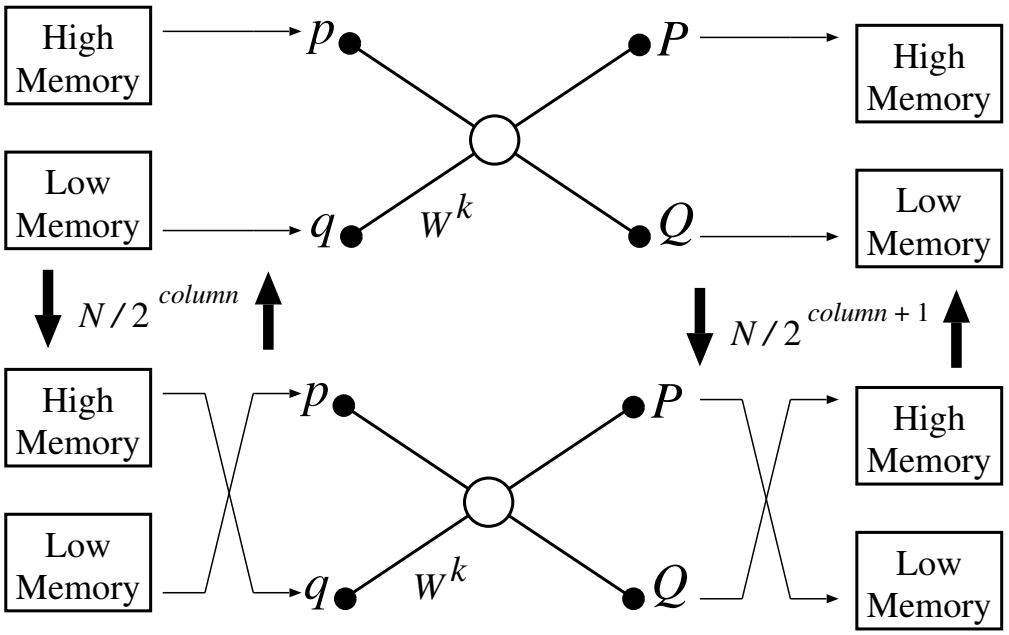

図 3.17 p, q, P, Q における読み書きメモリの決定方法

row そのものである．書き込みアドレスは，High メモリについては row そのものであるが，Low メモリについては，P を書き込む場合は $row - N/2^{column+1}$，Q を書き込む場合は $row + N/2^{column+1}$ となる．W^k に関しては，k の値がそのまま Sin RAM と Cos RAM のアドレスとなっている．$(row, column)$ 番目のバタフライ演算における k の求め方は次のとおりである．まず，row を ν $(= \log_2 N)$ ビットに 2 進展開したものに対して図 3.18 のようにビット入れ換えを行い，row_{rev} とする．k は row_{rev} を $(\nu - column)$ ビット左シフトすることで得られる．

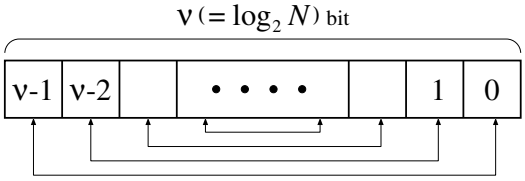

図 3.18　ビット入れ換え

FFT の逆変換は，図 3.15 に示した順変換の演算フローを逆からたどったものとなるので，メモリの種類やアドレスの決定方法は上述の説明の p，q を書き込み，P，Q を読み出しと読みかえたものになる．

Complex multiplier で項ごとの積を行うステージと Scaler でスケーリングを行うステージにおけるメモリ制御は単純で，High メモリの最下位アドレスから Low メモリの最上位アドレスまでを順番に演算していけばよい．Rounder-Carrier で桁上げと丸めを行うステージについても同様に，High メモリの最下位アドレスから Low メモリの最上位アドレスまでを順番に演算していけばよいが，下位桁からの桁上げが決定されるまで，演算が開始できないため，メモリへの書き込みや読み出しもその分だけ待つ必要がある点に注意が必要である．

3.6　実装と評価結果

3.6.1　実装条件

第 3.4.3 節の結果に基づき，桁数に対する最小のデータ長で FFT 乗算器を実装し，その面積と性能を調べた．基数 r は 16 とする．実装は，Verilog-HDL で記述した回路を Synopsys 社の Design Compiler 2003.6-SP1 で論理合成するという方法で行った．また，評価は論理合成結果から得られた回路面積と，回路の性能を決定する遅延時間を用いて行った．論理合成の際に用いたセル・ライブラリは日立製作所の仕様を基に VDEC[26] が作成した CMOS 0.18μm テクノロジのものである．また，

合成時の条件には性能を優先するような制約を与えた．

面積に関する評価には controller 以外のモジュールを個別に合成した結果の合計値を用いた．また，速度に関する評価にはこれらのモジュールの最大遅延時間を用いた．controller を評価から除いた理由は，面積と性能が，乗算桁数に対してほぼ一定であり，その他のモジュールと比較して面積や遅延時間が小さいためである．memory モジュールの容量は 8 エントリを保持できる大きさにした．1 エントリは，1 回の演算で最も多くのデータを必要とする butterfly モジュールでの演算 1 回分のデータを保持するのに必要な大きさとした．

3.6.2 乗算桁数に対する面積と最大遅延時間

まず，FFT 乗算器の面積と乗算桁数の関係について調べた．図 3.6 の結果をもとに，与えられた乗算桁数に対して最小のデータ長で FFT 乗算器を構成し，その面積を測定した．結果を図 3.19 に示す．グラフの横軸は桁数の対数，縦軸は面積を表し

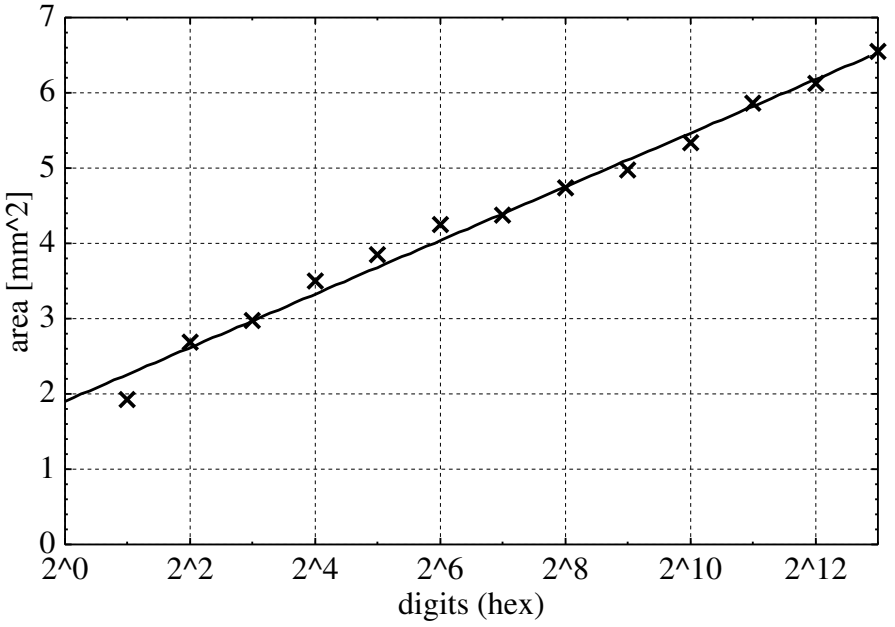

図 3.19 FFT 乗算器の面積と乗算桁数の関係 ($r = 16$)

ている．2^{13} 桁どうしの乗算に注目すると，図 3.6 より，最小の指数部長は 7 ビット，仮数部長は 27 ビットである．これに符号ビットを加えた 35 ビットが最小のデータ長である．この表現を用いた場合，図 3.19 から，面積は約 6.55mm^2 になることがわかる．IEEE754 の 64 ビット表現と同じく，符号ビット，指数部 11 ビット，仮数部 52 ビットからなるデータ表現を用いて本実験と同じ条件で面積を求めた

結果，16.0mm^2 であった．この結果から，最適なデータ長を用いて 2^{13} 桁どうしの乗算を行う FFT 乗算器を構成した場合，64 ビットの表現と比較して面積を約 60% 削減できることがわかる．

同様に，乗算桁数と最大値遅延時間についても調べた．結果を図 3.20 に示す．グ

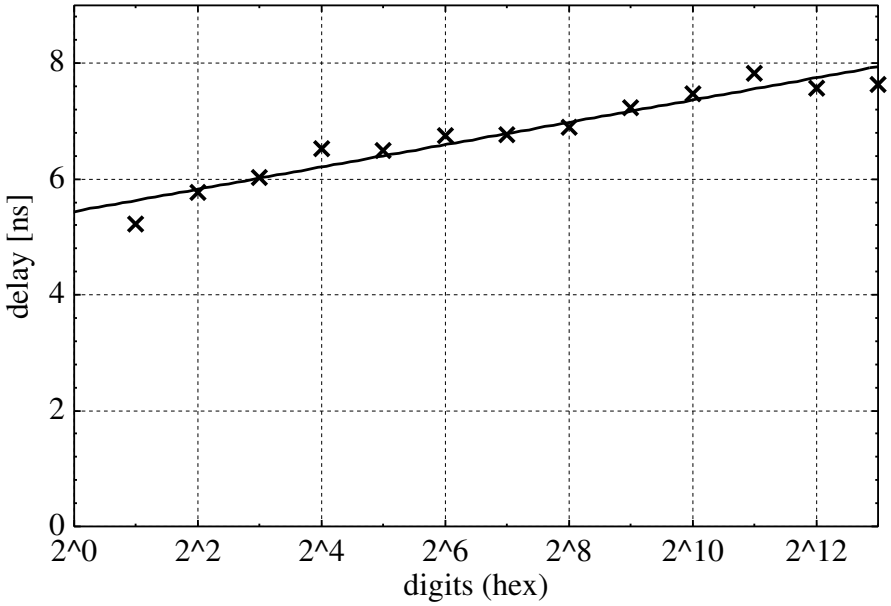

図 3.20　FFT 乗算器の最大遅延時間と乗算桁数の関係 ($r = 16$)

ラフの横軸は桁数の対数であり，縦軸は最大遅延時間を表している．グラフから，2^{13} 桁どうしの乗算を行う FFT 乗算器の最大遅延時間は 7.63ns であることがわかる．また，64 ビット表現を用いた場合の最大遅延時間を測定したところ，10.3ns であった．このことから，最大遅延時間は約 26% 削減できることがわかる．

3.6.3　パイプライン化による演算速度の向上

前節で評価した FFT 乗算器の浮動小数点加減算器や乗算器などのモジュールは，モジュール間にレジスタが挿入されているものの，モジュール内部はパイプライン化されていない．ここでは，モジュール内部のパイプライン化による速度向上を行う．

パイプライン化にあたり，回路のクリティカルパスを調べたところ，仮数部を乗算するために浮動小数点乗算器内で使用されている Wallace Tree 乗算器にあった．適切なパイプライン化を行うためには，面積と性能のトレードオフを調べる必要がある．そこで，2^{13} 桁の場合において Wallace Tree 乗算器のパイプライン化を行い，浮動小数点乗算器全体の面積と最大遅延時間の変化を調べた．結果を表 3.5 に示す．

表から，パイプライン段数が 4 段以上になると，遅延時間はあまり変化しないこ

表 3.5 浮動小数点乗算器内部で用いられている Wallace Tree 乗算器のパイプライン段数による最大遅延時間と面積の変化

pipeline depth	area[mm^2]	delay[ns]
1	0.25	4.56
2	0.29	2.63
3	0.30	2.06
4	0.31	1.60
5	0.35	1.55
6	0.39	1.54

とがわかる．これは，Wallace Tree 乗算器に存在したクリティカルパスが，パイプライン化によって短くなり，浮動小数点乗算器内にある他のパスの長さと同じかそれ以下になったことを表している．このことから，浮動小数点乗算器は，4 段パイプラインの Wallace Tree 乗算器を用いることで，最も面積と遅延時間のバランスが良くなることがわかった．

その他のモジュールも，このときの浮動小数点乗算器の遅延時間にあわせてパイプライン化を行った．これを本 FFT 乗算器における最適なパイプライン化と呼ぶ．最適なパイプライン化を行った結果，各モジュールのレイテンシ (パイプライン段数) は表 3.6 のようになった．ここに $L_{\mathrm{butterfly}}$，L_{cmul}，L_{scl}，L_{rc} は，それぞれ (inv-)butterfly，complex multiplier，scaler，rounder-carrier におけるレイテンシを表す．

表 3.6 FFT 乗算器の各モジュールにおけるレイテンシ

Pipeline	$L_{\mathrm{butterfly}}$	L_{cmul}	L_{scl}	L_{rc}
Not optimized	9	7	2	4
Optimized	22	17	2	10

FFT 乗算器の演算速度は，演算にかかる遅延時間と総クロック数の積で決まる．そこで，FFT 乗算に必要なステップ数 T_{fftmul} を求める．FFT は，n 桁どうしの乗算で $2n$ 桁の積を得るために，$2n$ 個の入力列を変換する．1 段あたり n 回のパイプライン化されたバタフライ演算が行われる．したがって 1 段あたりのステップ数は $(L_{\mathrm{butterfly}} + n - 1)$ である．これを $\log_2 2n$ 段分繰り返すので，そのステップ数

T_{fft} は，式 (3.31) となる．FFT の逆変換の計算量も同様である．

$$T_{\mathrm{fft}} = (L_{\mathrm{butterfly}} + n - 1)\log_2 2n \tag{3.31}$$

項ごとの乗算は，パイプライン化された $2n$ 回の複素数乗算を行うので，ステップ数 T_{cmul} は式 (3.32) となる．

$$T_{\mathrm{cmul}} = L_{\mathrm{cmul}} + (2n - 1) \tag{3.32}$$

シフトによる 2 のべき乗の除算は，パイプライン化された $2n$ 回のシフトを行うので，ステップ数 T_{scl} は式 (3.33) となる．

$$T_{\mathrm{scl}} = L_{\mathrm{scl}} + (2n - 1) \tag{3.33}$$

各桁の丸めと桁上げは，パイプライン化されていない $2n$ 回の演算を行うので，ステップ数 T_{rc} は式 (3.34) となる．

$$T_{\mathrm{rc}} = L_{\mathrm{rc}}(2n - 1) \tag{3.34}$$

式 (3.31)，(3.32)，(3.33)，(3.34) から，FFT 乗算器全体のステップ数 T_{fftmul} は乗数順変換，被乗数順変換，逆変換に必要な 3 回分の T_{fft} と，その他のステップ数をすべて 1 回分ずつ加算したものとなるので，式 (3.37) となる．

$$\begin{aligned} T_{\mathrm{fftmul}} &= 3T_{\mathrm{fft}} + T_{\mathrm{cmul}} + T_{\mathrm{scl}} + T_{\mathrm{rc}} && (3.35)\\ &= 3(L_{\mathrm{butterfly}} + n - 1)\log_2 2n + L_{\mathrm{cmul}} + (2n - 1) + L_{\mathrm{scl}} \\ &\quad + (2n - 1) + L_{\mathrm{rc}}(2n - 1) && (3.36)\\ &= 3(L_{\mathrm{butterfly}} + n - 1)\log_2 n + (2L_{\mathrm{rc}} + 7)n \\ &\quad + (3L_{\mathrm{butterfly}} + L_{\mathrm{cmul}} + L_{\mathrm{scl}} - L_{\mathrm{rc}} - 3) && (3.37) \end{aligned}$$

なお，各モジュールが memory モジュールへアクセスするための時間 (キャッシュレイテンシ) は，memory モジュールが同一チップ内に配置されているため短く，本設計では各モジュールでの演算とオーバラップさせているので，式 (3.37) には現れない．

次に，最適化実装を行った FFT 乗算器の最大遅延時間を求めるため，表 3.7 に最適なパイプライン化を行った後の各モジュールの面積と遅延時間をまとめる．表から，FFT 乗算器の最大遅延時間は 1.89ns であることがわかる．

以上の結果から，FFT 乗算器の性能が求まる．まず，式 (3.37) より，2^{13} 桁における総クロック数は 541,563 である．これに，最大遅延時間である 1.89ns を乗じると，演算時間は 1.02ms である．一方，パイプライン化前の総クロック数は 442,709，遅延時間は 7.63ns であるので，演算時間は 3.38ms である．このことから，最適なパイプライン化を行うことにより，約 3.3 倍の速度向上が得られることがわかった．

表 3.7 最適なパイプライン化を行った FFT 乗算器における各モジュールの面積と最大遅延時間 ($n = 2^{13}$)

module	area[mm^2]	delay[ns]
complex mul	2.01	1.89
butterfly	2.98	1.81
inv-butterfly	3.06	1.81
scaler	0.02	1.26
rounder-carrier	0.38	1.74
memory	0.60	1.60
all	9.05	1.89

前節で述べたパイプライン化前の面積は 6.55mm^2 であった，表 3.7 より，パイプライン化後の面積は 9.05 mm^2 である．したがって，パイプライン化により面積は 38% 増加したことになる．しかし，同世代の汎用プロセッサの面積が約 200mm^2 [45] であることを考えると，パイプライン化後の面積はこの 5% 程度にすぎない．

3.6.4 ソフトウェア FFT 乗算との速度比較

前章の考察から，2^5 桁 (10 進数 39 桁) から 2^{13} 桁どうしの乗算に必要な演算時間を求め，ソフトウェアとの速度比較をおこなった．2^{13} 桁以下の FFT 乗算器は，2^{13} 桁の場合より小さい回路規模で実現できるため，遅延時間は 2^{13} 桁以下のすべての桁数で 1.89ns とした．

ソフトウェアを実行する計算機の条件として，CPU に本ハードウェア実装と同世代の 0.18μm プロセスを使用している Intel 社 [46] の Pentium 4 1.7GHz，OS に FreeBSD 5.4，コンパイラに gcc 3.4.2 を用いた．また，メモリの容量は 512 MB である．FFT には FFTW 3.0.1 を用いた．FFTW [32] は様々な種類の FFT を複数の計算機アーキテクチャ用に最適化して実装した高速な FFT ライブラリである．今回は FFT の対象となるデータが整数ベクトルであるため，複素数 FFT と比較して速度や使用メモリ量の面で有利な実数 FFT を用いた．比較結果を表 3.8 に示す．ソフトウェアとハードウェアの速度倍率が一定でないのは，ソフトウェアで使われている FFTW が 1,000 から 10,000 点の FFT において高い性能を発揮するように実装されているからであると思われる [47]．表から速度倍率は，2^5 から 2^{13} の範囲で，19.7 倍から 34.3 倍，平均は約 25.7 倍であることがわかった．

3.6.5　ソフトウェア実装された他の演算法との比較

前節と同様の計算機環境において，Karatsuba 法を用いた高速な多倍長乗算を提供するソフトウェア exflib[16] とソフトウェア FFT 乗算の演算速度を比較した．exflib のバージョンは 20050709 である．その他の条件は前節と同様である．結果を両対数グラフにして図 3.21 に示す．

exflib のグラフの傾きは 1.61 であった．これは，Karatsuba 法の計算時間 $O(n^{1.585})$ に近い．また，グラフから約 2^{21} 桁以上で FFT 乗算が有利になることが読み取れる．この時の exflib の乗算時間は 16.5s であった．

そこで，2^{21} 桁において，ハードウェア FFT 乗算と exflib によるソフトウェア Karatsuba 乗算の速度比較を行った．図 3.6 から 2^{21} 桁の乗算に必要な指数部長と仮数部長を求めたところ，指数部が 9 ビット，仮数部が 40 ビットであった．このビット長で最適なパイプライン化を行った FFT 乗算器を設計したところ，各演算器のレイテンシは表 3.9 のようになった．また，各モジュールの最大遅延時間と面積は表 3.10 のようになった．また， 表 3.9 に示したレイテンシと式 (3.37) から，2^{21} 桁の乗算に必要なクロック数は 197,134,081 であった．これに，表 3.10 が示す最大遅延時間 1.96ns を乗じると，0.39s になる．一方，2^{21} 桁における exflib の乗算時間は 16.5s であった．よって，本研究で実装した FFT 乗算器は，FFT 乗算が Karatsuba 乗算より高速になる 2^{21} 桁において，42 倍高速であることがわかる．また同じ桁数において FFTW による FFT 乗算の実行時間は 13.9s であった．よって FFTW との性能差は 35 倍となる．

表 3.8　ソフトウェアとハードウェアによる FFT 乗算のパフォーマンス比較

Digits	Soft[ms]	Hard[ms]	Soft/Hard
2^5	0.0917	0.0033	27.8
2^6	0.1242	0.0063	19.7
2^7	0.3924	0.0126	31.1
2^8	0.8854	0.0258	34.3
2^9	1.4930	0.0535	27.9
2^{10}	2.6840	0.1116	24.1
2^{11}	5.2400	0.2337	22.4
2^{12}	10.8000	0.4893	22.1
2^{13}	22.6200	1.0236	22.1

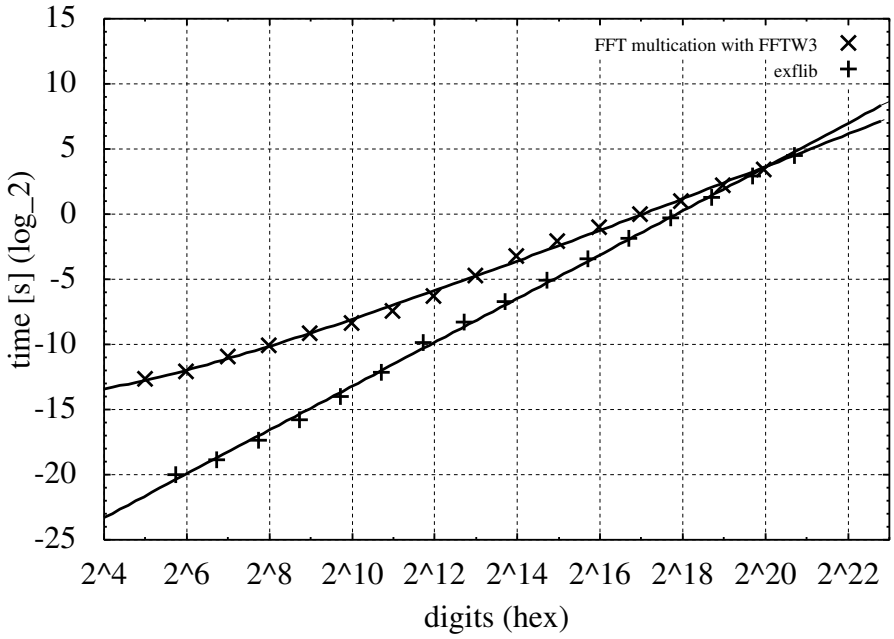

図 3.21 FFTW による FFT 乗算と exflib による Karatsuba 乗算の速度比較

表 3.9 $n = 2^{21}$ における FFT 乗算器の各モジュールのレイテンシ

Pipeline	$L_{\mathrm{butterfly}}$	L_{cmul}	L_{scl}	L_{rc}
Optimized	28	21	2	12

3.6.6 既存の FFT ハードウェアとの関連と他のハードウェア乗算器との比較

過去に発表されている FFT ハードウェアには様々な種類がある．これらは，バタフライ演算器とメモリを実装したものである．これらの実装は，データ長やデータ表現については，主な使用目的が DSP であるので，乗算に必要な精度が考慮されていない．必要な精度を満たしていると仮定すれば，complex multiplier，scaler，rounder-carrier を追加することで乗算が可能となる．また，アーキテクチャによっては，バタフライ演算器で使われている複素数乗算器を流用することも考えられるので，その場合には complex multiplier は不要となる．

図 3.21 より，2^{21} 桁以下では，FFT 乗算より，exflib の方が高い性能を持つ．よって，exflib で使われている Karatsuba 法をハードウェア実装するという選択もある．ここでは，ハードウェア FFT 乗算器と 32 ビットハードウェア Karatsuba 乗算器

表 3.10　最適なパイプライン化を行った FFT 乗算器における各モジュールの面積と最大遅延時間 ($n = 2^{21}$)

module	area[mm^2]	delay[ns]
complex mul	3.79	1.96
butterfly	5.35	1.96
inv-butterfly	5.47	1.93
scaler	0.04	1.32
rounder-carrier	0.61	1.76
memory	0.84	1.63
all	16.1	1.96

[23] の面積コストを比較する．比較にあたり，広い応用範囲を持つ FFT ハードウェア (butterfly，inv-butterfly，complex multiplier) はすでに用意されているものとし，コストとしてカウントしない．このとき，32 ビット (16 進数 2^3 桁) FFT 乗算器の面積 (scaler と rounder-carrier モジュールの合計面積) が 0.15mm^2 であるのに対し，32 ビット Karatsuba 乗算器の面積は，0.18μm テクノロジに換算すると，最も小さいもので 0.10mm^2 である．すなわち，32 ビットにおいては，FFT 乗算器と Karatsuba 乗算器の面積コストは同程度と言える．より詳しい比較には 2^{21} 桁までのハードウェア Karatsuba 乗算器のコスト評価が必要である．これについては次章で述べる．また，この比較から，FFT ハードウェアを乗算に流用した場合に追加が必要なハードウェアの面積コストは小さいこともわかる．

3.7　カスタムチップの試作

ハードウェア FFT 乗算器の実装可能性を検討するため，実際にカスタムチップによる試作を行った．

3.7.1　試作の条件と方針

設計した FFT 乗算器をカスタムチップとして試作するためには，入出力や電源を含めた VLSI のレイアウトが必要になる．レイアウトの例を図 3.22 に示す．カスタムチップ化する VLSI は，その内周に，入出力用の I/O パッド (I/O pad) と I/O セル (I/O cell) が並べられる．さらに内側に電源リング (VCC ring) とグラウンドリング (GND ring) を配置し，そのさらに内側が回路を配置するコア部 (Core) とな

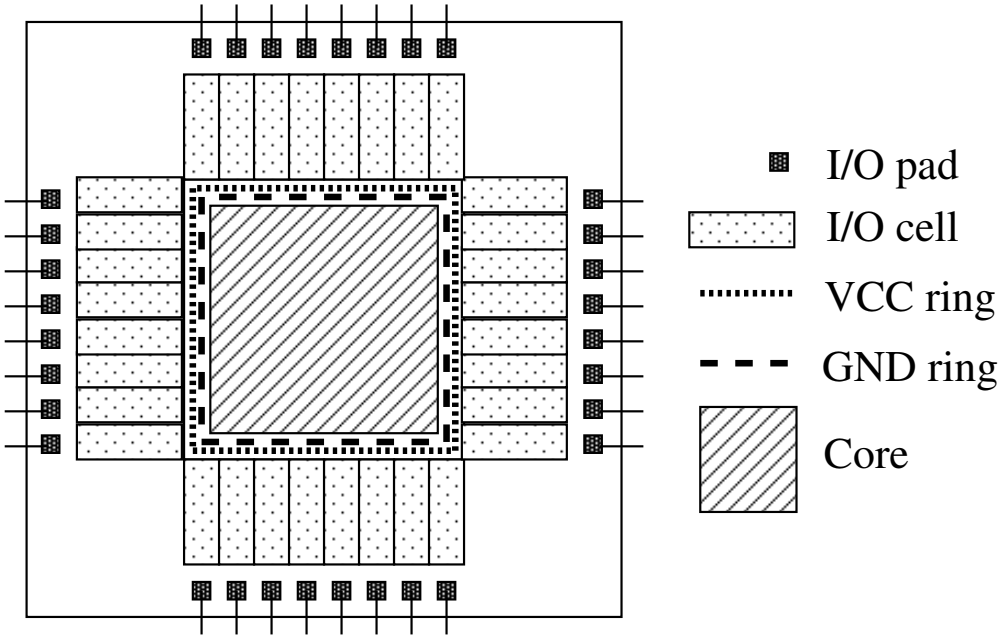

図 3.22　レイアウトの例

る．このコア部の面積は，I/O パッドや I/O セルの大きさなどの設計規則に依存するため，実際に定められたデザインルールにしたがった配置配線作業を行なわなければ見積もることができない．そこで，FFT 乗算器を実装した場合の配置面積と，それに対応するチップの大きさを見積もるため，VLSI の試作を行った．今回行ったような専用チップの試作には，回路面積の大きさによって多大な費用が必要となる．そこで，回路面積を削減するため，データ表現に 16 ビットの浮動小数点表現を用いた．さらに実装する FFT 乗算器は 16 進数 2 桁 × 2 桁の演算を行う 2 桁版の 16 ビット FFT 乗算器とした．この小型の FFT 乗算器をカスタムチップ化し，実際の配置面積を調べ，その配置面積から，より実用的な FFT 乗算器の配置面積やチップサイズを見積もる．表 3.11 にカスタムチップの試作環境を示す．

表 3.11　チップ試作環境

配置配線ツール	Synopsys 社 ApolloII 2000.2.3.4.0.9
検証ツール	Cadence 社 Dracula 4.9.05-2002
Layout editor	Cadence 社 Virtuoso 5.0
Technology	日立製作所 CMOS 0.18μm
配線層	Poly Si 1 層，メタル 5 層
外寸	2.8mm 角

3.7.2　データ表現形式

試作する FFT 乗算器に用いる 16 ビットの浮動小数点表現のビットパターンは，符号ビット，指数部が 5 ビット，仮数部が 10 ビットとなっている．これは，2 桁どうしの FFT 乗算の演算過程において発生する最大値を表現できることを考慮して指数部を 5 ビットとし，さらに十分な精度を確保するために仮数部を 10 ビットとしたものである．第 3.4.2 節で述べたように，FFT 乗算は最大値どうしの演算によって最大の誤差が発生する．したがって，16 進数 2 桁版 FFT 乗算器の場合，FF × FF の乗算が正確に行われれば誤差の影響を受けずに正しい演算が行われることが保証される．これはシミュレーションにおいて確認済みである．また，これ以外の仕様は今まで述べてきた本実装における浮動小数点表現と同様である．

3.7.3　16 進数 2 桁版 16 ビット FFT 乗算器の面積

16 進数 2 桁版 FFT 乗算器を論理合成した結果から求めた面積を表 3.12 に示す．表から memory と controller を含む FFT 乗算器全体の面積が 1.40mm^2 であるこ

表 3.12　2 桁版 16 ビット FFT 乗算器の面積

モジュール	面積 [mm^2]
complex multiplier	0.24
butterfly	0.36
inv-butterfly	0.37
scaler	0.01
rounder carrier	0.04
controller	0.32
memory	0.06
all	1.40

とがわかる．この面積がコア部の面積以下であれば試作の面積的な条件を満たしていることになる．

3.7.4　VLSI のレイアウト

論理合成によって得られた FFT 乗算器のネットリストを配置配線した結果を図 3.23 に示す．配置した結果，コア部の面積は，全 VLSI 面積の 20% であり，約 1.56

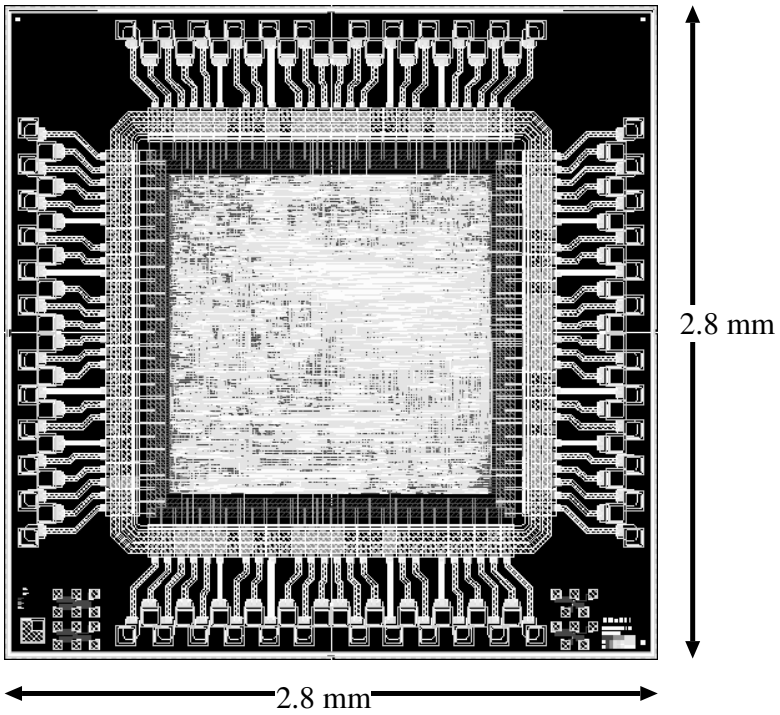

図 3.23　FFT 乗算器の VLSI レイアウト図

mm^2 であった．よって，16 ビット版 FFT 乗算器は 2.8mm 角のチップに十分実装可能である．

3.7.5　試作結果

図 3.24 に試作した VLSI を示す．図 3.23 とくらべると，FFT 乗算器が実装されているコア部とそのまわりにある I/O パッドが確認できる．試作した VLSI をチップとしてボード上に組み込むためには，入出力用のピンと I/O パッドを配線しなければならない．次に，試作した VLSI と，それをチップとしてパッケージングしたものを，図 3.25 に示す．このように，16 ビット版 FFT 乗算器を実際のカスタムチップに実装することができた．

第 3.6.5 節で述べた．2^{21} 桁の乗算を行う FFT 乗算器の面積は 16.1 mm^2 であった．これがカスタムチップのコア部に収まり，全 VLSI 面積の 20% を占めると考え

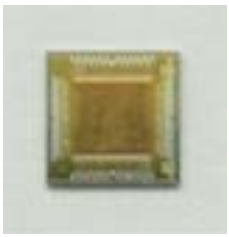

図 3.24　試作したチップ

図 3.25　試作した VLSI (左) とパッケージングされたチップ (右)

ると，残りの 80% の面積は 64.0mm^2 となる．これらの面積を加算すると 80mm^2 であるので，9mm 角程度の VLSI に実装可能であることがわかる．このことから，より実用的な FFT 乗算器であっても，9mm 角程度の大きさで VLSI に実装可能であることがわかった．実際には VLSI のサイズが大きくなれば，コア部が占有できる面積は 20% より大きくなるため，9mm 角よりもさらに小さなサイズで実装が可能であると考える．

3.8 本章のまとめ

本章では，FFT による乗算アルゴリズムのハードウェア実装と評価を行った．以下にその内容をまとめる．

3.8.1 FFT 乗算器におけるデータ表現の最適化

実験的な誤差解析に基づいて，FFT 乗算に用いる浮動小数点表現の最適なデータ長を求めた．最適なデータ表現を求めるために，最大値どうしの乗算が最大の誤差を与えるという点に着目し，r 進数 n 桁における最大値どうしの乗算で正しい結果が得られるようなデータ長を測定し，そのデータ長でハードウェアを構成した．この最適なデータ表現を用いて 16 進数 2^{13} 桁の乗算を行う FFT 乗算器を実装したところ，最大遅延時間と面積はそれぞれ 7.63ns と 6.55mm^2 であった．一方，標準的な IEEE754 64 ビット浮動小数点表現と同じビット長を用いて実装した FFT 乗算器の最大遅延時間と面積はそれぞれ 10.3ns と 16.0mm^2 であった．このことから，誤差に着目して最適な実装を行うことで，遅延時間を 26%，面積を 60 % 削減した FFT 乗算器を実装することができた．

3.8.2 FFT 乗算器の最適なパイプライン化

作成した FFT 乗算器に最適なパイプライン化をおこなった．最適なパイプライン化を行った後の FFT 乗算器の最大遅延時間は 1.89ns であった．これに 16 進数 2^{13} 桁の乗算を 1 回行うために必要なクロック数 541,563 を掛け合わせることで，演算時間は 1.02ms であることがわかった．一方，パイプライン化前の FFT 乗算器が 1 回の乗算を行うために必要な時間は 3.38ms であった．これにより，最適なパイプライン化によって，3.3 倍の性能を持つ FFT 乗算器を実装することができた．なお，パイプライン化後の FFT 乗算器の面積は 9.05 mm^2 であり，これは一般的な汎用プロセッサの 5% 程度であった．

3.8.3 ソフトウェア FFT 乗算との速度比較

ソフトウェアによる FFT 乗算との速度比較を行った．ソフトウェアの実行には，ハードウェア FFT 乗算器と同じテクノロジで実装された Pentium 4 1.7GHz を用いた．また FFT ライブラリとして，高速な FFTW を用いた．比較の結果，ハードウェア FFT 乗算器の速度は，ソフトウェアと比較して 16 進数 2^5 桁から 2^{13} 桁の

乗算において，19.7 倍から 34.3 倍，平均すると 25.7 倍高速であることがわかった．

3.8.4　ソフトウェア Karatsuba 乗算との速度比較

FFT 乗算が，Karatsuba 乗算とほぼ同じ速度になる 16 進数 2^{21} 桁 (10 進数約 252 万桁) 以上の乗算において，ソフトウェア実装された Karatsuba 乗算 (exflib) と FFT 乗算器の速度を比較した．比較にあたり，2^{21} 桁の乗算を行う最適な FFT 乗算器を実装した．その結果，ハードウェア FFT 乗算器の乗算時間が 0.39s，exflib による乗算時間が 16.5s であった．この結果から，ハードウェア FFT 乗算は，exflib と比較して 42 倍の性能を実現できることがわかった．また同じ桁数において FFTW による FFT 乗算の実行時間は 13.9s であった．よって FFTW との性能差は 35 倍となる．

3.8.5　既存の FFT ハードウェアとの関連と他のハードウェア乗算器との比較

FFT ハードウェア (butterfly，inv-butterfly，complex multiplier) はすでに用意されているものとし，コストとしてカウントしないという条件で，ハードウェア FFT 乗算器と 32 ビットハードウェア Karatsuba 乗算器 [23] の面積コストを比較した．その結果，32 ビット (16 進数 2^3 桁) FFT 乗算器の面積 (scaler と rounder-carrier モジュールの合計面積) が 0.15mm^2 であるのに対し，32 ビット Karatsuba 乗算器の面積は，0.18μm テクノロジに換算すると，最も小さいもので約 0.10mm^2 であった．よって，32 ビットにおいては，FFT 乗算器と Karatsuba 乗算器の面積コストは同程度であることがわかった．また，この比較から，FFT ハードウェアを乗算に流用した場合に追加が必要なハードウェアの面積コストは小さいこともわかった．

3.8.6　チップ試作

最後に，16 進数 2 桁版 16 ビット FFT 乗算器をチップに実装し，試作を行った．その結果，2.8mm 角のチップサイズで実装可能であることが確認できた．この実装により，16 進数 2^{21} 桁の乗算を行う FFT 乗算器でも 9mm 角かそれ以下のチップサイズで実現できることがわかった．

第 4 章

Karatsuba 法によるハードウェア多倍長乗算器

本章では，Karatsuba 法をハードウェア実装し，その性能とコストを評価する．本章の構成は次の通りである．

第 4.1 節 Karatsuba 法を用いた乗算をハードウェア実装することの意義を述べる．
第 4.2 節 Karatsuba 法による乗算アルゴリズムを説明する．
第 4.3 節 Karatsuba 乗算器の設計にあたり，設計の選択肢を示す．
第 4.4 節 設計の 1 つの選択肢である Recursive Karatsuba 乗算器の構成法を検討し，実装結果を示す．
第 4.5 節 第 4.4 節の結果を踏まえて，設計のもう 1 つの選択肢である Iterative Karatsuba 乗算器の構成法を検討し，実装結果を示す．
第 4.6 節 本章のまとめを述べる．

4.1 Karatsuba 法による乗算ハードウェア実装の意義

本章では 2-way 法によるハードウェア Karatsuba 乗算器について述べる．2-way 法による Karatsuba 乗算は，n 桁の乗算を $O(n^{1.58})$ の時間で行うことができる．Karatsuba 法には，他にも 3-way，4-way，$\cdots$ がある．これらの方法は，2-way 法よりも小さい計算量で乗算を行うことができるが，そのアルゴリズムは複雑であり性能コスト比があまり良くない．

Karatsuba 法のハードウェア実装にあたり，$O(n^{1.58})$ である計算量を面積として消費する方法と時間として消費する方法がある．本章では，これら 2 通りの方法についてそれぞれ検討を行う．

Karatsuba 法のハードウェアについては，過去に楕円曲線暗号等に利用するため，ガロア体 (Galois Field，GF) 上の乗算を行う演算器が実装，評価された例は多い [6, 20]．しかし，整数多倍長乗算を行うハードウェア Karatsuba 乗算器の実装と評価に関する報告は少ない．過去に，32 ビット整数 Karatsuba 乗算器を実装，評価した例 [23] があるが，より大きなビット長に対応した整数 Karatsuba 乗算器に関する報告はない．本章では，アプリケーションでの利用が期待される数百ビット以上の整数 Karatsuba 乗算器をハードウェア実装し，評価した結果を述べる．

4.2　Karatsuba 乗算アルゴリズム

Karatsuba 法による乗算アルゴリズムを説明する．A と B をそれぞれ $2n$ ビットの整数とする．これらを半分に分割し，それぞれ

$$A = a_1 2^n + a_0 \tag{4.1}$$
$$B = b_1 2^n + b_0 \tag{4.2}$$

とする．すると，A と B の積 P は次式で与えられる．

$$\begin{aligned} P = A \cdot B &= (a_1 2^n + a_0) \cdot (b_1 2^n + b_0) \\ &= a_1 \cdot b_1 2^{2n} + (a_1 \cdot b_0 + a_0 \cdot b_1) 2^n + a_0 \cdot b_0 \end{aligned} \tag{4.3}$$

ここで，

$$a_1 \cdot b_0 + a_0 \cdot b_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (a_1 \cdot b_1 + a_0 \cdot b_0) \tag{4.4}$$

という関係を用いると，式 (4.3) は次のように書き換えることができる．

$$\begin{aligned} P = {} & a_1 \cdot b_1 2^{2n} \\ & + ((a_1 + a_0) \cdot (b_1 + b_0) - (a_1 \cdot b_1 + a_0 \cdot b_0)) 2^n \\ & + a_0 \cdot b_0 \end{aligned} \tag{4.5}$$

書き換え前の式 (4.3) には n ビット乗算が $a_1 \cdot b_1$，$a_1 \cdot b_0$，$a_0 \cdot b_1$，$a_0 \cdot b_0$ の 4 回あり，加算は 3 回ある．一方，式 (4.5) には n ビット乗算が $a_1 \cdot b_1$，$(a_1 + a_0) \cdot (b_1 + b_0)$，$a_0 \cdot b_0$ の 3 回あり，加算は 6 回ある．したがって，この式変形により加算の数を 3 回増やす代わりに n ビット乗算の回数を 1 回削減できる．

Karatsuba 法は，式 (4.5) に含まれる 3 回の乗算それぞれに対してさらに再帰的に適用することができる．この再帰的な適用を繰り返すことで，$O(n^2)$ であった計算量を，最終的に $O(n^{1.58})$ $(\log_2 3 \approx 1.58)$ まで削減することができる．ソフトウェア実装においては，この再帰を 1 語長の乗算になるまで繰り返す．

4.3 設計の選択肢

Karatsuba アルゴリズムのハードウェア設計において,

1. 演算器を必要な数だけ用いて組合せ回路で実現する方法 (図 4.1)
2. 固定された数の演算器 (乗算器と加算器) を繰り返し用いて順序回路で実現する方法 (図 4.2)

の 2 通りの方法が考えられる. 以降, 組合せ回路で実現されたものを RKM

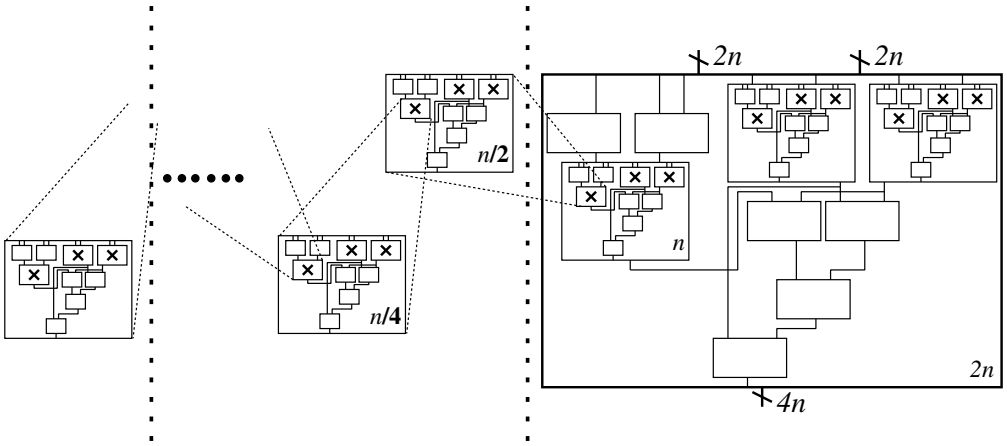

図 4.1 Recursive Karatsuba multiplier (RKM) の構成

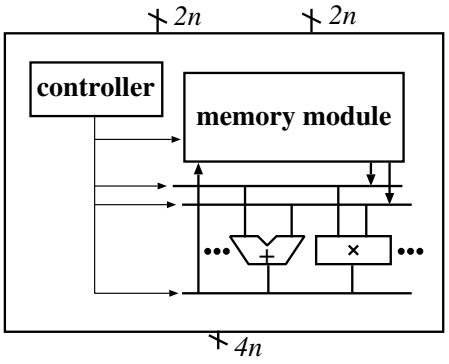

図 4.2 Iterative Karatsuba multiplier (IKM) の構成

(Recursive Karatsuba Multiplier), 順序回路で実現されたものを IKM (Iterative Karatsuba Multiplier) と呼ぶ.

RKM については，Karatsuba アルゴリズムの適用により必要な乗算器の数を減らすことができるため，一般的な乗算器として広く用いられている Wallace Tree Multiplier (WTM) と比較して，面積のオーダを $O(n^2)$ から $O(n^{1.58})$ に削減することができる．しかし，組み合わせ回路による乗算器の面積は，ビット長が大きくなると Karatsuba 法によって $O(n^{1.58})$ に削減できたとしても実際にどの程度の大きさになるのかはわからない．また，計算量の上では WTM より有利である RKM の面積がゲートレベルにおける評価でも有利であるかは，実装結果による評価を行わなければわからない．

ビット長が大きい場合は，チップ面積の制約から，組み合わせ回路による乗算器の実装には限界があるため，IKM を用いるのが妥当である．IKM については乗算器を使う回数を減らすことができるため，計算時間のオーダを $O(n^2)$ から $O(n^{1.58})$ に削減することができる．IKM は内部に用いる基本乗算器のビット長によって性能とコストが変わる．よって，基本乗算器のビット長を変えることで，性能とコストのトレードオフが可能である．またその時，基本乗算器を RKM と WTM のどちらで実現するべきかについての検討が必要である．以上に述べた，RKM と IKM の特徴を表 4.1 にまとめる．

表 4.1　RKM (Recursive Karatsuba Multiplier) と IKM (Iterative Karatsuba Multiplier) の比較 († 実装に依存する)

	回路構成	面積	乗算時間	適用範囲
RKM	組み合わせ回路	$O(n^{1.58})$	$O(\log n)$	比較的小さなビット長
IKM	順序回路	固定 †	$O(n^{1.58})$	大きなビット長

本章では，上に述べた点を明らかにするため，RKM と IKM の両方について Karatsuba アルゴリズムの適用回数や基本乗算器のビット長を変えて VLSI 実装を行い，性能とコストを評価する．

4.4 組み合わせ回路による Karatsuba 乗算器 (Recursive Karatsuba Multiplier, RKM)

本章では，RKM の設計について述べる．はじめに，Karatsuba アルゴリズムを一度だけ適用した 32 ビットの RKM を実装し，文献 [23] の結果と比較する．次に，RKM の設計をより一般的な形で行う．

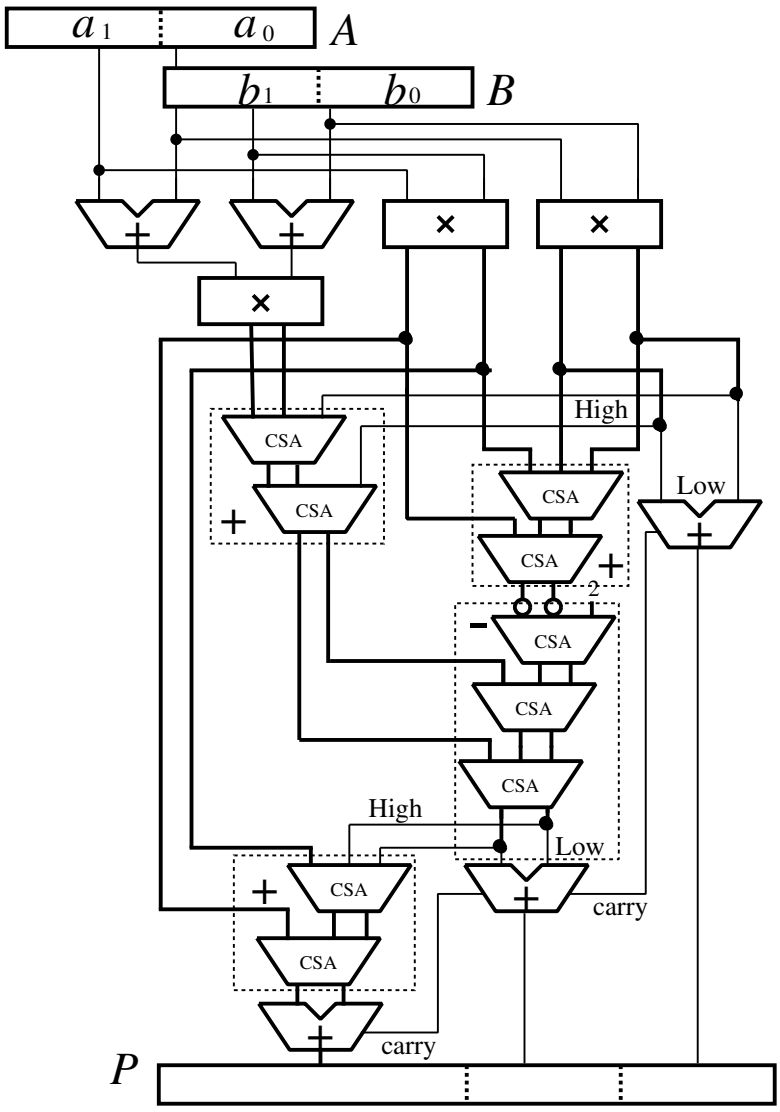

図 4.3 Carry-Save RKM の構成

4.4.1 32 ビット RKM

設計

式 (4.5) を基に設計した RKM の構成を図 4.3 に示す．RKM 内で用いる乗算器には WTM を用いる．アルゴリズムを一度だけ適用するため，WTM のビット長は入力の半分である 16 になる．また，WTM の出力を累算する加算器として，CSA (Carry Save Adder) を用いる．CSA を用いることで，中間値の加算において桁上げを伝搬する必要がなく，遅延時間を削減することができる．ただし，最終的な積を出力する最終段の加算器には CPA (Carry Propagation Adder) を用いる．この構

成を Carry-Save RKM と呼ぶ．比較のため，全ての加算を CPA で行う RKM も設計する．構成を図 4.4 に示す．この構成を Binary RKM と呼ぶ．なお，RKM 内部

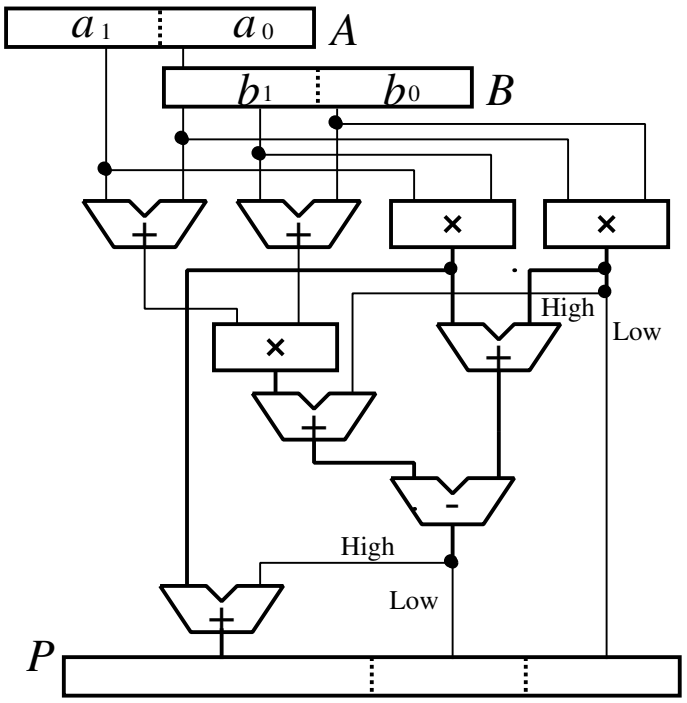

図 4.4 Binary RKM の構成

で用いる WTM と CPA は，Synopsys 社の提供する IP (Intellectual Property) から，DW02_mult，DW01_add (DW01_sub) をそれぞれ用いる．なお，CSA と CPA の詳細な構造については付録 A に示す．

実装結果と考察

設計した Carry-Save RKM と Binary-RKM を Verilog-HDL で記述し，Design Compiler Ver. W-2004.12-SP2 を用いて論理合成を行い，その結果から最大遅延時間と面積を評価した．論理合成に用いたセル・ライブラリは，前章と同様に日立製作所の 0.18μm プロセスルールで作成されたものである．合成時の制約条件としては，回路面積が最小となるような条件を与えた．合成結果を表 4.2 に示す．

さらに，文献 [23] との比較を行うため，Carry-Save RKM に対して ROHM 社の仕様に基づき 0.35μm プロセスルールを用いて作成されたセル・ライブラリを用いた論理合成も行った．この時，最大遅延時間を最小にする制約条件を与えた．文献 [23] の結果との比較を表 4.3 にまとめる．

表 4.2 より，最終的な加算以外を CSA で行うことで，最大遅延時間を 22% 削減

表 4.2 HITACHI 0.18μm ライブラリを用いた 32 ビット RKM の合成結果 (面積優先)

	delay[ns]	area[mm^2]
Binary RKM	12.03	0.216
Carry-Save RKM	9.44	0.228

表 4.3 ROHM 0.35μm ライブラリを用いた 32 ビット RKM の合成結果 (遅延時間優先)

	delay[ns]	area[mm^2]
Carry-Save RKM	8.95	0.741
Ref.[23]	7.70	0.580

できることがわかった．ただし，面積コストは 6% 増加している．これは，桁上げを伝搬せず Carry Save 形式で値を表現しているため，桁上げを伝搬させて通常の 2 進数で演算を行った場合と比較して，多くの全加算器が必要となるためである．面積の削減を主な目的としている RKM において，このように面積が増加するような回路構成は，一見，本来の目的に合わないが，増加する面積に対して，遅延時間の削減量が大きいため，性能コスト比においては，良い設計であるといえる．したがって，以降の設計においても最終段以外の加算を CSA で行う回路構成を採用する．

また，表 4.3 から今回設計した Carry-Save RKM は，文献 [23] で示されている RKM より，最大遅延時間が 1.16 倍，面積が 1.28 倍であることがわかる．この理由として，文献 [23] では，最終段の CPA を 2 ビットまたは 3 ビットごとに人手で最適化したのに対し，本設計では IP を用いた自動合成を行ったためと考えられる．

4.4.2 RKM の一般形

本節では，より一般的な形で RKM を設計する．

設計

乗算ビット長を $2^k = 2n$ とし，RKM の基本となる WTM のビット長を 2^l とすると，再帰の回数は $k - l$ 回となる．ただし，k，l $(0 < l < k)$ は整数である．前章で述べた 32 ビット RKM は基本乗算器として 16 ビット WTM を採用している．したがって，$k = 5$，$l = 4$ の特別なケースであるといえる．

一般的な形の RKM を設計するにあたり，回路を次に述べる 3 種類のコンポーネントに分けて設計を行った．

- 最終的な積を得るために Carry Save (CS) 形式の積を CPA を用いて Binary (B) 形式に変換する機構を持つコンポーネント
- CPA を必要としないコンポーネント
- 2^l ビットの WTM を持つコンポーネント

これらのコンポーネントは RKM のモジュール構成において上層，中間層，最下層にあたるため，それぞれ T (Topmost)，I (Intermediate)，U (Undermost) と表す．加えて，コンポーネント I，U は，それぞれ，入力値を CS 形式で受け取るもの (I_{cs}，U_{cs}) と，B 形式で受け取るもの (I_b，U_b) の 2 通りがあるため，それぞれに添字 cs，b を付けて表現する．特に，CS 形式で入力値を受け取る U コンポーネント (U_{cs}) には CS 形式で入力された値を B 形式に変換してから WTM へ入力するため，CPA が必要となる．以上をまとめると，計 5 種類のモジュールが必要となる．各モジュールの構成を図 4.5，4.6，4.7，4.8，4.9 に示す．図中に網かけで示された V モジュールは，式 (4.5) にしたがって桁上げ保存加算を行うモジュールである．これらは全てのコンポーネントで共通のため，V モジュールとして表記する．　図中の KaratsubaCS と KaratsubaB は，再帰の回数によって，表 4.4 に示す各モジュールに置き換えられる．例えば，1 回再帰の場合，T モジュール中の

表 4.4　再帰回数によるモジュールの置き換え

number of recursions	1	≥ 2
KaratsubaCS	U_{cs}	I_{cs}
KaratsubaB	U_b	I_b

KaratsubaCS と Karatsuba B はそれぞれ U_{cs} と U_b に置き換えられる．これにより，前章で示した図 4.3 の RKM が得られる．2 回再帰以上の場合，T モジュール中の KaratsubaCS と Karatsuba B はそれぞれ I_{cs} と I_b に置き換えられ，さらに，I_{cs} と I_b 中の KaratsubaCS と KaratsubaB はそれぞれ U_{cs} と U_b に置き換えられる．

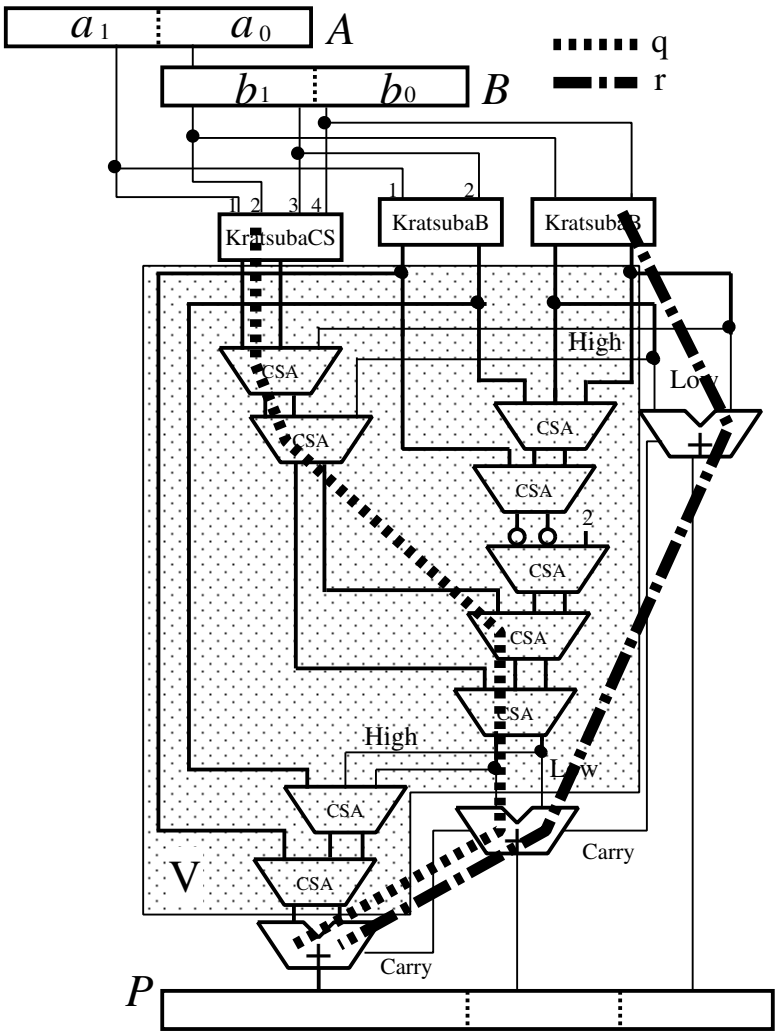
a_1
a_0
A
b_1
b_0
B
q
r
1 2 3 4
1 2
KratsubaCS
KratsubaB
KratsubaB
CSA
CSA
CSA
CSA
CSA
2
CSA
CSA
CSA
High
Low
High
Low
CSA
CSA
V
Carry
Carry
P

図 4.5　T の構成

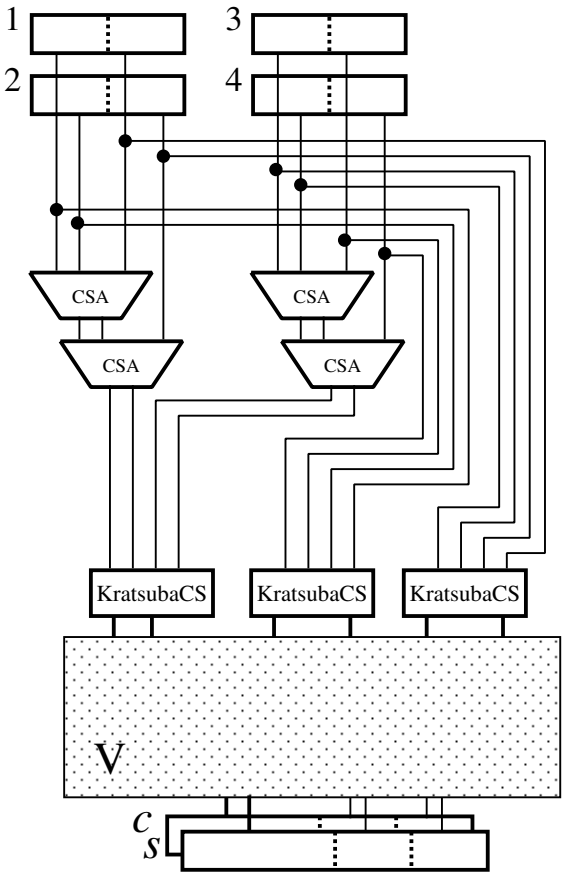

図 4.6　I_{cs} の構成

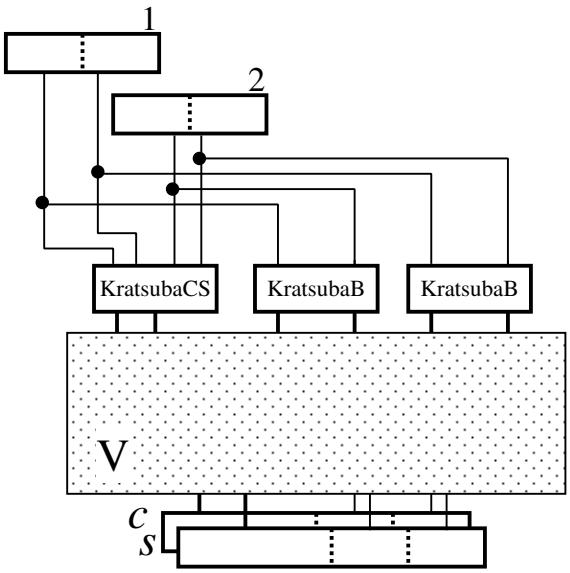

図 4.7　I_{b} の構成

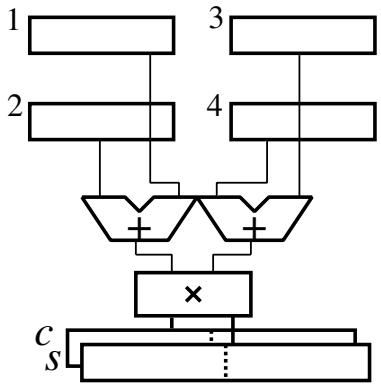

図 4.8 U_{cs} の構成

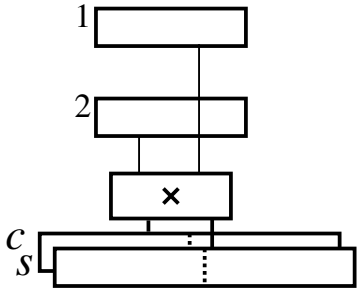

図 4.9 U_{b} の構成

最大遅延時間と面積

設計を行った RKM の最大遅延時間と面積は，構成要素の最大遅延時間と面積から計算により見積もることができる．これにより，任意のビット長，再帰の回数，基本乗算器のビット長において RKM の最大遅延時間と面積を見積もることができる．

図 4.5，4.6，4.7，4.8，4.9 より，この RKM の最大遅延時間と面積は，それぞれ，式 (4.6) から式 (4.12) と式 (4.13) から式 (4.17) で見積もることができる．

$$DT(k) = \mathrm{MAX}(DT^{\mathrm{q}}(k),\ DT^{\mathrm{r}}(k)) \tag{4.6}$$
$$DT^{\mathrm{q}}(k) = DI_{\mathrm{cs}}(k-1) + 4 \cdot D_{\mathrm{CSA}}(k) + D_{\mathrm{CPA}}(k-1) + D_{\mathrm{CPA}}(k) \tag{4.7}$$
$$DT^{\mathrm{r}}(k) = DI_{\mathrm{b}}(k-1) + 2 \cdot D_{\mathrm{CPA}}(k-1) + D_{\mathrm{CPA}}(k) \tag{4.8}$$
$$DI_{\mathrm{cs}}(j) = DI_{\mathrm{cs}}(j-1) + 2 \cdot D_{\mathrm{CSA}}(j-1) + 6 \cdot D_{\mathrm{CSA}}(j) \tag{4.9}$$
$$DI_{\mathrm{b}}(j) = DI_{\mathrm{cs}}(j-1) + 6 \cdot D_{\mathrm{CSA}}(j) \tag{4.10}$$
$$DU_{\mathrm{cs}}(l) = D_{\mathrm{CPA}}(l) + D_{\mathrm{W}}(l) \tag{4.11}$$
$$DU_{\mathrm{b}}(l) = D_{\mathrm{W}}(l) \tag{4.12}$$

$$AT(k) = AI_{\mathrm{cs}}(k-1) + 2 \cdot AI_{\mathrm{b}}(k-1) + 9 \cdot A_{\mathrm{CSA}}(k) + 2 \cdot A_{\mathrm{CPA}}(k-1) + A_{\mathrm{CPA}}(k) \tag{4.13}$$
$$AI_{\mathrm{cs}}(j) = 3 \cdot AI_{\mathrm{cs}}(j-1) + 4 \cdot A_{\mathrm{CSA}}(j-1) + 9 \cdot A_{\mathrm{CSA}}(j) \tag{4.14}$$
$$AI_{\mathrm{b}}(j) = AI_{\mathrm{cs}}(j-1) + 2 \cdot AI_{\mathrm{b}}(j-1) + 9 \cdot A_{\mathrm{CSA}}(j) \tag{4.15}$$
$$AU_{\mathrm{cs}}(l) = 2 \cdot A_{\mathrm{CPA}}(l) + A_{\mathrm{W}}(l) \tag{4.16}$$
$$AU_{\mathrm{b}}(l) = A_{\mathrm{W}}(l) \tag{4.17}$$

ここに，$DT(k)$ と $AT(k)$ は 2^k ビット RKM 全体の最大遅延時間と面積を表す．$DI_{\{\mathrm{cs},\ \mathrm{b}\}}(j)$ と $AI_{\{\mathrm{cs},\ \mathrm{b}\}}(j)$ は I コンポーネントの最大遅延時間と面積，$DU_{\{\mathrm{cs},\ \mathrm{b}\}}(l)$ と $AU_{\{\mathrm{cs},\ \mathrm{b}\}}(l)$ は U コンポーネントの最大遅延時間と面積をそれぞれ表す．ただし，$0 < l < j < k$ である．$D_{\mathrm{CSA}}(i)$ と $D_{\mathrm{CPA}}(k)$ は，それぞれ，2^i ビット CSA と 2^k ビット CPA の遅延時間を意味する．同様に，$A_{\mathrm{CSA}}(i)$ と $A_{\mathrm{CPA}}(k)$ は 2^i ビット CSA と 2^k ビット CPA の面積を表す．ただし，i は正整数である．$D_{\mathrm{W}}(l)$ と $A_{\mathrm{W}}(l)$ は 2^l ビット WTM の最大遅延時間と面積を表す．ただし，$j-1=l$ となる場合，すなわち，最後の再帰は，U コンポーネントとなるため，$DI_{\mathrm{cs}}(l) = DU_{\mathrm{cs}}(l)$，$AI_{\mathrm{cs}}(l) = AU_{\mathrm{cs}}(l)$ となる．

T コンポーネントのクリティカルパスは，図 4.5 に示すパス q および r のどちらかである．V モジュールの中を通るパス q は，4 段の CPA を経由する．対して，パス r は 1 段の CPA を経由し，パス q と合流する．また，再帰をより深くたどってゆくと，パス q と r はどちらも，U_{cs} コンポーネントを経由していることがわかる．したがって，パス q と r の違いとは，CSA を 4 段経由するか，CPA を 1 段経由す

るかの違いになる．よって，CSA 4 段分の遅延時間が CPA 1 段分の遅延時間よりも大きい場合はパス q がクリティカルパスとなり，そうでない場合はパス r がクリティカルパスとなる．CSA の遅延時間が固定であるのに対して，CPA の遅延時間はビット長や構成によって大きく異なる．よって，式 (4.7)，(4.8) では，パス q と r の遅延時間をそれぞれ $DT^{\mathrm{q}}(k)$ と $DT^{\mathrm{r}}(k)$ で表し，式 (4.6) でより大きい方を最大の遅延時間としている．I，U コンポーネントの最大遅延時間および T，I，U コンポーネントの面積は一意に定まるため 1 つの式で表すことができる．

実装結果と考察

式 (4.6) から (4.17) を用いて，2^9 ビット (512 ビット) までの RKM の面積とコストを評価した．基本となる乗算器として，16 ビット WTM ($l = 4$) を用いた．この時，16 ビット乗算器の遅延時間と面積は，それぞれ $D_{\mathrm{W}}(2^4) = 4.15\mathrm{ns}$ と $A_{\mathrm{W}}(2^4) = 0.054\mathrm{mm}^2$ であった．なお，測定時に行った論理合成の条件は，第 4.4.1 節で行った，日立製作所の $0.18\mu\mathrm{m}$ プロセスルールを用いたセルライブラリによる合成と同様である．合成時の制約も，同じく，回路面積が最小となるような条件を与えた．

CPA の遅延時間と面積は，Synopsys 社が提供するライブラリに含まれる加算器 (DW01_add) を論理合成した結果から求めた．この時，加算器の構成は CLA (Carry Lookahead Adder) とした．結果を表 4.5 にまとめる．

表 4.5　CPA (DW01_add) の最大遅延時間と面積

bit length	2^4	2^5	2^6	2^7	2^8	2^9
delay [ns]	1.38	1.82	2.31	3.02	3.97	5.21
area [mm^2]	0.006	0.012	0.026	0.057	0.126	0.275

CSA の最大遅延時間と面積は，16 ビットの場合で，0.27ns と $0.004\mathrm{mm}^2$ であった．CSA の最大遅延時間は，ビット長が増加しても変化しない．ただし，面積については，ビット長に比例するものとして計算した．

前述の式とパラメータを基に，2^5 ビットから 2^9 ビットまでの RKM の最大遅延時間と面積を見積もった．結果を表 4.6 に示す．さらに比較のため，表 4.7 に Synopsys 社が提供するライブラリに含まれる乗算器 (DW02_mult) の合成結果を示す．合成条件は RKM と同様である．

次に，面積と最大遅延時間の変化をグラフ化したものをそれぞれ図 4.10 と図 4.11 に示す．なお，両図中の x 軸は 2 を底とするビット長の対数を表す．また，図 4.10

表 4.6　RKM 最大遅延時間と面積

bit length	2^5	2^6	2^7	2^8	2^9
delay [ns]	9.81	13.37	16.95	21.21	26.24
area [mm^2]	0.270	0.972	3.275	10.602	33.469

表 4.7　WTM の最大遅延時間と面積

bit length	2^4	2^5	2^6	2^7
delay [ns]	4.15	5.44	6.90	8.09
area [mm^2]	0.054	0.184	0.675	2.574

の y 軸は 2 を底とした面積の対数を，図 4.11 の y 軸は遅延時間を表す．

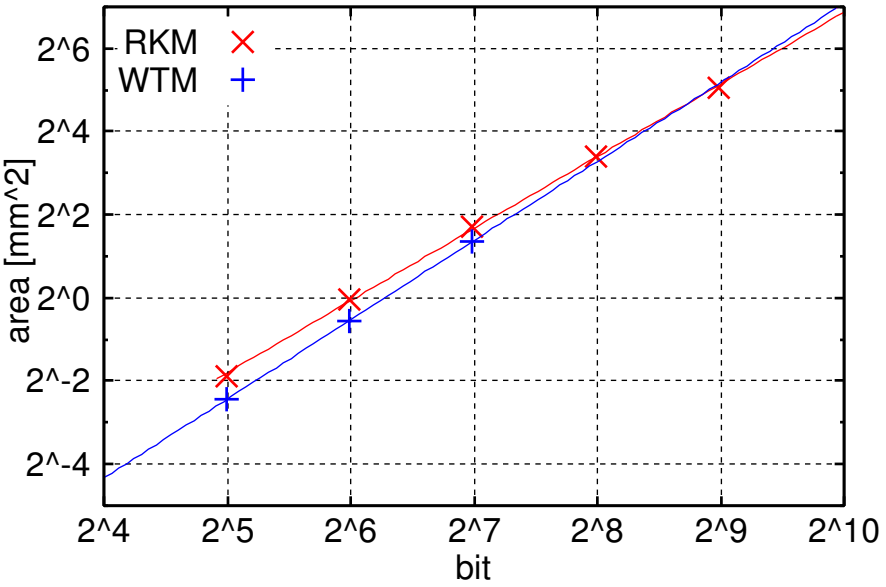

図 4.10　RKM の最大遅延時間と面積

まず，計算による値と実際の測定値を比較する．表 4.6 より，2^6 ビット RKM の最大遅延時間と面積の計算値はそれぞれ 13.37ns と 0.972mm^2 であることがわかる．一方，2^6 ビット RKM を Verilog-HDL で記述し，論理合成結果から最大遅延時間と面積を測定した結果，それぞれ 14.14ns と 0.864 mm^2 であった．この実測値と計算値の違いは，実測値が自動合成結果であることを考慮すると，許容範囲内である．このことから，本節で示した遅延時間と面積に関する式は実用的な範囲で面積と遅延時間を見積もることができると言える．

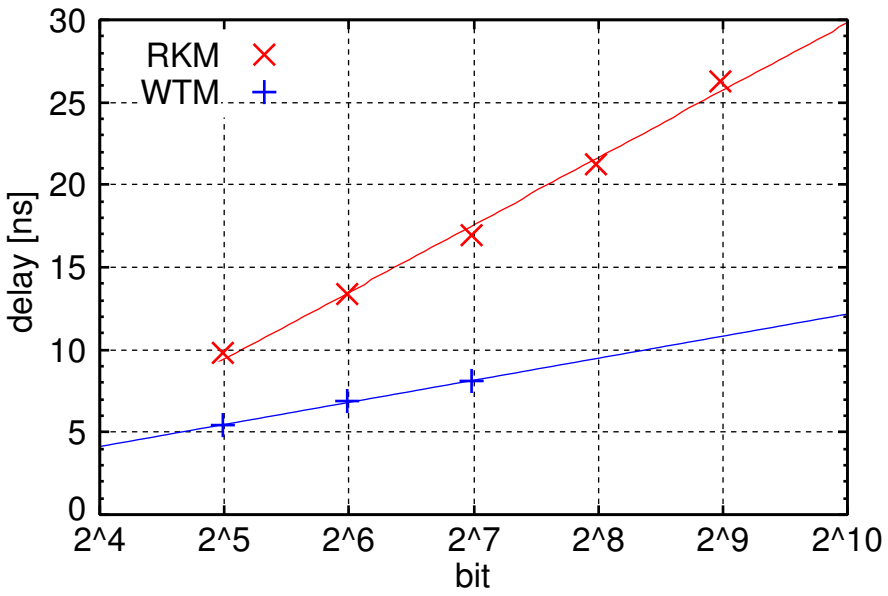

図 4.11　WTM の最大遅延時間と面積

次に，RKM と WTM の面積と遅延時間を比較し，考察する．図 4.10 に示した RKM の計算値を直線で近似した結果，傾きは Karatsuba アルゴリズムの理論値である $O(n^{1.58})$ と異なる 1.73 であった．これは，乗算以外の CPA や CSA による面積増加が原因であると考える．確認のため，式 (4.13) から式 (4.17) において，CPA や CSA の面積を 0 として同様に見積りの計算を行ったところ，傾きはほぼ 1.58 となった．WTM の測定値も同様に直線で近似したところ，傾きは約 1.90 であった．これは，理論値である $O(n^2)$ に近い．乗算ビット長の増加に対する WTM と RKM の増加に注目すると，約 2^9 ビット以下において WTM は RKM より面積が小さく，それ以上では RKM の面積は WTM より小さくなることがわかる．この両乗算器の面積の大小関係が逆転する点における面積は約 30mm^2 であった．遅延時間のオーダは，両乗算器とも $O(\log n)$ であるが，図 4.11 に示すように傾きは RKM の方が大きく，どのビット長においても WTM より大きい．

以上のことから，次のことが言える．速度重視の設計においては WTM を用いるべきである．面積重視の設計において，2^9 ビット以上では，RKM を用いる方が有利である．しかし，性能コスト比を考えた場合，RKM と WTM の面積が約 2^9 ビットの乗算で約 30mm^2 であることから，実用的な設計においては，WTM を用いる方が，性能コスト比に優れた設計ができると言える．これらの結果を踏まえると，Karatsuba 乗算を順序回路で実現する IKM の設計においては，基本乗算器として WTM を用いる方が性能コスト比の優れた IKM を実現することができると言える．

4.5 順序回路による Karatsuba 乗算器 (Iterative Karatsuba Multiplier, IKM)

本節では, 基本となる演算器を繰り返し用いる IKM を設計する．まず, Karatsuba アルゴリズムの適用回数を 2 回とした場合を例にして IKM の設計手法を示す．次に，ビット長とアルゴリズムの適用回数を変えて実装を行った結果を示す.

4.5.1 設計手法

本節ではガロア体上の Karatsuba 乗算器を設計する際のアプローチ [6] を参考に, 基本乗算器のビット長を n として，Karatsuba アルゴリズムを再帰的に 2 回適用した $4n$ ビット IKM を設計する．以降，n を基本ビット長と呼ぶ.

$4n$ ビットの乗数 A と被乗数 B を，次式で示すように n ビットずつ 4 個に分割する.

$$A = a_3 2^{3n} + a_2 2^{2n} + a_1 2^n + a_0 \tag{4.18}$$
$$B = b_3 2^{3n} + b_2 2^{2n} + b_1 2^n + b_0 \tag{4.19}$$

また，A と B の積 P を次のように表す.

$$P = p_7 2^{7n} + p_6 2^{6n} + p_5 2^{5n} + p_4 2^{4n} + p_3 2^{3n} + p_2 2^{2n} + p_1 2^n + p_0 \tag{4.20}$$

ここで,

$$A_1 = a_3 2^n + a_2 \tag{4.21}$$
$$A_0 = a_1 2^n + a_0 \tag{4.22}$$
$$B_1 = b_3 2^n + b_2 \tag{4.23}$$
$$B_0 = b_1 2^n + b_0 \tag{4.24}$$

とおくと,

$$A = A_1 2^{2n} + A_0 \tag{4.25}$$
$$B = B_1 2^{2n} + B_0 \tag{4.26}$$

となり，積 P は次式で与えられる.

$$P = AB \tag{4.27}$$
$$= A_1 B_1 2^{4n} + \big(A_{10} B_{10} - A_1 B_1 - A_0 B_0\big) 2^{2n} + A_0 B_0 \tag{4.28}$$

ここに,

$$A_{10} = A_1 + A_0 \tag{4.29}$$
$$B_{10} = B_1 + B_0 \tag{4.30}$$

と表現する．以降，このような 2 つの変数の加算を表現するため，それぞれの変数の添字を並べて書く．式 (4.28) に現れた項 A_1B_1，A_0B_0，$A_{10}B_{10}$ を展開すると，それぞれ以下のようになる．

$$\begin{aligned} A_1B_1 &= (a_3 2^n + a_2)(b_3 2^n + b_2) \\ &= a_3b_3 2^{2n} + (a_{32}b_{32} - a_3b_3 - a_2b_2)2^n + a_2b_2 \end{aligned} \tag{4.31}$$

$$\begin{aligned} A_0B_0 &= (a_1 2^n + a_0)(b_1 2^n + b_0) \\ &= a_1b_1 2^{2n} + (a_{10}b_{10} - a_1b_1 - a_0b_0)2^n + a_0b_0 \end{aligned} \tag{4.32}$$

$$\begin{aligned} A_{10}B_{10} &= (a_{31} 2^n + a_{20})(b_{31} 2^n + b_{20}) \\ &= a_{31}b_{31} 2^{2n} \\ &\quad + (a_{3120}b_{3120} - a_{31}b_{31} - a_{20}b_{20})2^n \\ &\quad + a_{20}b_{20} \end{aligned} \tag{4.33}$$

式 (4.31)，(4.32)，(4.33) を式 (4.28) に代入すると，最終的に以下の結果を得る．

$$\begin{aligned} P = {} & 2^{6n}a_3b_3 \\ & + 2^{5n}\big(a_{32}b_{32} - a_3b_3 - a_2b_2\big) \\ & + 2^{4n}\big(a_{31}b_{31} - a_3b_3 - a_1b_1 + a_2b_2\big) \\ & + 2^{3n}\big(a_{3120}b_{3120} - a_{31}b_{31} - a_{20}b_{20} - a_{32}b_{32} + a_3b_3 + a_2b_2\big) \\ & + 2^{2n}\big(a_{20}b_{20} - a_2b_2 - a_0b_0 + a_1b_1\big) \\ & + 2^{n}\big(a_{10}b_{10} - a_1b_1 - a_0b_0\big) \\ & + a_0b_0 \end{aligned} \tag{4.34}$$

式 (4.20) の係数 $p7$,...,$p0$ は，式 (4.34) に現れた係数から求めることができる．この様子を表 4.8 にまとめる．ただし，式 (4.34) の係数を構成する 9 個の項 a_0b_0，...，a_3b_3，$a_{10}b_{10}$，...，$a_{32}b_{32}$，$a_{3120}b_{3120}$ をそれぞれ $pp0$，...，$pp8$ と置き，これらを部分積と呼ぶ．部分積は n ビット以上の乗算の積であるため，$2n$ ビット以上の数である．したがって，下位 n ビット とそれ以上に分けて積算する必要がある．表中の L は各部分積の下位 n ビット，H はそれ以上の上位ビットを示す．また，積算中に桁上げが発生するため，全ての積算が終ったら，$p0$ から順に $p7$ まで桁上げの処理を行う必要がある．

Karatsuba アルゴリズムを 2 回以上再帰的に適用した場合でも，この導出で表 4.8 と似た表を得ることができる．表 4.9 に 1 回再帰の例，表 4.10 に 3 回再帰の例を示す．

表 4.8　式 (4.20) の係数と部分積の関係

Partial Products			p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0
$pp0$	a_0b_0	L					+	−	−	+
		H				+	−	−	+	
$pp1$	a_1b_1	L				−	+	+	−	
		H			−	+	+	−		
$pp2$	a_2b_2	L			−	+	+	−		
		H		−	+	+	−			
$pp3$	a_3b_3	L		+	−	−	+			
		H	+	−	−	+				
$pp4$	$a_{10}b_{10}$	L					−		+	
		H				−		+		
$pp5$	$a_{20}b_{20}$	L					−	+		
		H				−	+			
$pp6$	$a_{31}b_{31}$	L				+	−			
		H			+	−				
$pp7$	$a_{32}b_{32}$	L			+		−			
		H		+		−				
$pp8$	$a_{3120}b_{3120}$	L					+			
		H				+				

表 4.9　1 回再帰における IKM の演算

		p_3	p_2	p_1	p_0
a_0b_0	L			−	+
	H		−	+	
a_1b_1	L		+	−	
	H	+	−		
$a_{10}b_{10}$	L			+	
	H		+		

表 4.10 3 回再帰における IKM の演算

		p_{15}	p_{14}	p_{13}	p_{12}	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0
a_0b_0	L									−	+	+	−	+	−	−	+
	H								−	+	+	−	+	−	−	+	
a_1b_1	L								+	−	−	+	−	+	+	−	
	H							+	−	−	+	−	+	+	−		
a_2b_2	L							+	−	−	+	−	+	+	−		
	H						+	−	−	+	−	+	+	−			
a_3b_3	L						−	+	+	−	+	−	−	+			
	H					−	+	+	−	+	−	−	+				
a_4b_4	L					+	−	−	+	−	+	+	−				
	H				+	−	−	+	−	+	+	−					
a_5b_5	L				−	+	+	−	+	−	−	+					
	H			−	+	+	−	+	−	−	+						
a_6b_6	L			−	+	+	−	+	−	−	+						
	H		−	+	+	−	+	−	−	+							
a_7b_7	L		+	−	−	+	−	+	+	−							
	H	−	+	+	−	+	−	−	+								
$a_{10}b_{10}$	L									+		−		−		+	
	H								+		−		−		+		
$a_{20}b_{20}$	L									+	−			−	+		
	H								+	−			−	+			
$a_{31}b_{31}$	L								−	+			+	−			
	H							−	+			+	−				
$a_{32}b_{32}$	L							−		+		+		−			
	H						−		+		+		−				
$a_{40}b_{40}$	L									+	−	−	+				
	H								+	−	−	+					
$a_{51}b_{51}$	L								−	+	+	−					
	H							−	+	+	−						
$a_{54}b_{54}$	L					−		+		+		−					
	H				−		+		+		−						
$a_{62}b_{62}$	L							−	+	+	−						
	H						−	+	+	−							
$a_{64}b_{64}$	L					−	+			+	−						
	H				−	+			+	−							
$a_{73}b_{73}$	L						+	−	−	+							
	H					+	−	−	+								
$a_{75}b_{75}$	L				+	−			−	+							
	H			+	−			−	+								
$a_{76}b_{76}$	L			+		−		−		+							
	H		+		−		−		+								
$a_{3120}b_{3120}$	L									−				+			
	H								−				+				
$a_{5140}b_{5140}$	L									−		+					
	H								−		+						
$a_{6240}b_{6240}$	L									−	+						
	H								−	+							
$a_{7351}b_{7351}$	L								+	−							
	H							+	−								
$a_{7362}b_{7362}$	L							+		−							
	H						+		−								
$a_{7564}b_{7564}$	L					+				−							
	H				+				−								
$a_{73516240}\times$ $b_{73516240}$	L									+							
	H								+								

4.5.2　IKM の構成

前節で述べた手順で乗算を行う IKM の構成を図 4.12 に示す．IKM は，PPG

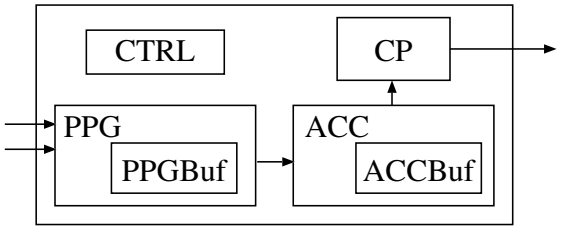

図 4.12　IKM の構成

(Partial Product Generator)，ACC(Accumulator)，CP (Carry Propagator) の 3 つの演算器と，コントロールモジュール (CTRL) で構成される．次節以降で，それぞれのモジュールについて詳細を述べる．

PPG の設計

PPG は，加算器，乗算器，バッファ (PPG Buffer) で構成され，部分積の生成を行う．構成を図 4.13 に示す．乗算器には，第 4.4.1 節での結果を踏まえ，RKM で

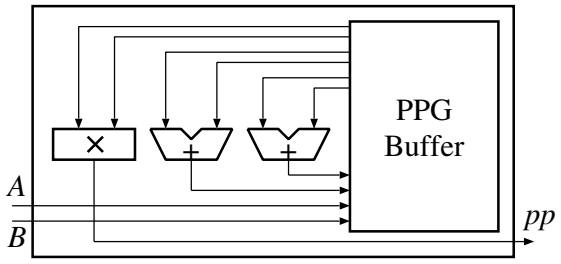

図 4.13　PPG の構成

はなく WTM を 1 個用いる．より高速な IKM を設計する場合には複数の乗算器を持つという構成もありえる．用いる WTM は，6 段にステージ分けされ，パイプライン化されたものを用いる．これは，遅延時間を IKM 内の加算器と同じ程度にするためである．この 6 段パイプラインの WTM には，Synopsys 社が提供するライブラリに含まれている乗算器 (DW02_mult_6_stage) を用いる．加算器のコストは乗算器に比べ非常に小さいので，本設計では 2 個用いる．Iterative Karatsuba 乗算は加算のあとに乗算を行うという演算が続くので，2 個の加算器を用いて，乗算器に入力する値を効率良く加算してゆくことで，コストの高い乗算器を持て余すことなく

動作させることができる．加算器にもまたライブラリとして提供されている加算器 (DW01_add) を利用した．

演算中のデータを保存する PPG Buffer には全ての演算器と入力端子に専用の I/O ポートを用意した．また，書き込みは同期的に，読み出しは非同期的に行うように設計した．PPG Buffer の容量は，データのビット長，エントリ数，演算器の個数に依存する．データのビット長については，基本ビット長に加え，部分積の演算過程で発生する桁上げを保存するだけの長さが必要である．部分積の演算中に発生する桁上げを保存するために必要なビット長は再帰の回数が 1 回増えるごとに 1 ビット増える．エントリ数は部分積の数に依存し，部分積の数もまた再帰の回数に依存する．再帰の回数が 1 回増えるごとに部分積の数は 3 倍になるため，エントリもそれに対応した数が必要になる．ただし，演算器の数を増やすとより効率のよい演算が可能になるため，再帰の回数が増えた場合でもエントリ数の増加を抑えることは可能である．

本実装では，基本ビット長が 16 で再帰の回数が 2 である場合，18 ビットのエントリが 10 個必要となる．

ACC の設計

ACC は，加減算器とバッファ (ACC Buffer) で構成され，PPG で生成された部分積を積算する．構成を図 4.14 に示す．用いる加減算器の数は，1 個の部分積に対

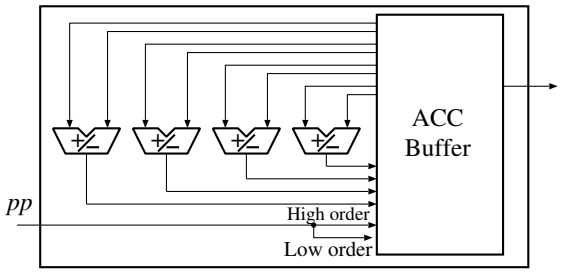

図 4.14　ACC の構成

して同時に行うことができる演算数を考慮して 4 とした．これは，表 4.8 において，1 行で行う最大の演算数に合わせたものである．設計によっては，より多くの加減算器を用いる構成も考えられる．加減算器は今までと同様にライブラリとして提供されている加減算器 (DW01_addsub) を用いた．

ACC Buffer の I/O ポートは PPG Buffer と同様に演算器および入力端子ごとに専用のものを用意した．ACC 内部では符号付き演算が行われるため，符号ビットが必要である．さらに，ACC 内部のビット長は加算による桁上げのために再帰の回数

が 1 回増えるごとに 2 ($= \lceil \log_2 3 \rceil$) ずつ増える．ACC Buffer の容量は部分積の数と演算器の個数に依存し，本実装では，基本ビット長が 16 で，再帰回数が 2 である場合，21 ビットのエントリが 18 個必要となる．

CP の設計

CP は，加算器とレジスタ (Carry Register) で構成され，ACC で積算された係数 (2 回再帰の場合は式 (4.20) の $p0$ から $p7$) を受け取り，順次桁上げを行うモジュールである．構成は図 4.15 のようになる．CP モジュールは，入力された値の下位 n

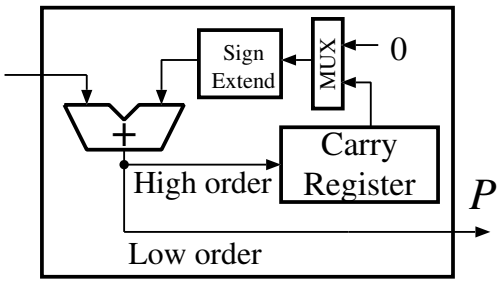

図 4.15 CP の構成

ビットとそれより上位のビットを分離して，上位のビットを Carry Register に保存し次のサイクルで入力される一つ上位の係数に加算する．ただし，最下位項 $p0$ に対する桁上げはないので，この場合は 0 を加算する．

CTRL の設計

CTRL モジュールは PPG，ACC，CP モジュールに対して，適切なタイミングで制御信号を送るモジュールである．代表的な制御信号の種類として，PPG Buffer や ACC Buffer に対する読み書きアドレスや Write Enable，ACC 内の加減算器に対する加減算選択信号，CP 内で最下位項に対して 0 を加算することを指示する信号がある．以下では，本実装における 2 回再帰の場合を例としてより具体的な制御の内容を述べる．

CTRL による制御のもとで，PPG は表 4.11 に示すようなスケジュールで演算を行う．表は，1 個の乗算器と 2 個の加算器について，入力のタイミングと出力結果が利用可能となるタイミングを示す．

ACC も同様に，CTRL モジュールによる制御のもとで表 4.12,4.13 に示すスケジュールで演算を行う． 表中の部分積に付けられた添字 L と H は，それぞれ，各部分積の下位 n ビットとそれ以外の上位ビットを意味する．CP に関しては，最下位項である $p0$ が入力されたときに加算すべき桁上げとして 0 を選択するマルチプ

表 4.11 R2IKM の PPG におけるスケジューリング

		Multiplier1		Adder1		Adder2	
Step	Input	Input	Output	Input	Output	Input	Output
1	a_0, b_0	−	−	−	−	−	−
2	a_1, b_1	a_0, b_0	−	−	−	−	−
3	a_2, b_2	a_1, b_1	−	a_1, a_0	−	b_1, b_0	−
4	a_3, b_3	a_2, b_2	−	a_2, a_0	a_{10}	b_2, b_0	b_{10}
5	−	a_3, b_3	−	a_3, a_1	a_{20}	b_3, b_1	b_{20}
6	−	a_{10}, b_{10}	−	a_3, a_2	a_{31}	b_3, b_2	b_{31}
7	−	a_{20}, b_{20}	−	a_{20}, a_{31}	a_{32}	b_{20}, b_{31}	b_{32}
8	−	a_{31}, b_{31}	$a_0b_0(= pp0)$	−	a_{2031}	−	b_{2031}
9	−	a_{32}, b_{32}	$a_1b_1(= pp1)$	−	−	−	−
10	−	a_{2031}, b_{2031}	$a_2b_2(= pp2)$	−	−	−	−
11	−	−	$a_3b_3(= pp3)$	−	−	−	−
12	−	−	$a_{10}b_{10}(= pp4)$	−	−	−	−
13	−	−	$a_{20}b_{20}(= pp5)$	−	−	−	−
14	−	−	$a_{31}b_{31}(= pp6)$	−	−	−	−
15	−	−	$a_{32}b_{32}(= pp7)$	−	−	−	−
16	−	−	$a_{2031}b_{2031}(= pp8)$	−	−	−	−

レクサを制御する.

ACC と CP の演算は，それぞれ先行する PPG と ACC での演算が完了する前に開始することができる．つまり，ACC は PPG が 8 ステップ目に到達した時点で最初の部分積 $pp0$ を受け取ることが可能なので，これに合わせて演算を開始する．また，ACC の演算において，最後に求まる係数の中で最も下位の係数は $p3$ である．この $p3$ は ACC の 17 ステップ目に出力されるため，$p0$ から $p2$ はこれに先行して順次 CP に入力することができる．したがって，積 P は乗数 A と 被乗数 B の最下位桁 a_0 と b_0 を入力した時点から，(8-1)+(17-1)+4=27 ステップ後に得られる．

表 4.12 R2IKM の ACC におけるスケジューリング (1/2)

		Adder-Subtracter1		Adder-Subtracter2	
Step	Input	Input	Output	Input	Output
1	pp0	-	-	-	-
2	pp1	p0,pp0L	-	p1,pp0L	-
3	pp2	p1,pp0H	p0+pp0L	p2,pp0H	p1-pp0L
4	pp3	p1,pp1L	p1+pp0H	p2,pp1L	p2-pp0H
5	pp4	p2,pp1H	p1-pp1L	p3,pp1H	p2+pp1L
6	pp5	p2,pp2L	p2-pp1H	p3,pp2L	p3+pp1H
7	pp6	p3,pp2H	p2-pp2L	p4,pp2H	p3+pp2L
8	pp7	p3,pp3L	p3-pp2H	p4,pp3L	p4+pp2H
9	pp8	p4,pp3H	p3+pp3L	p5,pp3H	p4-pp3L
10	-	p1,pp4L	p4+pp3H	p3,pp4L	p5-pp3H
11	-	p2,pp5L	p1+pp4L	p3,pp5L	p3-pp4L
12	-	p3,pp5H	p2+pp5L	p4,pp5H	p3-pp5L
13	-	p3,pp6L	p3+pp5H	p4,pp6L	p4-pp5H
14	-	p4,pp6H	p3-pp6L	p5,pp6H	p4+pp6L
15	-	p3,pp7L	p4-pp6H	p5,pp7L	p5+pp6H
16	-	p3,pp8L	p3-pp7L	p4,pp8H	p5+pp7L
17	-	-	p3+pp8L	-	p4+pp8H

表 4.13 R2IKM の ACC におけるスケジューリング (2/2)

		Adder-Subtracter3		Adder-Subtracter4	
Step	Input	Input	Output	Input	Output
1	$pp0$	–	–	–	–
2	$pp1$	$p2, pp0\mathrm{L}$	–	$p3, pp0\mathrm{L}$	–
3	$pp2$	$p3, pp0\mathrm{H}$	$p2 - pp0\mathrm{L}$	$p4, pp0\mathrm{H}$	$p3 + pp0\mathrm{L}$
4	$pp3$	$p3, pp1\mathrm{L}$	$p3 - pp0\mathrm{H}$	$p4, pp1\mathrm{L}$	$p4 + pp0\mathrm{H}$
5	$pp4$	$p4, pp1\mathrm{H}$	$p3 + pp1\mathrm{L}$	$p5, pp1\mathrm{H}$	$p4 - pp1\mathrm{L}$
6	$pp5$	$p4, pp2\mathrm{L}$	$p4 + pp1\mathrm{H}$	$p5, pp2\mathrm{L}$	$p5 - pp1\mathrm{H}$
7	$pp6$	$p5, pp2\mathrm{H}$	$p3 + pp2\mathrm{L}$	$p6, pp2\mathrm{H}$	$p4 - pp2\mathrm{H}$
8	$pp7$	$p5, pp3\mathrm{L}$	$p5 + pp2\mathrm{H}$	$p6, pp3\mathrm{L}$	$p6 - pp2\mathrm{H}$
9	$pp8$	$p6, pp3\mathrm{H}$	$p5 - pp3\mathrm{L}$	$p7, pp3\mathrm{H}$	$p6 + pp3\mathrm{L}$
10	–	$p2, pp4\mathrm{H}$	$p6 - pp3\mathrm{H}$	$p4, pp4\mathrm{H}$	$p7 + pp3\mathrm{H}$
11	–	–	$p2 + pp4\mathrm{H}$	–	$p4 - pp4\mathrm{H}$
12	–	–	–	–	–
13	–	–	–	–	–
14	–	–	–	–	–
15	–	$p4, pp7\mathrm{H}$	–	$p6, pp7\mathrm{H}$	–
16	–	–	$p4 - pp7\mathrm{H}$	–	$p6 + pp7\mathrm{H}$
17	–	–	–	–	–

4.5.3 実装結果と考察

再帰の回数を 1, 2, 3 にした IKM を設計し，それぞれ R1IKM, R2IKM, R3IKM とした．これらを Verilog-HDL で記述し，それぞれの IKM について基本ビット長を 4 から 128 まで変えて論理合成を行った．論理合成には第 4.4.1 節で述べた 0.18μm プロセスルールのライブラリを用いた．また，合成時の制約として遅延を最小にする条件を与えた．

論理合成の結果から，最大遅延時間, 面積, 乗算時間, 消費電力を求めた．結果を表 4.14 に示す．ただし，消費電力を求める時に用いた駆動電圧は 1.8V である．表の最

表 4.14 IKM の評価結果

Number of Recursions									
	Bit length	8	16	32	64	128	256	512	1024
1	Basic bit length	4	8	16	32	64	128	256	512
	Delay [ns]	2.00	2.20	2.39	2.82	2.94	3.26	-	-
	Area [mm^2]	0.13	0.19	0.39	0.92	1.98	5.35	-	-
	Time [ns]	28.00	30.80	33.46	39.48	41.16	45.64	-	-
	Power [mW]	16.28	26.39	56.27	161.2	398.2	1293	-	-
2	Basic bit length	2	4	8	16	32	64	128	256
	Delay [ns]	-	2.36	2.61	2.85	3.08	3.36	3.65	-
	Area [mm^2]	-	0.32	0.46	0.75	1.43	2.81	6.91	-
	Time [ns]	-	63.72	70.47	76.95	83.16	90.72	98.55	-
	Power [mW]	-	34.09	58.60	97.88	220.4	529.7	1530	-
3	Basic bit length	1	2	4	8	16	32	64	128
	Delay [ns]	-	-	3.11	3.15	3.30	3.59	3.94	4.48
	Area [mm^2]	-	-	0.87	1.18	1.77	2.91	5.23	10.9
	Time [ns]	-	-	245.7	248.9	260.1	283.6	311.3	353.9
	Power [mW]	-	-	85.40	118.6	179.6	329.6	705.7	1874

上段は乗数と被乗数のビット長 (Bit length) を表す．各 IKM について 1 行目は基本ビット長 (Basic bit length) を示す．4 行目は演算時間 (Time) を示し，それぞれの遅延時間 (Delay) にステップ数を乗じたものである．R1IKM, R2IKM, R3IKM それぞれに必要なステップ数はそれぞれ 14, 27, 79 であった．なお，R1IKM と R3IKM の ACC と PPG におけるスケジューリングは付録 D に示す．

さらにソフトウェアとの比較を行うため，exflib[16] を用いた Karatsuba 乗算の演算速度を測定した．exflib は Ver.20050709 を用いた．その他の条件は第 3.6.5 節

と同様に，Pentium 4 1.7 GHz，メモリ容量が 512MB，OS が FreeBSD 5.4，コンパイラが gcc 3.4.2 である．R1IKM，R2IKM，R3IKM と exflib の演算時間を比較したグラフを図 4.16 に示す．グラフの x 軸と y 軸はそれぞれ乗算ビット長と演算

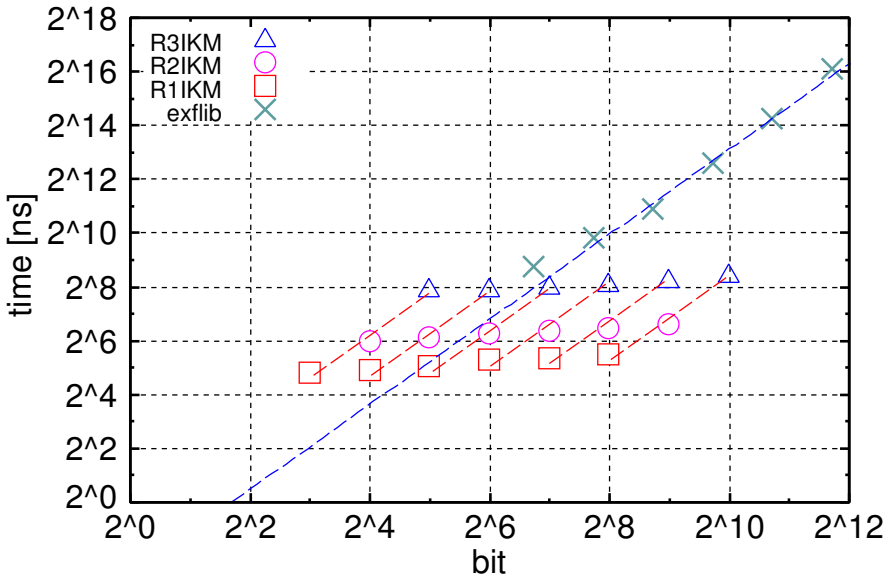

図 4.16　IKM とソフトウェア Karatsuba 乗算 (exflib) の速度比較

時間を対数で表している．また，図中の破線は傾き 1.58 の直線であり，Karatsuba アルゴリズムによる演算時間の理論値を示している．グラフから，exflib の測定値は理論値に近いことがわかる．また，R1IKM，R2IKM，R3IKM の測定値もまた理論値に近い．このことから，再帰の回数を増やした場合でも，ソフトウェアに対する性能比はほぼ一定であることがわかる．ただし，再帰の回数を増やすと，PPG，ACC 内部のバッファの容量が増えるため面積は増加する．また，図から，基本ビット長が大きいほど高い性能が得られ，基本ビット長を 32，64，128 にした IKM は，ソフトウェアと比較してそれぞれ約 5，10，30 倍の性能を持つことがわかる．この時，表 4.14 の中で最も大きい面積は，基本ビット長を 128 にした R3IKM の 10.9mm^2 であり，十分に実現可能な大きさである．

表 4.14 から，基本ビット長を 128 とした 1024 ビット R3IKM の消費エネルギーは，663nJ(=1874mW×353.9ns) であることがわかる．一方，ソフトウェアの場合，Pentium 4 1.7GHz の消費電力は約 63W であり [48]，856 ビット (10 進 256 桁) の乗算に要する計算時間が約 6366ns であった．したがって，消費エネルギーは 401μJ(=63W×6366ns) となり，ハードウェアの約 600 倍であることがわかった．

次に，R1IKM，R2IKM，R3IKM について，基本ビット長を変化させた場合の面積の変化をグラフにしたものを図 4.17 に示す．グラフの x 軸は乗算ビット長を対

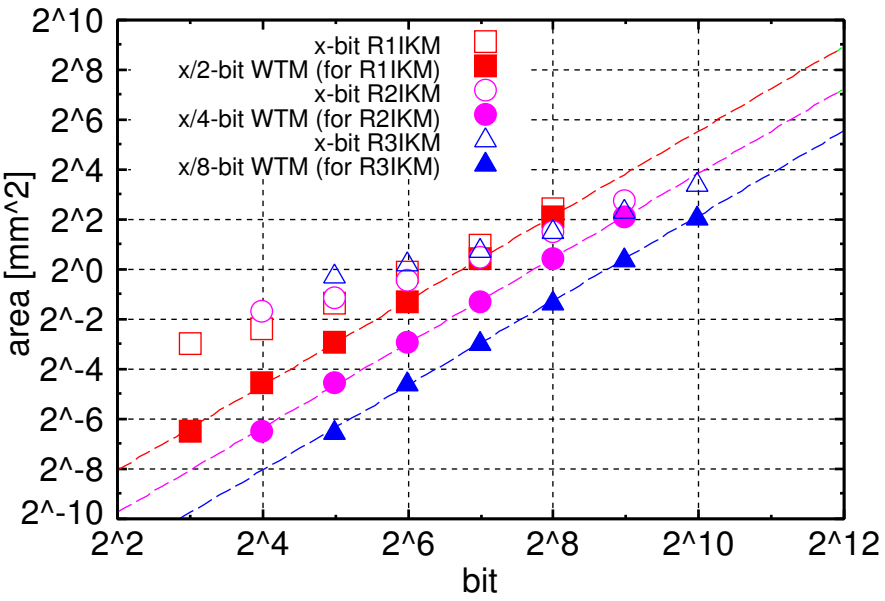

図 4.17　乗算ビット長 x に対する各 IKM の面積変化

数で，y 軸は面積を対数で表している．図中の□，○，△はそれぞれ，基本乗算器のビット長を 4 から 128 にしたときの R1IKM，R2IKM，R3IKM の面積を表している．また，■，●，▲はそれぞれ，R1IKM，R2IKM，R3IKM に使われている WTM の面積を表している．したがって，横軸 x に対してそれぞれビット長 $x/2$，$x/4$，$x/8$ における WTM の面積を意味している．

図 4.17 からわかるように，各 IKM の面積は，ビット長が大きくなるにしたがって，それぞれ内部で用いられている WTM の面積に近づいている．この理由について考察する．IKM の面積の内訳は主に WTM，PPG Buffer，ACC Buffer，加算器である．これらについて個別に考察する．まず，各バッファのエントリ数は再帰の回数のみに依存する．よって，再帰の回数を変えずに基本ビット長を増やした場合，その数は変わらない．ただし，エントリ長は増加する．よって，PPG Buffer と ACC Buffer の面積は $O(n)$ である．加算器の面積については明らかに $O(n)$ である．各 IKM の内部で用いられている WTM の面積については図から $O(n^{1.70})$ であることがわかる．本来の WTM における面積の理論値は $O(n^2)$ である．この差は，合成時に速度優先の制約条件を与えたためであると考える．実際に面積優先で合成した結果，面積は $O(n^{1.90})$ であり．理論値に近いものとなった．よって，これらの構成要素のなかで最も高いオーダを持つのが WTM であり，乗算桁数が大きくなったときに WTM の面積が支配的になるため，図のように WTM と，その WTM を用いた IKM の面積の差は小さくなる．また，基本乗算器のビット長を変えずに再帰の回数を増やした場合の面積増加については図から $O(n)$ であることがわかる．

4.6　本章のまとめ

本章では，Karatsuba 乗算アルゴリズムを組合せ回路で実現する RKM と，順序回路で実現する IKM の性能とコストを評価した．以下にその内容をまとめる．

4.6.1　32 ビット RKM の実装と評価

Karatsuba 乗算を組み合わせ回路で実現する RKM の実装と評価を行った．ここで，Karatsuba アルゴリズムを 1 回適用した 32 ビット RKM を実装し，最大遅延時間と面積を評価した．実装は，途中加算をすべて CSA で行う Carry-Save RKM と CPA で行う Binary RKM の 2 通り行い，両者の最大遅延時間と面積を比較した．その結果，Carry-Save RKM の 最大遅延時間は Binary RKM よりも 22% 短いことがわかった．ただし，面積は Binary RKM より 6% 大きかった．この結果から，Carry-Save RKM のコストパフォーマンスは Binary RKM よりも良いことがわかった．

4.6.2　一般形 RKM の実装と評価

より大きなビット長の乗算が可能な RKM を設計するため，基本ビット長や再帰の回数を固定しない，より一般的な形の RKM を設計し評価した．設計した RKM と WTM の最大遅延時間と面積を比較した結果，2^9 ビット以上において，RKM の面積コストは一般的な WTM より小さくなることがわかった．この時の面積は約 30mm^2 であった．遅延に関してはどのビット長においても RKM の方が WTM より大きかった．このことから，性能コスト比を考えると，WTM の方が RKM より優れていることがわかった．

4.6.3　IKM の実装と評価

Karatsuba 乗算を順序回路で実現する IKM の設計と評価を行った．その結果，基本ビット長を 32，64，128 ビットにした IKM は，ソフトウェアと比較してそれぞれ約 5，10，30 倍の性能を実現できることを示した．この時，最も大きい面積は，基本ビット長を 128 ビットにし，Karatsuba アルゴリズムを 3 回 再帰的に適用した R3IKM の 10.9mm^2 であり，十分に利用可能な大きさであった．消費エネルギーについては，基本ビット長を 128 とした R3IKM の消費エネルギーは汎用プロセッサと比較して 1/600 であることがわかった．

第 5 章

考察

本章では，第 3，4 章で述べた FFT 乗算器と Karatsuba 乗算器の評価結果をもとに，全体的な視点から多倍長乗算におけるハードウェア実装およびハードウェアとソフトウェアの協調設計について考察する．

5.1 ハードウェアによる多倍長乗算

本節では多倍長乗算のハードウェア実装について，全体的な視点からみた FFT 乗算器と Karatsuba 乗算器の使い分けについて考察する．

まず，比較的大きな桁の乗算を行う場合について考える．図 5.1 は 第 3，4 章で示した評価結果をグラフにまとめたものである．グラフには，ソフトウェア実装の FFT 乗算 (FFTW) と Karatsuba 乗算 (exflib)，およびハードウェア実装の FFT 乗算器と Iterative Karatsuba 乗算器 (IKM) の各性能が示されている．また，これらの性能を外挿したものが破線で示されている．ただし，IKM については 基本乗算器のビット長を 128 にした場合のみを示した．図からソフトウェア実装における FFT 乗算と IKM の性能は，2 進 2^{23} 桁 (16 進数 2^{21}) 付近で逆転することがわかる．また，ハードウェア実装においても，2 進数 2^{23} 桁付近で逆転している．このことから，FFT 乗算のソフトウェア実装とハードウェア実装，Karatsuba 乗算のソフトウェア実装とハードウェア実装，それぞれの性能比はほぼ同じであることがわかった．具体的には図から約 30 倍であることがわかる．

ここで 2 進 2^{23} 桁 における面積について考える．まず IKM に関して， この桁における IKM の面積を見積もるため，基本乗算器のビット長を 128 とし，再帰の回数を 1，2，3 と増加させた時の面積の変化を図 5.2 に示す．比較のため，2 進数 2^{15} 桁と 2^{23} 桁における FFT 乗算器の面積も図に示した．図から，IKM の面積は再帰の回数に比例して大きくなっていく様子がわかる．このとき増加する面積の多くは

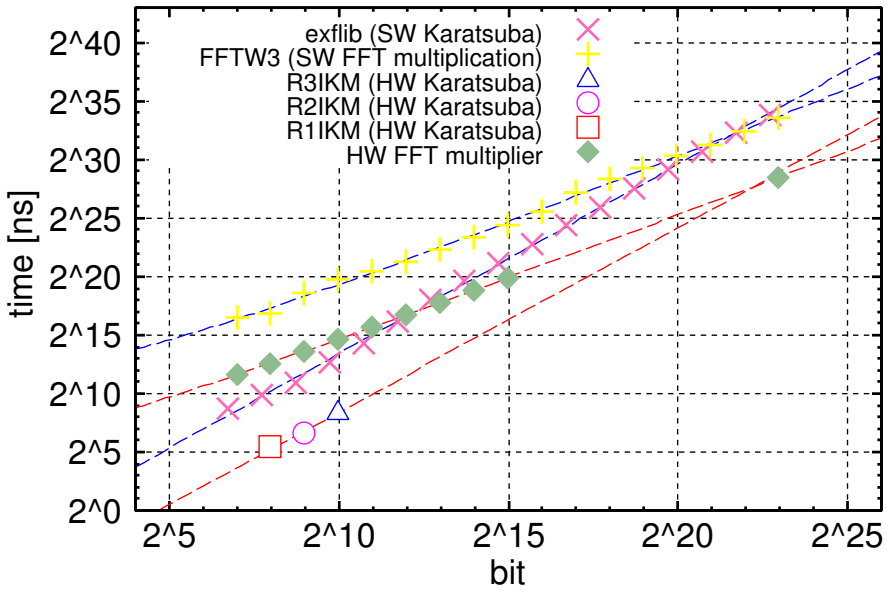

図 5.1　ソフトウェア／ハードウェアによる FFT 乗算器と Karatsuba 乗算器の性能比較

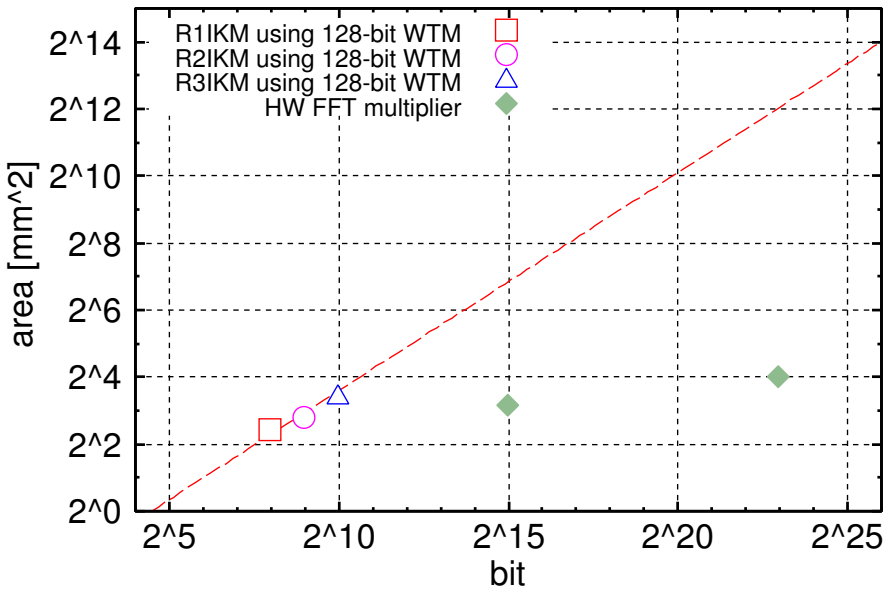

図 5.2　128 ビットの基本乗算器を用いた IKM における面積の外挿

バッファである．グラフから，2 進数 2^{23} 桁における IKM の面積を外挿したところ $2^{12}\mathrm{mm}^2$ と見積もることができる．一方，このときの FFT 乗算器の面積は $16\mathrm{mm}^2$ であった．したがって，この桁の乗算を行う場合は，IKM も FFT 乗算器と同様に内部のメモリをキャッシュとし，外部メモリを使うことを考えなくてはならない．

次に数百から数千ビットの比較的小さな多倍長乗算を考える．図 5.1 から，この時

の FFT 乗算器の性能はソフトウェア実装の Karatsuba 乗算器よりも劣る．逆に，IKM はソフトウェアも含めたあらゆる実装の中で最も高性能である．また，面積についても図 5.2 が示すように実装可能である．このことから，この付近の桁数では IKM が有効であることがわかった．

最後にこれらの中間に位置する数万ビットの乗算を考える．速度に関しては FFT 乗算器と Karatsuba 乗算器どちらの性能もソフトウェアより高い．したがって，この桁においては Karatsuba 乗算器も FFT 乗算器も共に有効であると言える．ただし，両実装とも外部メモリが必要である．

以上の考察を性能に関して表 5.1 にまとめる．なお，計算資源として，ソフトウエア実装には汎用プロセッサが，ハードウェア実装には面積コスト 4〜16mm^2 程度の演算器が必要である．

表 5.1 広い桁範囲にわたる乗算の性能．良い方から順に◎，○，△，×．

	ソフトウェア		ハードウェア	
	FFT	Karatsuba	FFT	Karatsuba
千ビット以下	×	○	△	◎
数千から数万	×	○	○	◎
数十万ビット程度	×	△	○	◎
数百万ビット程度	△	×	◎	○

5.2 協調設計による多倍長乗算

前節で行った考察は，ソフトウェア，またはハードウェア実装のみを考慮している．本論文の冒頭でも述べたように，ハードウェアを設計，評価する動機としてソフトウェアとの協調設計がある．そこで本節では，本研究で設計したハードウェア多倍長乗算器を用いたハードウェア／ソフトウェア協調設計について考察する．

多倍長乗算におけるハードウェアとソフトウェアの協調設計はあらゆる桁数において有効である．しかし，一般的に，特に協調設計が有効となるのは，ソフトウェア実装では性能が不足するような場合かハードウェア実装ではコストがかかりすぎるような場合である．前節の考察から多倍長乗算においてこのような状態になり得るのは，数万桁以上の乗算である．システム設計に許容されたバジェットによっては，このような桁数の乗算をハードウェア FFT 乗算器単体で行う選択肢もある．しかし，数万桁程度では，千ビット程度のハードウェア多倍長乗算器と汎用プロセッサを用いて協調設計を行うことで，より良い性能コスト比が得られると考えられる．

数千万の桁数以上においては，そもそもハードウェア単体による実装は現実的ではないので協調設計を行う必要がある．また，数百から数千ビットにおいても，バジェットが少ない場合には，協調設計を採用することは有効である．

第 6 章

おわりに

6.1 結論

本研究では，様々なアルゴリズムに基づく多倍長乗算を VLSI で実装し評価を行った．

多倍長乗算は，古くは高精度の数値計算などで用いられ，近年では素数判定，カオス計算，暗号計算など，様々な分野で利用されている．多倍長乗算はこのような利用例を考えると，今後その高速性がより求められるようになると考えられる．高速化へのアプローチとしてはアルゴリズムの改良やハードウェアによる実現が考えられる．アルゴリズムの改良は過去に多く行われ，代表的なものとして FFT 法や Karatsuba 法が広く知られている．しかし，多倍長乗算をハードウェアで実現し，その性能を評価した研究は少ない．近年のシステム設計においては，ソフトウェアとハードウェアを必要に応じて組み合わせた協調設計が多く行われているが，このような設計を行う際にも，ソフトウェアとハードウェアの両方に関する詳細な検討が必要である．このような背景を踏まえ，本論文では，多倍長乗算のハードウェア化による高速化に焦点をあて，FFT 法を用いた比較的大きな桁数の多倍長乗算から Karatsuba 法を用いた比較的小さな桁数の多倍長乗算について，それぞれをハードウェアで実装し，ソフトウェア実装との比較を行うことでそのコストや性能を明らかにした．

FFT 法を用いた FFT 乗算器の設計においては，演算器内部のデータ表現に用いる浮動小数点表現のビット長を実験的な誤差解析に基づいて最適な長さに決定することで性能とコストを最適化した乗算器を設計した．この最適なビット長は，最大値どうしの乗算が最大の誤差を与えるという点に着目し，最大値どうしの乗算が正しく行われるような長さとして求めた．この最適なデータ表現を用いて 2^{13} 桁の乗算を行う FFT 乗算器を実装したところ，最大遅延時間と面積はそれぞれ 7.63ns と 6.55mm^2 であった．一方，標準的な IEEE754 の 64 ビット浮動小数点表現と同じ

ビット長で実装した場合の最大遅延時間と面積はそれぞれ 10.3ns と 16.1mm^2 であった．このことから，誤差に着目して最適な実装を行うことで，遅延時間を 26%，面積を 60 % 削減した FFT 乗算器を実装することができた．

次に，FFT 乗算器に対して最適なパイプライン化を行った．パイプライン化後の FFT 乗算器の 1 回の乗算時間は 1.02ms であった．一方，パイプライン化前は 3.38ms であった．これにより，最適なパイプライン化によって，3.3 倍の性能を持つ FFT 乗算器を実装することができた．なお，パイプライン化後の FFT 乗算器の面積は 9.05 mm^2 であり，これは一般的な汎用プロセッサの 5% 程度であった．

このようにして作成したハードウェア FFT 乗算器 と，ソフトウェアによる FFT 乗算との速度比較を行った．ソフトウェアの実行には，ハードウェア FFT 乗算器と同じテクノロジで実装された Pentium 4 1.7GHz と FFT ライブラリとして高速な FFTW を用いた．比較の結果，ハードウェア FFT 乗算器の速度は，ソフトウェアと比較して 16 進数 2^5 桁から 2^{13} 桁の乗算において，19.7 倍から 34.3 倍，平均すると 25.7 倍高速であることがわかった．

次に，FFT 乗算が Karatsuba 乗算とほぼ同じ速度になる 16 進数 2^{21} 桁 (2 進数 2^{23} 桁) 以上の乗算において，ソフトウェア実装された Karatsuba 乗算 (exflib) と FFT 乗算器の速度を比較した．比較にあたり，16 進数 2^{21} 桁の乗算を行う最適な FFT 乗算器を実装した．比較の結果，ハードウェア FFT 乗算器の乗算時間が 0.39s，exflib による乗算時間が 16.5s であった．この結果から，ハードウェア FFT 乗算は，exflib と比較して 42 倍の性能を実現できることがわかった．また同じ桁数において FFTW による FFT 乗算の実行時間は 13.9s であった．よってソフトウェア実装された FFT 乗算との性能比は 35 倍となった．

16 進数 2 桁版 16 ビット FFT 乗算器をチップに実装し，試作を行った．その結果，2.8mm 角のチップサイズで実装可能であることが確認できた．この試作により，2^{21} 桁の乗算を行う FFT 乗算器でも 9mm 角かそれ以下のチップサイズで実現できることがわかった．

Karatsuba 法を用いた多倍長乗算器の設計においては，乗算器の構成に必要な内部演算器をすべて用意し，組み合わせ回路で実現する RKM と，固定された個数の内部演算器を繰り返し用いて順序回路で実現する IKM の 2 種類を設計した．

RKM の設計においては，まず，Karatsuba アルゴリズムを 1 回適用した 32 ビット RKM を実装し，最大遅延時間と面積を評価した．実装は，途中加算をすべて CSA で行う Carry-Save RKM と CPA で行う Binary RKM の 2 通り行い，両者の最大遅延時間と面積を比較した．その結果，Carry-Save RKM の 最大遅延時間は Binary RKM よりも 22% 短いことがわかった．ただし，面積は Binary RKM より 6% 大きかった．この結果から，Carry-Save RKM の性能コスト比は Binary RKM

よりも良いことがわかった．

次に，より大きなビット長の乗算が可能な RKM を設計するため，基本ビット長や再帰の回数を固定しない，より一般的な形の RKM を設計し評価した．設計した RKM と WTM の最大遅延時間と面積を比較した結果，2^9 ビット以上において，RKM の面積コストは一般的な WTM より小さくなることがわかった．この時の面積は約 30mm^2 であった．遅延に関してはどのビット長においても RKM の方が WTM より大きかった．このことから，性能コスト比を考えると，WTM の方が RKM より優れていることがわかった．

IKM の設計においては，基本ビット長を 32，64，128 ビットにした IKM は，ソフトウェアと比較してそれぞれ約 5，10，30 倍の性能を実現できることを示した．この時，最も大きい面積は，基本ビット長を 128 ビットにし，Karatsuba アルゴリズムを 3 回 再帰的に適用した R3IKM の 10.9mm^2 であり，十分に実装可能な大きさであった．消費エネルギーについては，基本ビット長を 128 とした R3IKM の消費エネルギーが汎用プロセッサと比較して 1/600 であることがわかった．

これらの結果を踏まえて全体的な視点からみた FFT 乗算器と Karatsuba 乗算器の使い分けについて考察した．まず，ハードウェアとソフトウェアいずれにおいても，FFT 乗算と Karatsuba 乗算 (IKM) は 2 進 2^{23} 桁 (16 進数 2^{21}) 付近で性能が逆転することがわかった．2 進 2^{23} 桁においてハードウェア実装とソフトウェア実装の性能比はいずれのアルゴリズムについても約 30 倍であった．面積については 2 進数 2^{23} 桁における IKM の面積を外挿したところ 2^{12}mm^2 であった．また，このときの FFT 乗算器の面積は実装結果から 16mm^2 であった．このことから，この桁の乗算を IKM で行う場合には FFT 乗算同様に外部メモリを用いる必要があることがわかった．

数百から数千ビットの比較的小さな多倍長乗算においては性能コスト比を考えると IKM が有効であることがわかった．数万ビットの乗算においては FFT 乗算器，Karatsuba 乗算器共にソフトウェアより高速であるため，どちらも有効であると言える．

ハードウェア／ソフトウェア協調設計について，システム設計に許容されたバジェットが少ない場合や数万桁の乗算においては千ビット程度のハードウェア多倍長乗算器と汎用プロセッサを用いた協調設計を行うことで，より良い性能コスト比が得られると考えられる．また数百万から数千万の桁数においては，そもそもハードウェア単体による実装は現実的ではないので協調設計を行う必要がある．

これらの結果から，FFT 法と Karatsuba 法の両ハードウェア実装おいて，パラメータに応じた性能コスト比の変化と適用範囲が明らかになった．本論文は，広い桁範囲における多倍長乗算のハードウェア化に関する詳細な研究結果を述べた唯一の

ものであり，多倍長乗算を用いたアプリケーションやシステムの実現において有益な指標となる．また，多倍長演算に関する実装技術の研究や開発，およびアプリケーションシステムの利用促進に大きく寄与すると考えられる．

6.2　今後の展望

本論文で述べた多倍長乗算のハードウェア実装は最適な設計に基づくものであるが，Cooley Tukey 以外の FFT アルゴリズムを用いた場合や，浮動小数点表現以外の表現 (固定小数点，ブロック小数点など) を用いた場合に，遅延時間や面積がどのように変わるのかについては研究の余地がある．FFT は分散処理が可能なアルゴリズムであるため，演算器を複数用意して並列度を上げることによる高速化も考えられる．また，本研究では FFT 乗算に複素数の FFT を用いた．FFT 法はこの他にも FMT による整数 FFT で実現することができる．この FMT をハードウェア実装し，性能，面積コスト，消費電力を本研究における実装と比較することで，さらに多くの実装選択肢を示すことができると考える．Karatsuba 法については，基本となる乗算器や加算器の個数を変えることで，どの程度の性能コスト比が実現できるのかについても調査が必要である．

また，本研究では主にゲートレベルでの評価を行ったが，今後は実際にシステムに組み込み，システム設計の立場から回路の構成を検討する必要がある．そのためにも，設計したハードウェアを実際に動くかたちでチップ化する必要がある．特に，メモリアーキテクチャの詳細設計が必要であり，これは今後の課題である．

近年では FPGA なども大容量化が進み，専用ハードウェアを組み込んだシステムを低コストで構築することができるようになった．また，VLSI の微細化やパッケージングも含めた実装技術も日々進歩しているため，今後はより高性能なハードウェアを安価に利用できるようになり，システム設計におけるソフトウェアとハードウェアの協調が有効となる場面はより多くなると考える．本研究の結果がこのような発展の一助となれば幸いである．

謝辞

本研究を進めるにあたり，お世話になった多くの方々に感謝いたします．

長年にわたり多くの御指導をいただいた電気通信大学情報工学科の阿部公輝助教授に感謝いたします．また，主任指導教員として御尽力いただいた電気通信大学情報工学科の尾内理紀夫教授に感謝いたします．FFT 乗算アルゴリズムに関して，重要な議論をしていただいた電気通信大学情報工学科の野下浩平教授に感謝いたします．また，FFT 乗算器に関する研究において，応用面で有益なアドバイスをいただいた電気通信大学情報工学科の加古孝教授に感謝いたします．FFT 乗算における誤差の見積もりに関して，多忙な時間をさいて有意義な議論をしていただいた電気通信大学情報工学科の山本野人教授に感謝いたします．本論文の査読を通して，論文の質を高めるために御尽力いただいた電気通信大学情報工学科の小林聡助教授に感謝いたします．日々の研究生活や本論文の作成にあたり多くのアドバイスをいただいた電気通信大学情報工学科の楯岡孝道博士に感謝いたします．FFT の設計に関して多くのアドバイスをいただいた NEC システムデバイス研究所の鈴木紀章博士に感謝いたします．卒業後も筆者の研究生活をさまざまな面で支えていただいた東京工科大学コンピュータサイエンス学部の松永俊雄教授に感謝いたします．本研究を始めるきっかけを与えていただき，その後も多倍長演算に関して様々な議論をしていただいた東京工科大学コンピュータサイエンス学部の月江伸弘博士に感謝いたします．最後に研究生活を支えていただいた家族と電気通信大学阿部公輝研究室の皆様に感謝いたします．

参考文献

[1] H. Fujiwara, “High-Accurate Numerical Method for Integral Equations of the First Kind under Multiple-Precision Arithmetic,” *Theoretical and Applied Mechanics Japan*, Vol.52, pp.193-203, 2003.

[2] M. Agrawal, N. Kayal, and N. Saxena, *PRIMES Is in P*, http://www.cse.iitk.ac.in/, 2002.

[3] 夛田光輝, 上島直樹, 茶谷芳裕, 鎌田弘之, “カオスシステムにおける有限桁演算の影響度の評価について,” 第 *17* 回 回路とシステム軽井沢ワークショップ論文集, pp.13-18, 2004.

[4] 月江伸弘, 小澤智, “カオス方程式の数値解における有効桁数の減少,” 情報処理学会論文誌, Vol.44, No.11, pp.2778-2786, 2003.

[5] J. C. Sprott, “Chaos and time-series analysis,” *Oxford University Press*, 2003.

[6] 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, May 2005.

[7] B. W.-K. Ling, C. Y.-F. Ho and P. K.-S. Tam, “Chaotic filter bank for computer cryptography,” *Chaos, Solitons & Fractals, In Press*, Available online 5 Jun. 2006.

[8] T.-I Chien and T.-L. Liao, “Design of secure digital communication systems using chaotic modulation, cryptography and chaotic synchronization,” *Chaos, Solitons and Fractals*, Vol.24, pp.241-255, 2005.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 2, 2nd Edition : Seminumerical Algorithms*, Addison-Wesley, MA, 1981.

[10] A. Karatsuba and Y. Ofman, “Multiplication of multidigit numbers on automata,” *Sov. Phys. Dokl.*, Vol.7, pp.595-596, 1963.

[11] A. L. Toom, “The complexity of a scheme of functional elements realizing the multiplication of integers,” *Sov. Math.*, Vol.3, pp.714-716, 1963.

[12] S. A. Cook, “On the minimum computation time of functions,” Ph.D thesis of Harvard University, Chap.III, May 1966, pp.51-77.

[13] A. Schönhage, V. Strassen, *Computing 1*, pp.182-196, 1966.

[14] A. Schönhage, V. Strassen, “Schnelle multiplikation grosser zahlen,” Computing 7, pp.282-289, 1971.

[15] Dan Zuras, “More on Squaring and Multiplying Large Integer,” *IEEE Trans. Computers*, Vol.43, No 8, pp.899-908, Aug. 1994.

[16] exflib - Extended Precision Float-Point Arithmetic Library, http://www-an.acs.i.kyoto-u.ac.jp/ fujiwara/exflib/exflib-index.html

[17] GMP, http://www.swox.com/gmp/

[18] FFTW, http://www.fftw.org/

[19] C. S. Wallace, “A Suggestion for a Fast Multiplier,” *IEEE Trans. Electronic Computers*, Vol.EC-13, No.2, pp.14-17, Feb, 1964.

[20] C. Grabbe, M. Bednara, J. Teich, J. von zur Gathen, and J. Shokrollahi, “FPGA designs of parallel high performance GF(2^{233}) multiplier,” *Proc. of the IEEE International Symposium on Circuits and Systems*, pp.362-369, May 2003.

[21] L. S. Cheng, A. Miri, and T. H. Yeap, “Improved FPGA Implementations of Parallel Karatsuba Multiplication over $\mathbb{GF}(2^n)$,” *Proc. of the 22nd Biennial Symposium on Communications*, Kingston, Canada, Jun. 2004.

[22] A. Weimerskirch and C. Paar, “Generalizations of the Karatsuba Algorithm for Efficient Implementaions,” Cryptology ePrint Archive, http://eprint.iarc.org/, Report 2006/224, 2006.

[23] 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並列乗算,” 電子情報通信学会 2005 年総合大会講演論文集, Vol.A-3, Mar. p.66, 2005.

[24] Verilog DOT COM, http://www.verilog.com

[25] Synopsys, http://www.synopsys.com

[26] VLSI Design and Education Center Homepage, http://www.vdec.u-tokyo.ac.jp

[27] Cadence, http://www.cadence.com

[28] 後保範, “多数桁分割乗算の高速計算法,” 情報処理学会論文誌, Vol.46, No.5, pp.1266-1273, May 2005.

[29] J. W. Cooley and J. W. Tukey, “An Algorithm for Machine Computation of

the Complex Fourier Series," *Mathematics of Computation*, Vol.19, pp.297-301, Apr. 1965.

[30] 安居院 猛, 中嶋 正之, "FFT の使い方", 秋葉出版, 1986.

[31] C. M. Fiduccia, "Fast Matrix Multiplication," *Proceedings of Third Annual ACM Symposium on Theory of Computing*, pp.45-49, 1971.

[32] M. Frigo, "A Fast Fourier Transform Compiler," *ACM SIGPLAN Notices*, Vol. 39, Issue 4, pp.642-655, Apr. 2004.

[33] J. van der Hoeven, "The Truncated Fourier Transform and Applications," *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, pp.290-296, 2004.

[34] B. M. Baas, "A Low-Power, High-Performance 1024-Point FFT Processor," *IEEE Journal of Solid-State Circuits*, Vol.34, No.3, pp.380-387, Mar. 1999.

[35] N. Miyamoto, L. Karnan, K Maruo, K. Kotani, and T. Ohmi, "A Small-Area High-Performance 512-Point 2-Dimensional FFT Single-Chip Processor," *Proceedings of the 2004 Conference on Asia South Pacific Design Automation : Electronic Design and Solution Fair 2004*, pp.537-538, Jan. 2004.

[36] ALTERA, http://www.altera.com.

[37] "*ALTERA White Paper : Floation-Point FFT Processor (IEEE754 Single Precision) Radix 2 Core*,"
http://www.altera.co.jp/literature/wp/wp_fft_radix2.pdf.

[38] C. J. Weinstein, "Roundoff Noise in Floating Point Fast Fourier Transform computation ," *IEEE Transactions on Audio and Electroacoustics*, Vol.AU-17, No.3, Sep. 1969.

[39] Tran-Thong and D. Liu, "Accumulation of Roundoff Errors in Floating Point FFT," *IEEE Transactions on Circuits and Systems*, Vol.CAS-24, No.3, Mar. 1877.

[40] I. Pitas and M. G. Strintzis, "Floating Point Error Analysis of Two-Dimensional Fast Fourier Transform Algorithms," *IEEE Transactions on Circuits and Systems*, Vol.35, No.1, Jan. 1988.

[41] D. C. Munson and B. Liu, "Floating Point Roundoff Error in the Prime Factor FFT," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol.ASSP-29, No.4, Aug. 1981.

[42] Y. Ma, "An Accurate Error Analysis Model for Fast Fourier Transform," *IEEE Transactions on Signal Processing*, Vol.45, No.6, Jun. 1997.

[43] P. Henrici, *Applied and Computational Complex Analysis*, Vol.3, Chap.13,

John Wiley & Sons, NY, 1986.

[44] 平山 弘, "FFT による高精度数の乗算," 情報処理学会 研究報告 *High Performance Computing*, Vol.65, No.65-6, pp.27-32, 1997.

[45] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd Edition*, Morgan Kaufmann, CA, 2003.

[46] Intel Co., http://www.intel.com

[47] benchFFT, http://www.fftw.org/benchfft/

[48] Intel(R) Pentium(R) 4 processor specifications, http://www.intel.co.jp/products/processor/pentium4/specs.htm

付録 A

加算回路

本章では基本的な算術演算回路である加算回路について述べる．ハードウェアで扱う加算回路は大きく桁上げ伝搬加算器と桁上げ保存加算器に分けられる．以降，それぞれについて述べる．

A.1 半加算器と全加算器

加算器は 1 ビットの加算を行う基本回路，半加算器 (Half Adder，HA) と全加算器 (Full Adder，FA) で構成される．半加算器は桁上げを考慮しない加算器であり，その論理は次式で表される．

$$s = x \oplus y \tag{A.1}$$
$$c = x \cdot y \tag{A.2}$$

ここに，x と y は加算する 2 値を，s，c はそれぞれ和と桁上げを表す．また，演算子 $\oplus$ と $\cdot$ はそれぞれ排他論理和と論理積を意味する．一方，全加算器は下位桁からの桁上げを考慮した加算を行い．その論理は次式で表される．

$$s_i = x_i \oplus y_i \oplus c_i \tag{A.3}$$
$$c_{i+1} = x_i \cdot y_i + y_i \cdot c_i + c_i \cdot x_i \tag{A.4}$$

添字 i は n ビットの値における i ビット目を意味する．特に c_i は i ビット目に対する下位桁からの桁上げを意味する．

A.2 桁上げ伝搬加算器

桁上げ伝搬加算 (Carry Propagation Adder，CPA) は加算により発生した桁上げ (Carry) を下位桁から上位桁に伝搬しながら順次加算を行う加算器である．桁上

げが伝搬するため，ビット長が大きくなるとより大きな遅延が生じる．したがって，CPA の高速化の際には，この桁上げをいかに高速に行うかが問題となる．次に，いくつかの代表的な CPA を示す．

A.2.1 順次桁上げ加算器 (Ripple Carry Adder)

FA を単純に直列接続したものを順次桁上げ加算器 (Ripple Carry Adder，RCA) と呼ぶ．RCA の構成を図 A.1 に示す．RCA はその構造上 n ビットの加算を行うた

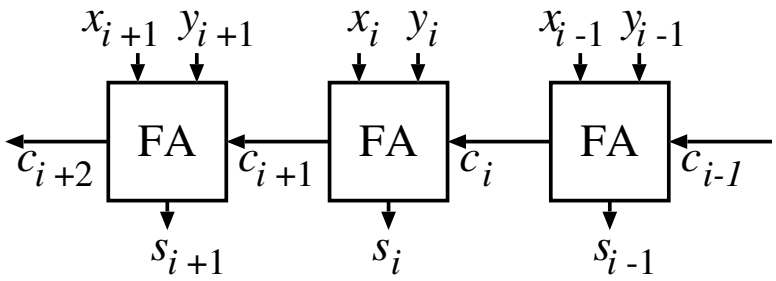

図 A.1 順次桁上げ加算器 (Ripple Carry Adder，RCA) の構成

めに $O(n)$ の遅延時間が必要であり，加算器の中では最も性能が低い．ただし，演算器の構造が単純であることからゲート数が少なく，低面積での実装が可能である．加えて FA の配列が規則的であるためマスク・レベルでの実装において配置配線が容易であるという利点がある．

A.2.2 桁上げ先見加算器 (Carry Look-ahead Adder)

桁上げ先見加算器は各桁における桁上げを並列に計算し，高速な桁上げを行うことで加算の遅延時間を削減した加算器である．今，n ビットの値 x と y の加算を考える．i 桁において桁上げが発生するのは $x_i \cdot y_i$ が 1 の場合である．また，下位桁から i 桁へ伝搬してきた桁上げ c_i がさらに c_{i+1} へ伝搬するのは，桁上げ c_i が 1 であり，かつ x_i と y_i のどちらかが 1 の場合である．したがって，上記の 2 つの場合のどちらかが成り立つとき，i 桁から $i+1$ 桁へ桁上げが発生する．これを論理式で表すと

$$c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot ci \tag{A.5}$$

$$= g_i + p_i \cdot c_i \tag{A.6}$$

となる．ここで，g_i は i 桁において発生する桁上げを意味することからジェネレータ (Generator) と呼ばれている．また，p_i は下位桁からの桁上げが上位へ伝搬することを意味していることから，プロパゲータ (Propagator) と呼ばれている．

式 (A.6) において，c_i を再帰的に代入していくと i 桁における桁上げを決定することができる．この様子を次式に示す．

$$
\begin{aligned}
c_{i+1} &= g_i + p_i \cdot c_i && \text{(A.7)}\\
&= g_i + p_i \cdot (g_{i-1} + p_{i-1} \cdot c_{i-1}) && \text{(A.8)}\\
&= g_i + p_i \cdot (g_{i-1} + p_{i-1} \cdot (g_{i-2} + p_{i-2} \cdot c_{i-2})) && \text{(A.9)}\\
&= g_i + p_i \cdot (g_{i-1} + p_{i-1} \cdot (g_{i-2} + p_{i-2} \cdot ((\ldots((g_0 + p_0 \cdot c_0))\ldots)) && \text{(A.10)}\\
&= g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot g_{i-2} + \cdots p_i \cdot p_{i-1} \cdots p_0 \cdot c_0 && \text{(A.11)}
\end{aligned}
$$

n 桁の全ての桁に対して式 (A.11) を用いることで，全ての桁の桁上げを並列に求めることができる．各桁の桁上げを並列に求める桁上げ先見回路 (Carry Look-ahead Circuit，CLC) を用いて図 A.2 に示すような桁上げ先見加算器 (Carry Look-ahead Adder，CLA) を構成することができる．ただし，各桁の加算は FA ではなく HA で行う．

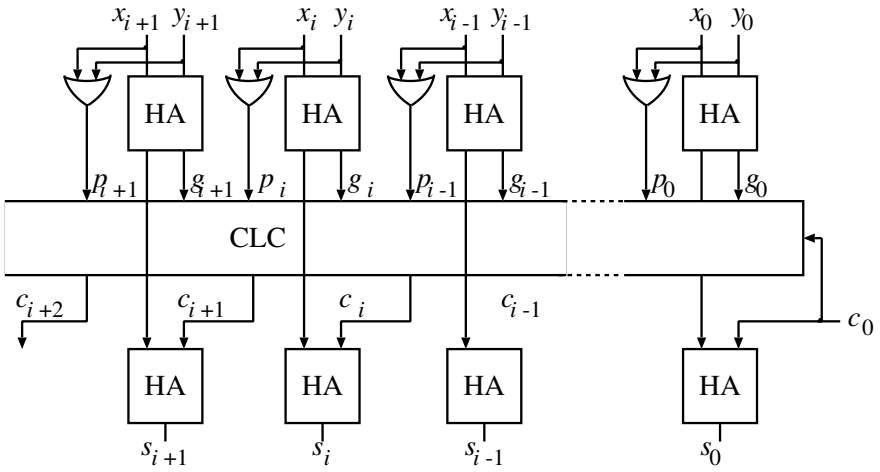

図 A.2　桁上げ先見加算器 (Carry Look-ahead Adder，CLA) の構成

A.2.3　**ブロック桁上げ先見加算器** (Block Carry Look-ahead Adder)

図 A.2 に示した CLA は上位桁になるほど桁上げ先見の論理が複雑になるため，ビット長が大きい時にハードウェア量が膨大になる．そこで，n ビットの数を k ビットずつのブロックに分け b 個のブロックを作り，各ブロック内で桁上げ先見回路を用いる (Block Carry Look-ahead Circuit，BCLC)．すると，b ブロック目から出力される桁上げ G_b は式 (A.11) と同様の導出で

$$G_b = g_{i+k-1} + p_{i+k-1} \cdot g_{i+k-2} + \cdot + p_{i+k-1} \cdot p_{i+k-2} \cdot \cdots p_{i+1} \cdot c_i \quad \text{(A.12)}$$

と表される．また，b ブロック内で桁上げが発生し，それが b ブロックの外に伝わる条件 P_b は

$$P_b = p_{i+k} \cdot p_{i+k-1} \cdot \cdots \cdot p_{i+1} \cdot p_i \tag{A.13}$$

である．b ブロック内の c_{i+k+1} 桁目における桁上げは，下位ブロックからの桁上げを C_h とすると

$$\begin{aligned} c_{i+k+1} = {} & g_{i+k} + p_{i+k} \cdot g_{i+k-1} + \cdots \\ & + p_{i+k} \cdots p_{i+k} \cdot g_i + p_{i+k} \cdots p_{i+1} \cdot p_i \cdot C_h \end{aligned} \tag{A.14}$$

で表される．

このようなブロック分けを行うことで，ブロック間では桁上げ伝搬が発生するが，ブロック内部では桁上げを抑制することができる．図 A.3 に BCLC を用いたブロック桁上げ先見加算器 (Block Carry Look-ahead Adder，BCLA) の構成を示す．前

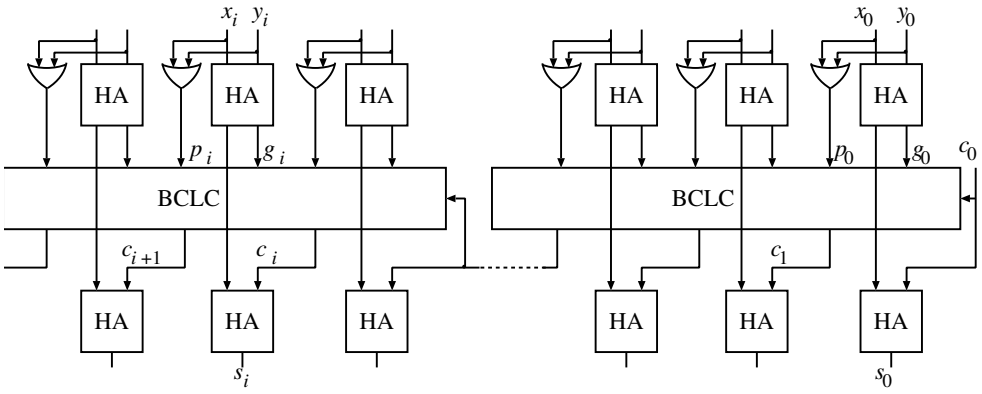

図 A.3 ブロック桁上げ先見加算器 (Block Carry Look-ahead Adder，BCLA) の構成

述のとおり，k が大きくなると回路規模の観点から加算器の構成が難しくなるため，一般に k は 4 などの小さな数にする．

BCLA から出力される G_b，P_b を用いて図 A.4 のように桁上げ先見回路を木構造に組み上げた桁上げ先見加算器を作ることができる．図に示した木構造の BCLA は 2 ブロックを 1 つのまとまりとして下段の桁上げ先見回路を生成している．k 個のブロックを 1 つのまとまりとし下段の桁上げ先見回路を生成してゆくことで n ビット加算器の遅延時間を $O(\log_k n)$ にすることができる．

近年では多くの場合，加算回路として BCLA が利用されるため，単に CLA といった場合にもこの BCLA を意味することが多い．

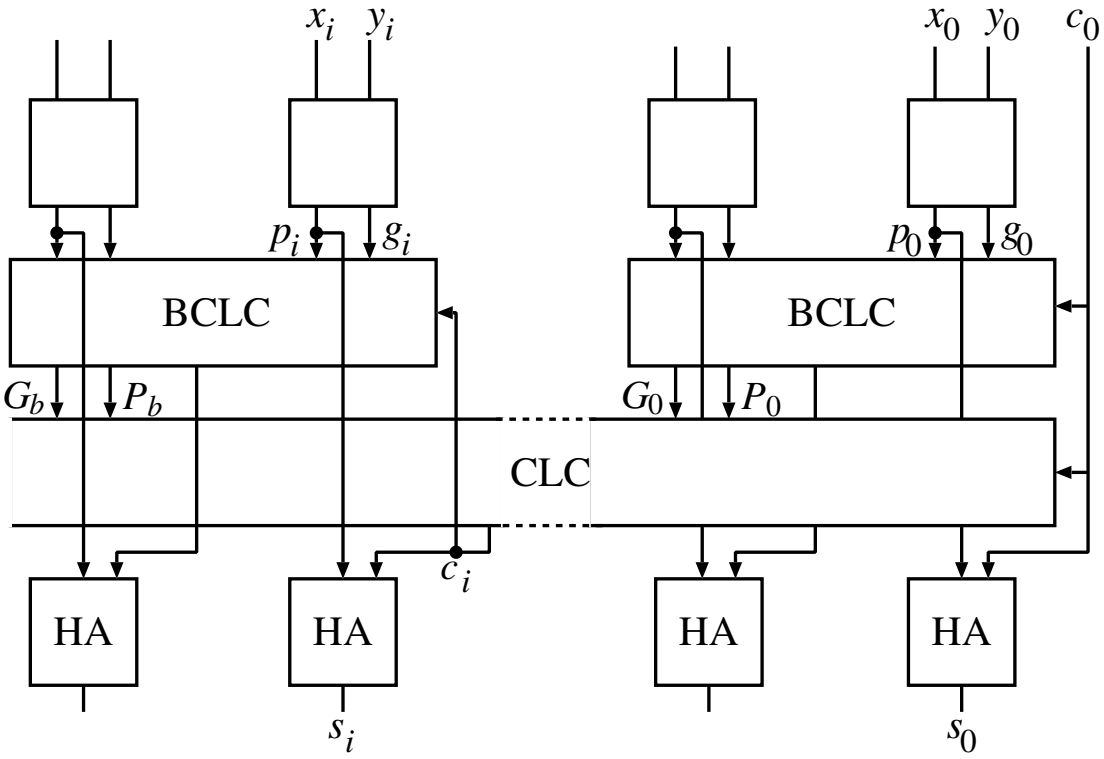

図 A.4　木構造のブロック桁上げ先見加算器の構成

A.3　桁上げ保存加算器 (Carry Save Adder)

今まで述べた加算回路は 2 値の加算を行う回路である．次に，いくつかの値を足し合わせる加算器，つまり合計を求める加算回路について考える．

全加算器は x，y，z を受け取り s，c を出力する．これは見方を変えると，3 値を入力して和を 2 値として得る加算である．通常の加算では，出力 c は桁上げとして上位ビットに渡されるが，多数の値を加算するときは s，c をそのまま維持し，必要になった時に初めて合算するという方法も考えられる．そこで，3 値を 2 値にまとめる桁上げ保存加算器 (Carry Save Adder，CSA) と最終的に得られた s と c を合算する CPA で加算器を構成する．この加算器は 3 値を 2 値にする桁上げ保存加算を 2 値になるまで繰り返し，最終的にその 2 値を CPA で 1 つの値に合算することにより，乗算における部分積の加算など，多数値の累算を効率の良く行うことができる．これを桁上げ保存加算と呼ぶ．CSA の構成は，図 A.5 に示すように，全加算器をビット長の分だけ並べた構造をしている．

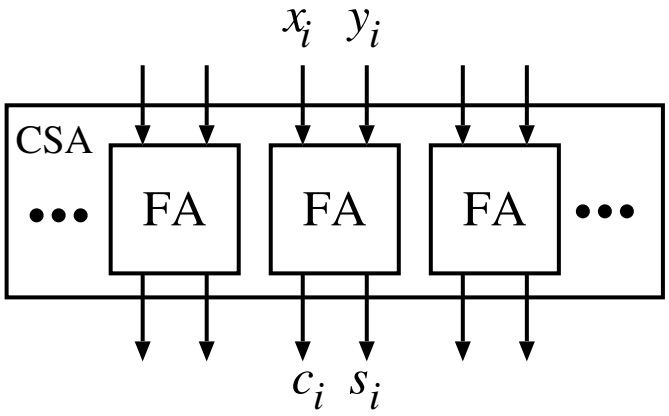

図 A.5　桁上げ保存加算器 (Carry Save Adder，CSA) の構成

付録 B

乗算回路

本章では，本文で扱った多倍長ではなく，32 ビットや 64 ビットなど，比較的小さな桁の乗算を行う乗算器について述べる．

B.1 筆算式乗算回路 (School Book 乗算)

乗算の最も基本的な形は図 B.1 に示す筆算式の乗算である．この方法を School Book 法と呼ぶこともある．School Book 法による乗算回路の構成は単純である．す

$$
\begin{array}{r}
1001 \\
\times\ 1101 \\
\hline
1001 \\
0000 \\
1001 \\
1001 \\
\hline
1110101
\end{array}
$$

図 B.1 筆算式乗算 (School Book 乗算)

なわち，被乗数を下位ビットから上位ビットに向かって 1 ビットずつ走査し，i 桁における値が 1 ならば乗数を i ビット左にシフトしたものを積として累算し，0 ならば何もしない．この時，n ビットどうしの乗算おいて，最大で n 個の値を累算する必要がある．累算される値を部分積と呼ぶ．

B.2 Booth Recode

乗算の過程において，はじめに部分積の生成が必要である．School Book 法を用いた n ビットの乗算において，部分積の数は n 個となるが，この部分積の数を減らすことができれば後の累算を高速に行うことができる．部分積の数を減らす代表的な方法として Booth Recode がある．

Booth Recode の基本的な考え方は基数変換と符号付き桁表現の組み合わせである．通常，計算機上で用いられる 2 進数値は 1 ビットで {0，1} を表現する．これを 2 ビットで {-1，0，1} を表現する符号付きの桁表現に変換する．変換式は，変換後の値の i 桁目を Y_i とすると次式で得られる．

$$Y_i = -1 \cdot y_i + y_{i-1} \tag{B.1}$$

特に $y_{-1} = 0$ ($i = 0$ の場合) とする．この変換により，1 が連続する部分において部分積は 0 となる．そのかわりに連続した 1 が始まる部分で，-1 を乗じた部分積を生成する．すなわちこれは，01110 = 10000 - 00010 であることを利用した変換であると言える．この Booth Recode を用いる際には，変換後の値が符号付きである点に注意する必要がある．

この考え方をさらに拡張する．上述の方法では単なる 2 進数を符号付き桁表現の 2 進数に変換したが，さらに高い基数の数に変換することを考える．基数を高くすることで，複数の桁をまとめて演算することができ，部分積の数を減らすことができる．基数として 4 を選ぶと，1 桁を {-2，-1，0，1，2} で表すことができ，部分積の生成がシフトのみで実現できるため都合がよい．2 進数から符号付き桁表現の 4 進数への変換は次式で与えられる．

$$Y_i = -2 \cdot y_{i+2} + y_i + y_{i-1} \tag{B.2}$$

先程と同様に，$y_{-1} = 0$ とする．この変換により，部分積の数を大幅に削減することができ，部分積の累算にかかる時間を短縮することができる．この方法は 2 次の Booth Recode と呼ばれ，現在，多くの乗算器で用いられている．

B.3 Wallace Tree による加算

乗算の次の過程である部分積の累算について，n 個の部分積を並列に加算することで高速な累算を行うことができる．

今，部分積の数を n とすると，並列化をしていない累算では n 段の加算が必要で

ある．ここで図 B.2 に示すように，$n/2$ 個の CLA を用いて木構造で加算を行うと，部分積を $\log n$ 段の加算で累算することができる．

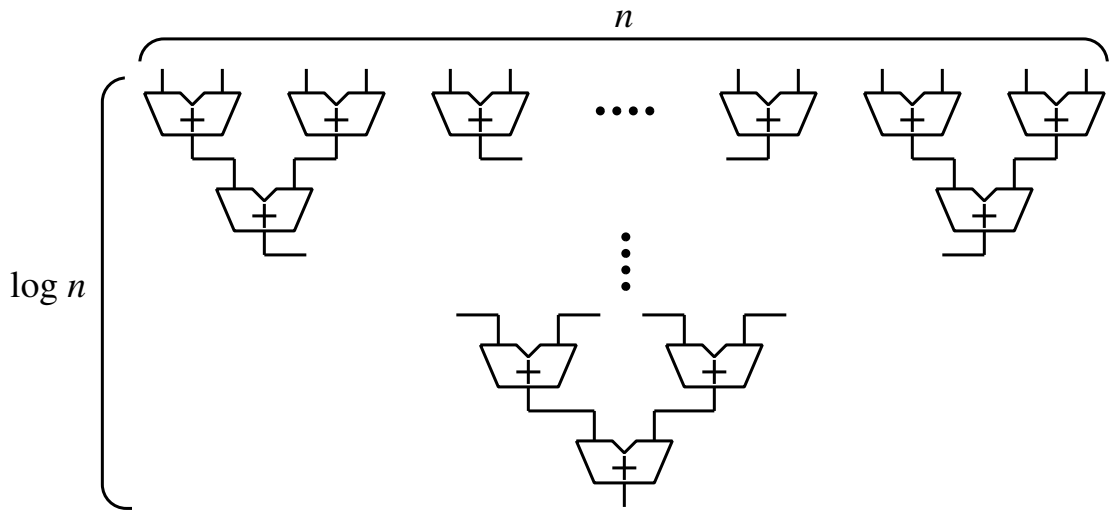

図 B.2　木構造による加算

この累算は，最終段以外の加算を CSA で行い，最後に残った 2 値を CPA で加算すると効率が良い．この考え方に基づいて構成された加算木を Wallace Tree と呼ぶ．Wallace Tree の構成を図 B.3 に示す．中間段に CSA を用いることで，桁上げは最終段でのみ発生するため，累算を高速に行うことが出来る．

B.4　Wallace Tree 乗算器

以上に述べた 2 次の Booth Recode と Wallace Tree を組み合わせることで高速な乗算器を設計することができる．この乗算器は Wallace Tree 乗算器 (Wallace Tree Multiplier，WTM) と呼ばれている．図 B.4 に 8 ビットの数 x と y の積 p を求める 2 次の Booth Recode を用いた Wallace Tree 乗算器の構成を示す．図中の Booth Recoder は 2 次の Booth Recode を行い，結果を出力するモジュールである．この例では 8 ビットの値を変換するため，4 個の Booth コードが得られる．次に，四角で囲まれた P で示されたモジュールは，Booth コードと x から部分積を生成する回路である．具体的には，x と Booth コードの値によって $-2x$，$-x$，0，x，$2x$ を生成する．同時に，右シフトと，必要に応じて符号拡張も行う．こうして得られた部分積を Wallace Tree によって加算する．

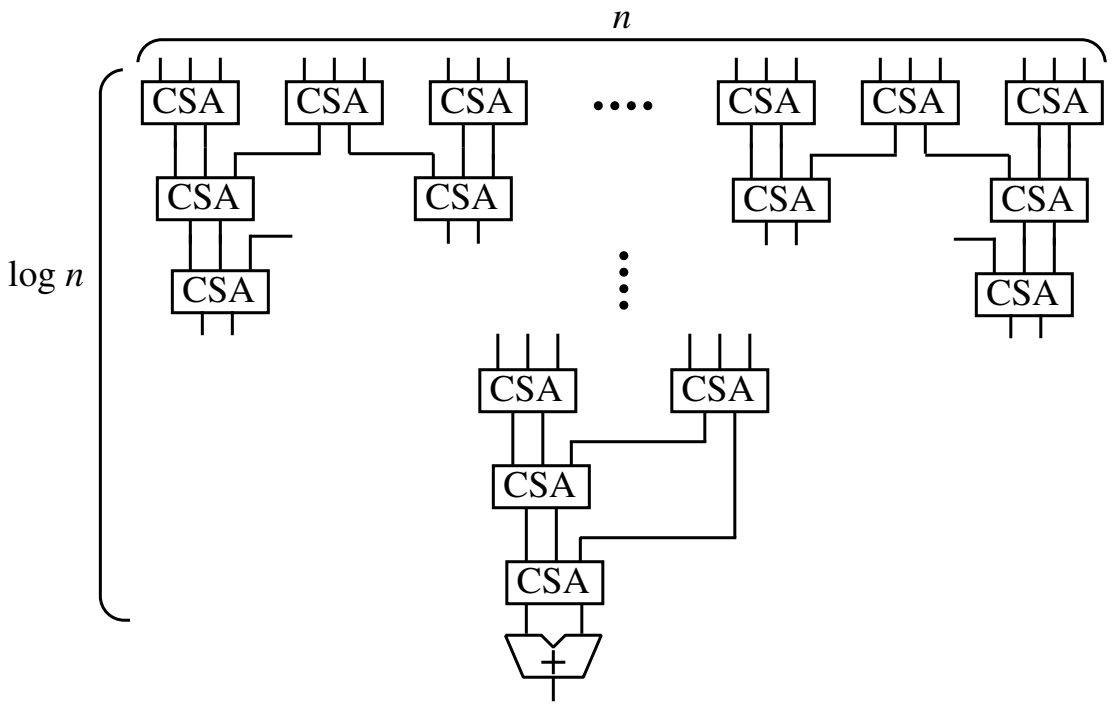

図 B.3　Wallace 木による加算

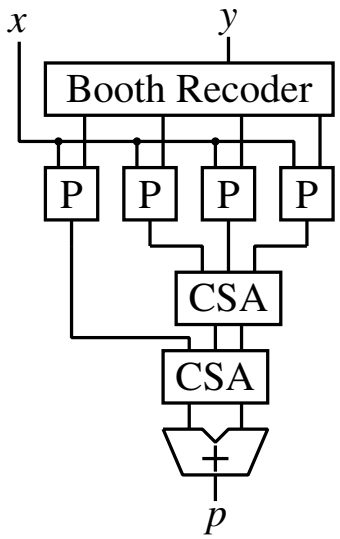

図 B.4　2 次の Booth Recode を用いた Wallace Tree 乗算器

付録 C

Cooley Tukey FFT

本章では，Cooley Tukey FFT アルゴリズムによる DFT の計算量削減手法を具体的に述べ，フロー図の導出を示す．複素ベクトル $x = (x)_{i=0,1,\ldots,N-1}$ の DFT は次式で定義される．

$$X_i = \sum_{k=0}^{N-1} x_k W_N^{ik} \tag{C.1}$$

ここに，j は虚数単位である．W_N^{ik} は回転子と呼ばれ，次式で定義される．

$$\begin{aligned} W_N^{ik} &= e^{\frac{-2\pi ikj}{N}} \\ &= \cos(\frac{2\pi ik}{N}) - j\sin(\frac{2\pi ik}{N}) \end{aligned} \tag{C.2}$$

本章では，$N = 8$ の場合を例にして説明を行う．

まず，アルゴリズムの基本となる回転子の性質について述べる．$N = 8$ における回転子 W_8^{ik} は

$$W_8^{ik} = \cos(\frac{-2\pi ik}{8}) - j\sin(\frac{-2\pi ik}{8}) \tag{C.3}$$

で表現される．この値は，ik によって図 C.1 に示すような周期性を持つ．図は極座標系上に回転子の軌跡を示したものであり，x 軸が実軸，y 軸が虚軸となっている．FFT はこの周期性を利用することで DFT における乗算回数を削減している．

次に，$N = 8$ における FFT アルゴリズムを説明する．$N = 8$ における DFT を

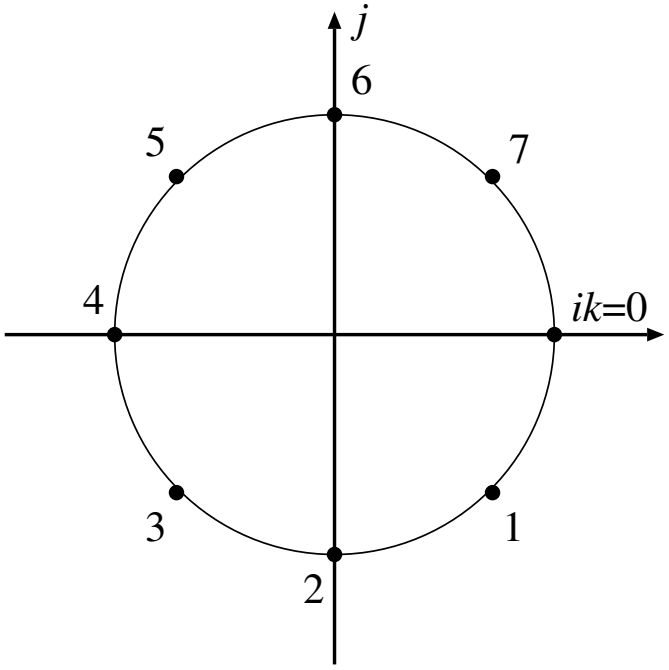

図 C.1　回転子 W_8^{ik} の ik による周期性

行列による式で表現すると次式のようになる．ただし，W_8^{ik} を W^{ik} と略記する．

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^8 & W^{10} & W^{12} & W^{14} \\ W^0 & W^3 & W^6 & W^9 & W^{12} & W^{15} & W^{18} & W^{21} \\ W^0 & W^4 & W^8 & W^{12} & W^{16} & W^{20} & W^{24} & W^{28} \\ W^0 & W^5 & W^{10} & W^{15} & W^{20} & W^{25} & W^{30} & W^{35} \\ W^0 & W^6 & W^{12} & W^{18} & W^{24} & W^{30} & W^{36} & W^{42} \\ W^0 & W^7 & W^{14} & W^{21} & W^{28} & W^{35} & W^{42} & W^{49} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{C.4}$$

先に述べた回転子の周期性により，式 (C.4) に現れた回転子 W^{ik} は $N = 8$ において $W^{(ik \bmod 8)}$ となるため，次式のように書くことができる．

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{C.5}$$

ここで，式 (C.5) の行列を，式 (C.6)，(C.7) のように偶数行と奇数行に分割する．

$$\begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{C.6}$$

$$\begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{C.7}$$

ここで，偶数の添字で構成された式 (C.6) について，

$$\mathcal{W}_{00} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^4 & W^0 & W^4 \\ W^0 & W^6 & W^4 & W^2 \end{bmatrix} \tag{C.8}$$

とおくと，

$$\begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} = (\mathcal{W}_{00} \mid W^0 \cdot \mathcal{W}_{00}) \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{C.9}$$

と書くことができる．ただし，式 (C.9) 中の $\mid$ は行列どうしの単純な結合を意味する．すると，式 (C.9) における乗算について，x_i と W^i，x_{i+4} と W^i $(0 < i < 3)$ の

乗算の和は，$x_i + W^0 \cdot x_{i+4}$ と W^i の乗算と等しい．よって，式 (C.9) は

$$\begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} = \mathcal{W}_{00} \begin{bmatrix} x_0 + W^0 \cdot x_4 \\ x_1 + W^0 \cdot x_5 \\ x_2 + W^0 \cdot x_6 \\ x_3 + W^0 \cdot x_7 \end{bmatrix} \tag{C.10}$$

$$= \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^4 & W^0 & W^4 \\ W^0 & W^6 & W^4 & W^2 \end{bmatrix} \begin{bmatrix} x_0 + W^0 \cdot x_4 \\ x_1 + W^0 \cdot x_5 \\ x_2 + W^0 \cdot x_6 \\ x_3 + W^0 \cdot x_7 \end{bmatrix} \tag{C.11}$$

と変形することができる．式 (C.7) についても同様に

$$\mathcal{W}_{01} = \begin{bmatrix} W^0 & W^1 & W^2 & W^3 \\ W^0 & W^3 & W^6 & W^1 \\ W^0 & W^5 & W^2 & W^7 \\ W^0 & W^7 & W^6 & W^5 \end{bmatrix} \tag{C.12}$$

とすると，式 (C.7) は

$$\begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} = (\mathcal{W}_{01} \mid W^4 \cdot \mathcal{W}_{00}) \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{C.13}$$

と変形することができる．$W^4 = -W^0$ なので，式 (C.13) における乗算について，x_i と W^i，x_{i+4} と W^i $(0 < i < 3)$ の乗算の和は，$x_i - W^0 \cdot x_{i+4}$ と W^i の乗算に等しい．よって，式 (C.13) は

$$\begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} = \mathcal{W}_{01} \begin{bmatrix} x_0 - W^0 \cdot x_4 \\ x_1 - W^0 \cdot x_5 \\ x_2 - W^0 \cdot x_6 \\ x_3 - W^0 \cdot x_7 \end{bmatrix} \tag{C.14}$$

$$= \begin{bmatrix} W^0 & W^1 & W^2 & W^3 \\ W^0 & W^3 & W^6 & W^1 \\ W^0 & W^5 & W^2 & W^7 \\ W^0 & W^7 & W^6 & W^5 \end{bmatrix} \begin{bmatrix} x_0 - W^0 \cdot x_4 \\ x_1 - W^0 \cdot x_5 \\ x_2 - W^0 \cdot x_6 \\ x_3 - W^0 \cdot x_7 \end{bmatrix} \tag{C.15}$$

となる．

さらに式 (C.11)，(C.15) をそれぞれ偶数行と奇数行に分割すると

$$\begin{bmatrix} X_0 \\ X_4 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^4 & W^0 & W^4 \end{bmatrix} \begin{bmatrix} x_0 + W^0 \cdot x_4 \\ x_1 + W^0 \cdot x_5 \\ x_2 + W^0 \cdot x_6 \\ x_3 + W^0 \cdot x_7 \end{bmatrix} \tag{C.16}$$

$$\begin{bmatrix} X_2 \\ X_6 \end{bmatrix} = \begin{bmatrix} W^0 & W^2 & W^4 & W^6 \\ W^0 & W^6 & W^4 & W^2 \end{bmatrix} \begin{bmatrix} x_0 + W^0 \cdot x_4 \\ x_1 + W^0 \cdot x_5 \\ x_2 + W^0 \cdot x_6 \\ x_3 + W^0 \cdot x_7 \end{bmatrix} \tag{C.17}$$

$$\begin{bmatrix} X_1 \\ X_5 \end{bmatrix} = \begin{bmatrix} W^0 & W^1 & W^2 & W^3 \\ W^0 & W^5 & W^2 & W^7 \end{bmatrix} \begin{bmatrix} x_0 - W^0 \cdot x_4 \\ x_1 - W^0 \cdot x_5 \\ x_2 - W^0 \cdot x_6 \\ x_3 - W^0 \cdot x_7 \end{bmatrix} \tag{C.18}$$

$$\begin{bmatrix} X_3 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^3 & W^6 & W^1 \\ W^0 & W^7 & W^6 & W^5 \end{bmatrix} \begin{bmatrix} x_0 - W^0 \cdot x_4 \\ x_1 - W^0 \cdot x_5 \\ x_2 - W^0 \cdot x_6 \\ x_3 - W^0 \cdot x_7 \end{bmatrix} \tag{C.19}$$

となる．ここで,

$$\mathcal{W}_{10} = \begin{bmatrix} W^0 & W^0 \\ W^0 & W^4 \end{bmatrix} \tag{C.20}$$

$$\mathcal{W}_{11} = \begin{bmatrix} W^0 & W^2 \\ W^0 & W^6 \end{bmatrix} \tag{C.21}$$

$$\mathcal{W}_{12} = \begin{bmatrix} W^0 & W^1 \\ W^0 & W^5 \end{bmatrix} \tag{C.22}$$

$$\mathcal{W}_{13} = \begin{bmatrix} W^0 & W^3 \\ W^0 & W^7 \end{bmatrix} \tag{C.23}$$

と置くと，分割した式はそれぞれ

$$\begin{bmatrix} X_0 \\ X_4 \end{bmatrix} = \mathcal{W}_{10} \begin{bmatrix} (x_0 + W^0 \cdot x_4) + W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) + W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.24}$$

$$\begin{bmatrix} X_2 \\ X_6 \end{bmatrix} = \mathcal{W}_{11} \begin{bmatrix} (x_0 + W^0 \cdot x_4) - W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) - W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.25}$$

$$\begin{bmatrix} X_1 \\ X_5 \end{bmatrix} = \mathcal{W}_{12} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^2 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^2 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.26}$$

$$\begin{bmatrix} X_3 \\ X_7 \end{bmatrix} = \mathcal{W}_{13} \begin{bmatrix} (x_0 - W^0 \cdot x_4) - W^2 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) - W^2 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.27}$$

となり,

$$\begin{bmatrix} X_0 \\ X_4 \end{bmatrix} = \begin{bmatrix} W^0 & W^0 \\ W^0 & W^4 \end{bmatrix} \begin{bmatrix} (x_0 + W^0 \cdot x_4) + W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) + W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.28}$$

$$\begin{bmatrix} X_2 \\ X_6 \end{bmatrix} = \begin{bmatrix} W^0 & W^2 \\ W^0 & W^6 \end{bmatrix} \begin{bmatrix} (x_0 + W^0 \cdot x_4) - W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) - W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.29}$$

$$\begin{bmatrix} X_1 \\ X_5 \end{bmatrix} = \begin{bmatrix} W^0 & W^1 \\ W^0 & W^5 \end{bmatrix} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^2 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^2 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.30}$$

$$\begin{bmatrix} X_3 \\ X_7 \end{bmatrix} = \begin{bmatrix} W^0 & W^3 \\ W^0 & W^7 \end{bmatrix} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^6 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^6 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.31}$$

を得る．さらにこれらを分割すると,

$$X_0 = \begin{bmatrix} W^0 & W^0 \end{bmatrix} \begin{bmatrix} (x_0 + W^0 \cdot x_4) + W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) + W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.32}$$

$$X_4 = \begin{bmatrix} W^0 & W^4 \end{bmatrix} \begin{bmatrix} (x_0 + W^0 \cdot x_4) + W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) + W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.33}$$

$$X_2 = \begin{bmatrix} W^0 & W^2 \end{bmatrix} \begin{bmatrix} (x_0 + W^0 \cdot x_4) - W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) - W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.34}$$

$$X_6 = \begin{bmatrix} W^0 & W^6 \end{bmatrix} \begin{bmatrix} (x_0 + W^0 \cdot x_4) - W^0 \cdot (x_2 + W^0 \cdot x_6) \\ (x_1 + W^0 \cdot x_5) - W^0 \cdot (x_3 + W^0 \cdot x_7) \end{bmatrix} \tag{C.35}$$

$$X_1 = \begin{bmatrix} W^0 & W^1 \end{bmatrix} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^2 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^2 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.36}$$

$$X_5 = \begin{bmatrix} W^0 & W^5 \end{bmatrix} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^2 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^2 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.37}$$

$$X_3 = \begin{bmatrix} W^0 & W^3 \end{bmatrix} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^6 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^6 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.38}$$

$$X_7 = \begin{bmatrix} W^0 & W^7 \end{bmatrix} \begin{bmatrix} (x_0 - W^0 \cdot x_4) + W^6 \cdot (x_2 - W^0 \cdot x_6) \\ (x_1 - W^0 \cdot x_5) + W^6 \cdot (x_3 - W^0 \cdot x_7) \end{bmatrix} \tag{C.39}$$

となる，これらを展開することで

$$\begin{aligned} X_0 = {} & W^0\{(x_0 + W^0 \cdot x_4) + W^0 \cdot (x_2 + W^0 \cdot x_6)\} \\ & + W^0\{(x_1 + W^0 \cdot x_5) + W^0 \cdot (x_3 + W^0 \cdot x_7)\} \end{aligned} \tag{C.40}$$

$$\begin{aligned} X_4 = {} & W^0\{(x_0 + W^0 \cdot x_4) + W^0 \cdot (x_2 + W^0 \cdot x_6)\} \\ & - W^0\{(x_1 + W^0 \cdot x_5) + W^0 \cdot (x_3 + W^0 \cdot x_7)\} \end{aligned} \tag{C.41}$$

$$\begin{aligned} X_2 = {} & W^0\{(x_0 + W^0 \cdot x_4) - W^0 \cdot (x_2 + W^0 \cdot x_6)\} \\ & + W^2\{(x_1 + W^0 \cdot x_5) - W^0 \cdot (x_3 + W^0 \cdot x_7)\} \end{aligned} \tag{C.42}$$

$$\begin{aligned} X_6 = {} & W^0\{(x_0 + W^0 \cdot x_4) - W^0 \cdot (x_2 + W^0 \cdot x_6)\} \\ & - W^2\{(x_1 + W^0 \cdot x_5) - W^0 \cdot (x_3 + W^0 \cdot x_7)\} \end{aligned} \tag{C.43}$$

$$
\begin{aligned}
X_1 &= W^0\{(x_0 - W^0 \cdot x_4) + W^2 \cdot (x_2 - W^0 \cdot x_6)\} \\
&\quad + W^1\{(x_1 - W^0 \cdot x_5) + W^2 \cdot (x_3 - W^0 \cdot x_7)\} \qquad \text{(C.44)} \\
X_5 &= W^0\{(x_0 - W^0 \cdot x_4) + W^2 \cdot (x_2 - W^0 \cdot x_6)\} \\
&\quad - W^1\{(x_1 - W^0 \cdot x_5) + W^2 \cdot (x_3 - W^0 \cdot x_7)\} \qquad \text{(C.45)}
\end{aligned}
$$

$$
\begin{aligned}
X_3 &= W^0\{(x_0 - W^0 \cdot x_4) - W^2 \cdot (x_2 - W^0 \cdot x_6)\} \\
&\quad + W^3\{(x_1 - W^0 \cdot x_5) - W^2 \cdot (x_3 - W^0 \cdot x_7)\} \qquad \text{(C.46)} \\
X_7 &= W^0\{(x_0 - W^0 \cdot x_4) - W^2 \cdot (x_2 - W^0 \cdot x_6)\} \\
&\quad - W^3\{(x_1 - W^0 \cdot x_5) - W^2 \cdot (x_3 - W^0 \cdot x_7)\} \qquad \text{(C.47)}
\end{aligned}
$$

を得る．式 (C.40) から (C.47) をフロー図で表すと図 3.7 になる．

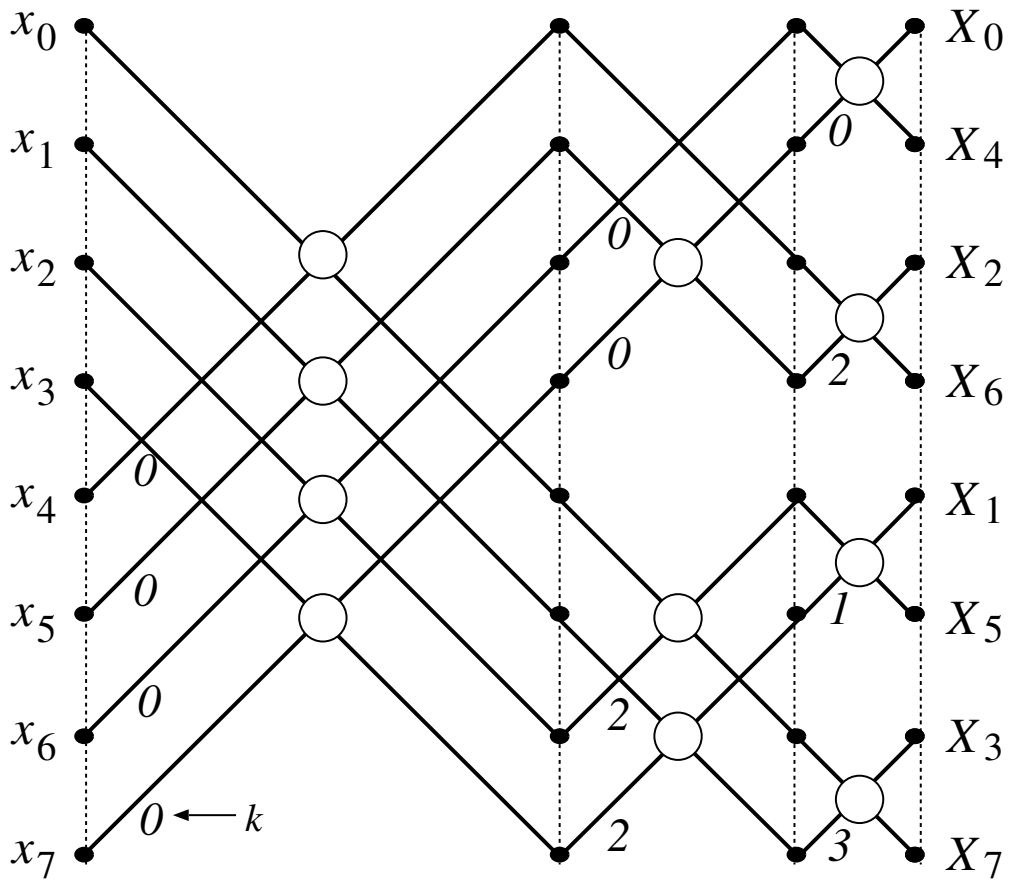

図 C.2　$N = 8$ における基数 2 の時間間引き型 FFT の演算フロー

付録 D

R1IKM と R3IKM における PPG と ACC のスケジューリング

本章では，第 4.5.1 節で述べた再帰の回数を 1 回と 3 回にした IKM における PPG と ACC のスケジューリングを示す．それぞれの IKM を R1IKM，R3IKM と呼ぶ．スケジューリングにおいて，PPG と ACC で用いられる演算器の数は第 4.5.2 節で述べたものと同じとした．

D.1 R1IKM

D.1.1 PPG

表 D.1 R1IKM の PPG におけるスケジューリング

		Multiplier1		Adder1		Adder2	
*hline*Step	Input	Input	Output	Input	Output	Input	Output
1	a_0, b_0	−	−	−	−	−	−
2	a_1, b_1	a_0, b_0	−	−	−	−	−
3	−	a_1, b_1	−	a_1, a_0	−	b_1, b_0	−
4	−	a_{10}, b_{10}	−	−	a_{10}	−	b_{10}
5	−	−	−	−	−	−	−
6	−	−	−	−	−	−	−
7	−	−	−	−	−	−	−
8	−	−	$a_0 b_0 (= pp0)$	−	−	−	−
9	−	−	$a_1 b_1 (= pp1)$	−	−	−	−
10	−	−	$a_{10} b_{10} (= pp2)$	−	−	−	−

D.1.2 ACC

表 D.2 R1IKM の ACC におけるスケジューリング (1/2)

		Adder-Subtracter1		Adder-Subtracter2	
Step	Input	Input	Output	Input	Output
1	$pp0$	−	−	−	−
2	$pp1$	$p0, pp0\mathrm{L}$	−	$p1, pp0\mathrm{L}$	−
3	$pp2$	$p1, pp0\mathrm{H}$	$p0 + pp0\mathrm{L}$	$p2, pp1\mathrm{L}$	$p1 - pp0\mathrm{L}$
4	−	$p1, pp1\mathrm{L}$	$p1 + pp0\mathrm{H}$	$p2, pp1\mathrm{H}$	$p2 + pp0\mathrm{L}$
5	−	$p1, pp1\mathrm{L}$	$p1 - pp1\mathrm{L}$	$p2, pp1\mathrm{H}$	$p2 - pp1\mathrm{H}$
6	−	−	$p1 + pp2\mathrm{L}$	−	$p2 + pp2\mathrm{H}$

表 D.3 R1IKM の ACC におけるスケジューリング (2/2)

		Adder-Subtracter3		Adder-Subtracter4	
Step	Input	Input	Output	Input	Output
1	$pp0$	−	−	−	−
2	$pp1$	$p2, pp0\mathrm{H}$	−	−	−
3	$pp2$	$p3, pp1\mathrm{H}$	$p2 - pp0\mathrm{H}$	−	−
4	−	−	$p3 + pp1\mathrm{H}$	−	−
5	−	−	−	−	−
6	−	−	−	−	−

D.2 R3IKM のスケジューリング

D.2.1 PPG

表 D.4 R3IKM の PPG におけるスケジューリング (1/2)

		Multiplier1	
Step	Input	Input	Output
1	a_0, b_0	–	–
2	a_1, b_1	a_0, b_0	–
3	a_2, b_2	a_1, b_1	–
4	a_3, b_3	a_2, b_2	–
5	a_4, b_4	a_3, b_3	–
6	a_5, b_5	a_4, b_4	–
7	a_6, b_6	a_5, b_5	–
8	a_7, b_7	a_6, b_6	$a_0b_0(=pp0)$
9	–	a_7, b_7	$a_1b_1(=pp1)$
10	–	a_{10}, b_{10}	$a_2b_2(=pp2)$
11	–	a_{20}, b_{20}	$a_3b_3(=pp3)$
12	–	a_{31}, b_{31}	$a_4b_4(=pp4)$
13	–	a_{32}, b_{32}	$a_5b_5(=pp5)$
14	–	a_{40}, b_{40}	$a_6b_6(=pp6)$
15	–	a_{51}, b_{51}	$a_7b_7(=pp7)$
16	–	a_{54}, b_{54}	$a_{10}b_{10}(=pp8)$
17	–	a_{62}, b_{62}	$a_{20}b_{20}(=pp9)$
18	–	a_{64}, b_{64}	$a_{31}b_{31}(=pp10)$
19	–	a_{73}, b_{73}	$a_{32}b_{32}(=pp11)$
20	–	a_{75}, b_{75}	$a_{40}b_{40}(=pp12)$
21	–	a_{76}, b_{76}	$a_{51}b_{51}(=pp13)$
22	–	a_{3120}, b_{3120}	$a_{54}b_{54}(=pp14)$
23	–	a_{5140}, b_{5140}	$a_{62}b_{62}(=pp15)$
24	–	a_{6240}, b_{6240}	$a_{64}b_{64}(=pp16)$
25	–	a_{7351}, b_{7351}	$a_{73}b_{73}(=pp17)$
26	–	a_{7362}, b_{7362}	$a_{75}b_{75}(=pp18)$
27	–	a_{7564}, b_{7564}	$a_{76}b_{76}(=pp19)$
28	–	$a_{73516240}, b_{73516240}$	$a_{3120}b_{3120}(=pp20)$
29	–	–	$a_{5140}b_{5140}(=pp21)$
30	–	–	$a_{6240}b_{6240}(=pp22)$
31	–	–	$a_{7351}b_{7351}(=pp23)$
32	–	–	$a_{7362}b_{7362}(=pp24)$
33	–	–	$a_{7564}b_{7564}(=pp25)$
34	–	–	$a_{73516240}b_{73516240}(=pp26)$

表 D.5 R3IKM の PPG におけるスケジューリング (2/2)

		Adder1		Adder2	
Step	Input	Input	Output	Input	Output
1	a_0, b_0	–	–	–	–
2	a_1, b_1	–	–	–	–
3	a_2, b_2	a_1, a_0	–	b_1, b_0	–
4	a_3, b_3	a_2, a_0	a_{10}	b_2, b_0	b_{10}
5	a_4, b_4	a_3, a_1	a_{20}	b_3, b_1	b_{20}
6	a_5, b_5	a_3, a_2	a_{31}	b_3, b_2	b_{31}
7	a_6, b_6	a_4, a_0	a_{32}	b_4, b_0	b_{32}
8	a_7, b_7	a_5, a_1	a_{40}	b_5, b_1	b_{40}
9	–	a_5, a_4	a_{51}	b_5, b_4	b_{51}
10	–	a_6, a_2	a_{54}	b_6, b_2	b_{54}
11	–	a_6, a_4	a_{62}	b_6, b_4	b_{62}
12	–	a_7, a_3	a_{64}	b_7, b_3	b_{64}
13	–	a_7, a_5	a_{73}	b_7, b_5	b_{73}
14	–	a_7, a_6	a_{75}	b_7, b_6	b_{75}
15	–	a_{31}, a_{20}	a_{76}	b_{31}, b_{20}	b_{76}
16	–	a_{51}, a_{40}	a_{3120}	b_{51}, b_{40}	b_{3120}
17	–	a_{62}, a_{40}	a_{5140}	b_{62}, b_{40}	b_{5140}
18	–	a_{73}, a_{51}	a_{6240}	b_{73}, b_{51}	b_{6240}
19	–	a_{73}, a_{62}	a_{7351}	b_{73}, b_{62}	b_{7351}
20	–	a_{75}, a_{64}	a_{7362}	b_{75}, b_{64}	b_{7362}
21	–	a_{7351}, a_{6240}	a_{7564}	b_{7351}, b_{6240}	b_{7564}
22	–	–	$a_{73516240}$	–	$b_{73516240}$
23	–	–	–	–	–
24	–	–	–	–	–
25	–	–	–	–	–
26	–	–	–	–	–
27	–	–	–	–	–
28	–	–	–	–	–
29	–	–	–	–	–
30	–	–	–	–	–
31	–	–	–	–	–
32	–	–	–	–	–
33	–	–	–	–	–
34	–	–	–	–	–

D.2.2 ACC

表 D.6 R3IKM の ACC におけるスケジューリング (1/4)

		Adder-Subtracter1		Adder-Subtracter2	
Step	Input	Input	Output	Input	Output
1	$pp0$	–	–	–	
2	$pp1$	$p0, pp0\mathrm{L}$	–	$p1, pp0\mathrm{L}$	–
3	$pp2$	$p4, pp0\mathrm{L}$	$p0 + pp0\mathrm{L}$	$p5, pp0\mathrm{L}$	$p1 - pp0\mathrm{L}$
4	$pp3$	$p1, pp0\mathrm{H}$	$p4 - pp0\mathrm{L}$	$p2, pp0\mathrm{H}$	$p5 + pp0\mathrm{L}$
5	$pp4$	$p5, pp0\mathrm{H}$	$p1 + pp0\mathrm{H}$	$p6, pp0\mathrm{H}$	$p2 - pp0\mathrm{H}$
6	$pp5$	$p1, pp1\mathrm{L}$	$p5 - pp0\mathrm{H}$	$p2, pp1\mathrm{L}$	$p6 + pp0\mathrm{H}$
7	$pp6$	$p5, pp1\mathrm{L}$	$p1 - pp1\mathrm{L}$	$p6, pp1\mathrm{L}$	$p2 + pp1\mathrm{L}$
8	$pp7$	$p2, pp1\mathrm{H}$	$p5 + pp1\mathrm{L}$	$p3, pp1\mathrm{H}$	$p6 - pp1\mathrm{L}$
9	$pp8$	$p6, pp1\mathrm{H}$	$p2 - pp1\mathrm{H}$	$p7, pp1\mathrm{H}$	$p3 + pp1\mathrm{H}$
10	$pp9$	$p2, pp2\mathrm{L}$	$p6 + pp1\mathrm{H}$	$p3, pp2\mathrm{L}$	$p7 - pp1\mathrm{H}$
11	$pp10$	$p6, pp2\mathrm{L}$	$p2 - pp2\mathrm{L}$	$p7, pp2\mathrm{L}$	$p3 + pp2\mathrm{L}$
12	$pp11$	$p3, pp2\mathrm{H}$	$p6 + pp2\mathrm{L}$	$p4, pp2\mathrm{H}$	$p7 - pp2\mathrm{L}$
13	$pp12$	$p7, pp2\mathrm{H}$	$p3 - pp2\mathrm{H}$	$p8, pp2\mathrm{H}$	$p4 + pp2\mathrm{H}$
14	$pp13$	$p3, pp3\mathrm{L}$	$p7 + pp2\mathrm{H}$	$p4, pp3\mathrm{L}$	$p8 - pp2\mathrm{H}$
15	$pp14$	$p7, pp3\mathrm{L}$	$p3 + pp3\mathrm{L}$	$p8, pp3\mathrm{L}$	$p4 - pp3\mathrm{L}$
16	$pp15$	$p4, pp3\mathrm{H}$	$p7 - pp3\mathrm{L}$	$p5, pp3\mathrm{H}$	$p8 + pp3\mathrm{L}$
17	$pp16$	$p8, pp3\mathrm{H}$	$p4 + pp3\mathrm{H}$	$p9, pp3\mathrm{H}$	$p5 - pp3\mathrm{H}$
18	$pp17$	$p4, pp4\mathrm{L}$	$p8 - pp3\mathrm{H}$	$p5, pp4\mathrm{L}$	$p9 + pp3\mathrm{H}$
19	$pp18$	$p8, pp4\mathrm{L}$	$p4 - pp4\mathrm{L}$	$p9, pp4\mathrm{L}$	$p5 + pp4\mathrm{L}$
20	$pp19$	$p5, pp4\mathrm{H}$	$p8 + pp4\mathrm{L}$	$p6, pp4\mathrm{H}$	$p9 - pp4\mathrm{L}$
21	$pp20$	$p9, pp4\mathrm{H}$	$p5 - pp4\mathrm{H}$	$p10, pp4\mathrm{H}$	$p6 + pp4\mathrm{H}$
22	$pp21$	$p5, pp5\mathrm{L}$	$p9 + pp4\mathrm{H}$	$p6, pp5\mathrm{L}$	$p10 - pp4\mathrm{H}$
23	$pp22$	$p9, pp5\mathrm{L}$	$p5 + pp5\mathrm{L}$	$p10, pp5\mathrm{L}$	$p6 - pp5\mathrm{L}$
24	$pp23$	$p6, pp5\mathrm{H}$	$p9 - pp5\mathrm{L}$	$p7, pp5\mathrm{H}$	$p10 + pp5\mathrm{L}$
25	$pp24$	$p10, pp5\mathrm{H}$	$p6 + pp5\mathrm{H}$	$p11, pp5\mathrm{H}$	$p7 - pp5\mathrm{H}$
26	$pp25$	$p6, pp6\mathrm{L}$	$p10 - pp5\mathrm{H}$	$p7, pp6\mathrm{L}$	$p11 + pp5\mathrm{H}$
27	$pp26$	$p10, pp6\mathrm{L}$	$p6 + pp6\mathrm{L}$	$p11, pp6\mathrm{L}$	$p7 - pp6\mathrm{L}$
28	–	$p7, pp6\mathrm{H}$	$p10 - pp6\mathrm{L}$	$p8, pp6\mathrm{H}$	$p11 + pp6\mathrm{L}$
29	–	$p11, pp6\mathrm{H}$	$p7 + pp6\mathrm{H}$	$p12, pp6\mathrm{H}$	$p8 - pp6\mathrm{H}$
30	–	$p7, pp7\mathrm{L}$	$p11 - pp6\mathrm{H}$	$p8, pp7\mathrm{L}$	$p12 + pp6\mathrm{H}$
31	–	$p11, pp7\mathrm{L}$	$p7 - pp7\mathrm{L}$	$p12, pp7\mathrm{L}$	$p8 + pp7\mathrm{L}$
32	–	$p8, pp7\mathrm{H}$	$p11 + pp7\mathrm{L}$	$p9, pp7\mathrm{H}$	$p12 - pp7\mathrm{L}$
33	–	$p12, pp7\mathrm{H}$	$p8 - pp7\mathrm{H}$	$p13, pp7\mathrm{H}$	$p9 + pp7\mathrm{H}$

表 D.7　R3IKM の ACC におけるスケジューリング (2/4)

		Adder-Subtracter1		Adder-Subtracter2	
Step	Input	Input	Output	Input	Output
34	–	$p1, pp8\mathrm{L}$	$p12 + pp7\mathrm{H}$	$p3, pp8\mathrm{L}$	$p13 - pp7\mathrm{H}$
35	–	$p2, pp8\mathrm{H}$	$p1 + pp8\mathrm{L}$	$p4, pp8\mathrm{H}$	$p3 - pp8\mathrm{L}$
36	–	$p2, pp9\mathrm{L}$	$p2 + pp8\mathrm{H}$	$p3, pp9\mathrm{L}$	$p4 - pp8\mathrm{H}$
37	–	$p3, pp9\mathrm{H}$	$p2 + pp9\mathrm{L}$	$p4, pp9\mathrm{H}$	$p3 - pp9\mathrm{L}$
38	–	$p3, pp10\mathrm{L}$	$p3 + pp9\mathrm{H}$	$p4, pp10\mathrm{L}$	$p4 - pp9\mathrm{H}$
39	–	$p4, pp10\mathrm{H}$	$p3 - pp10\mathrm{L}$	$p5, pp10\mathrm{H}$	$p4 + pp10\mathrm{L}$
40	–	$p3, pp11\mathrm{L}$	$p4 - pp10\mathrm{H}$	$p5, pp11\mathrm{L}$	$p5 + pp10\mathrm{H}$
41	–	$p4, pp11\mathrm{H}$	$p3 - pp11\mathrm{L}$	$p6, pp11\mathrm{H}$	$p5 + pp11\mathrm{L}$
42	–	$p4, pp12\mathrm{L}$	$p4 - pp11\mathrm{H}$	$p5, pp12\mathrm{L}$	$p6 + pp11\mathrm{H}$
43	–	$p5, pp12\mathrm{H}$	$p4 + pp12\mathrm{L}$	$p6, pp12\mathrm{H}$	$p5 - pp12\mathrm{L}$
44	–	$p5, pp13\mathrm{L}$	$p5 + pp12\mathrm{H}$	$p6, pp13\mathrm{L}$	$p6 - pp12\mathrm{H}$
45	–	$p6, pp13\mathrm{H}$	$p5 - pp13\mathrm{L}$	$p7, pp13\mathrm{H}$	$p6 + pp13\mathrm{L}$
46	–	$p5, pp14\mathrm{L}$	$p6 - pp13\mathrm{H}$	$p7, pp14\mathrm{L}$	$p7 + pp13\mathrm{H}$
47	–	$p6, pp14\mathrm{H}$	$p5 - pp14\mathrm{L}$	$p8, pp14\mathrm{H}$	$p7 + pp14\mathrm{L}$
48	–	$p6, pp15\mathrm{L}$	$p6 - pp14\mathrm{H}$	$p7, pp15\mathrm{L}$	$p8 + pp14\mathrm{H}$
49	–	$p7, pp15\mathrm{H}$	$p6 - pp15\mathrm{L}$	$p8, pp15\mathrm{H}$	$p7 + pp15\mathrm{L}$
50	–	$p6, pp16\mathrm{L}$	$p7 - pp15\mathrm{H}$	$p7, pp16\mathrm{L}$	$p8 + pp15\mathrm{H}$
51	–	$p7, pp16\mathrm{H}$	$p6 - pp16\mathrm{L}$	$p8, pp16\mathrm{H}$	$p7 + pp16\mathrm{L}$
52	–	$p7, pp17\mathrm{L}$	$p7 - pp16\mathrm{H}$	$p8, pp17\mathrm{L}$	$p8 + pp16\mathrm{H}$
53	–	$p8, pp17\mathrm{H}$	$p7 + pp17\mathrm{L}$	$p9, pp17\mathrm{H}$	$p8 - pp17\mathrm{L}$
54	–	$p7, pp18\mathrm{L}$	$p8 + pp17\mathrm{H}$	$p8, pp18\mathrm{L}$	$p9 - pp17\mathrm{H}$
55	–	$p8, pp18\mathrm{H}$	$p7 + pp18\mathrm{L}$	$p9, pp18\mathrm{H}$	$p8 - pp18\mathrm{L}$
56	–	$p7, pp19\mathrm{L}$	$p8 + pp18\mathrm{H}$	$p9, pp19\mathrm{L}$	$p9 - pp18\mathrm{H}$
57	–	$p8, pp19\mathrm{H}$	$p7 + pp19\mathrm{L}$	$p10, pp19\mathrm{H}$	$p9 - pp19\mathrm{L}$
58	–	$p3, pp20\mathrm{L}$	$p8 + pp19\mathrm{H}$	$p7, pp20\mathrm{L}$	$p10 - pp19\mathrm{H}$
59	–	$p5, pp21\mathrm{L}$	$p3 + pp20\mathrm{L}$	$p7, pp21\mathrm{L}$	$p7 - pp20\mathrm{L}$
60	–	$p6, pp22\mathrm{L}$	$p5 + pp21\mathrm{L}$	$p7, pp22\mathrm{L}$	$p7 - pp21\mathrm{L}$
61	–	$p7, pp22\mathrm{H}$	$p6 + pp22\mathrm{L}$	$p8, pp23\mathrm{L}$	$p7 - pp22\mathrm{L}$
62	–	$p7, pp23\mathrm{L}$	$p7 + pp22\mathrm{H}$	$p8, pp23\mathrm{H}$	$p8 + pp23\mathrm{L}$
63	–	$p7, pp24\mathrm{L}$	$p7 - pp23\mathrm{L}$	$p9, pp24\mathrm{L}$	$p8 - pp23\mathrm{H}$
64	–	$p7, pp25\mathrm{L}$	$p7 - pp24\mathrm{L}$	$p11, pp25\mathrm{L}$	$p9 + pp24\mathrm{L}$
65	–	$p7, pp26\mathrm{L}$	$p7 - pp25\mathrm{L}$	$p8, pp26\mathrm{H}$	$p11 + pp25\mathrm{L}$
66	–	–	$p7 + pp26\mathrm{L}$	–	$p8 + pp26\mathrm{H}$

表 D.8　R3IKM の ACC におけるスケジューリング (3/4)

		Adder-Subtracter3		Adder-Subtracter4	
Step	Input	Input	Output	Input	Output
1	$pp0$	−	−	−	−
2	$pp1$	$p2, pp0\mathrm{L}$	−	$p3, pp0\mathrm{L}$	−
3	$pp2$	$p6, pp0\mathrm{L}$	$p2 - pp0\mathrm{L}$	$p7, pp0\mathrm{L}$	$p3 + pp0\mathrm{L}$
4	$pp3$	$p3, pp0\mathrm{H}$	$p6 + pp0\mathrm{L}$	$p4, pp0\mathrm{H}$	$p7 - pp0\mathrm{L}$
5	$pp4$	$p7, pp0\mathrm{H}$	$p3 - pp0\mathrm{H}$	$p8, pp0\mathrm{H}$	$p4 + pp0\mathrm{H}$
6	$pp5$	$p3, pp1\mathrm{L}$	$p7 + pp0\mathrm{H}$	$p4, pp1\mathrm{L}$	$p8 - pp0\mathrm{H}$
7	$pp6$	$p7, pp1\mathrm{L}$	$p3 + pp1\mathrm{L}$	$p8, pp1\mathrm{L}$	$p4 - pp1\mathrm{L}$
8	$pp7$	$p4, pp1\mathrm{H}$	$p7 - pp1\mathrm{L}$	$p5, pp1\mathrm{H}$	$p8 + pp1\mathrm{L}$
9	$pp8$	$p8, pp1\mathrm{H}$	$p4 + pp1\mathrm{H}$	$p9, pp1\mathrm{H}$	$p5 - pp1\mathrm{H}$
10	$pp9$	$p4, pp2\mathrm{L}$	$p8 - pp1\mathrm{H}$	$p5, pp2\mathrm{L}$	$p9 + pp1\mathrm{H}$
11	$pp10$	$p8, pp2\mathrm{L}$	$p4 + pp2\mathrm{L}$	$p9, pp2\mathrm{L}$	$p5 - pp2\mathrm{L}$
12	$pp11$	$p5, pp2\mathrm{H}$	$p8 - pp2\mathrm{L}$	$p6, pp2\mathrm{H}$	$p9 + pp2\mathrm{L}$
13	$pp12$	$p9, pp2\mathrm{H}$	$p5 + pp2\mathrm{H}$	$p10, pp2\mathrm{H}$	$p6 - pp2\mathrm{H}$
14	$pp13$	$p5, pp3\mathrm{L}$	$p9 - pp2\mathrm{H}$	$p6, pp3\mathrm{L}$	$p10 + pp2\mathrm{H}$
15	$pp14$	$p9, pp3\mathrm{L}$	$p5 - pp3\mathrm{L}$	$p10, pp3\mathrm{L}$	$p6 + pp3\mathrm{L}$
16	$pp15$	$p6, pp3\mathrm{H}$	$p9 + pp3\mathrm{L}$	$p7, pp3\mathrm{H}$	$p10 - pp3\mathrm{L}$
17	$pp16$	$p10, pp3\mathrm{H}$	$p6 - pp3\mathrm{H}$	$p11, pp3\mathrm{H}$	$p7 + pp3\mathrm{H}$
18	$pp17$	$p6, pp4\mathrm{L}$	$p10 + pp3\mathrm{H}$	$p7, pp4\mathrm{L}$	$p11 - pp3\mathrm{H}$
19	$pp18$	$p10, pp4\mathrm{L}$	$p6 + pp4\mathrm{L}$	$p11, pp4\mathrm{L}$	$p7 - pp4\mathrm{L}$
20	$pp19$	$p7, pp4\mathrm{H}$	$p10 - pp4\mathrm{L}$	$p8, pp4\mathrm{H}$	$p11 + pp4\mathrm{L}$
21	$pp20$	$p11, pp4\mathrm{H}$	$p7 + pp4\mathrm{H}$	$p12, pp4\mathrm{H}$	$p8 - pp4\mathrm{H}$
22	$pp21$	$p7, pp5\mathrm{L}$	$p11 - pp4\mathrm{H}$	$p8, pp5\mathrm{L}$	$p12 + pp4\mathrm{H}$
23	$pp22$	$p11, pp5\mathrm{L}$	$p7 - pp5\mathrm{L}$	$p12, pp5\mathrm{L}$	$p8 + pp5\mathrm{L}$
24	$pp23$	$p8, pp5\mathrm{H}$	$p11 + pp5\mathrm{L}$	$p9, pp5\mathrm{H}$	$p12 - pp5\mathrm{L}$
25	$pp24$	$p12, pp5\mathrm{H}$	$p8 - pp5\mathrm{H}$	$p13, pp5\mathrm{H}$	$p9 + pp5\mathrm{H}$
26	$pp25$	$p8, pp6\mathrm{L}$	$p12 + pp5\mathrm{H}$	$p9, pp6\mathrm{L}$	$p13 - pp5\mathrm{H}$
27	$pp26$	$p12, pp6\mathrm{L}$	$p8 - pp6\mathrm{L}$	$p13, pp6\mathrm{L}$	$p9 + pp6\mathrm{L}$
28	−	$p9, pp6\mathrm{H}$	$p12 + pp6\mathrm{L}$	$p10, pp6\mathrm{H}$	$p13 - pp6\mathrm{L}$
29	−	$p13, pp6\mathrm{H}$	$p9 - pp6\mathrm{H}$	$p14, pp6\mathrm{H}$	$p10 + pp6\mathrm{H}$
30	−	$p9, pp7\mathrm{L}$	$p13 + pp6\mathrm{H}$	$p10, pp7\mathrm{L}$	$p14 - pp6\mathrm{H}$
31	−	$p13, pp7\mathrm{L}$	$p9 + pp7\mathrm{L}$	$p14, pp7\mathrm{L}$	$p10 - pp7\mathrm{L}$
32	−	$p10, pp7\mathrm{H}$	$p13 - pp7\mathrm{L}$	$p11, pp7\mathrm{H}$	$p14 + pp7\mathrm{L}$
33	−	$p14, pp7\mathrm{H}$	$p10 + pp7\mathrm{H}$	$p15, pp7\mathrm{H}$	$p11 - pp7\mathrm{H}$

表 D.9　R3IKM の ACC におけるスケジューリング (4/4)

		Adder-Subtracter3		Adder-Subtracter4	
Step	Input	Input	Output	Input	Output
34	–	$p5, pp8\mathrm{L}$	$p14 - pp7\mathrm{H}$	$p7, pp8\mathrm{L}$	$p15 + pp7\mathrm{H}$
35	–	$p6, pp8\mathrm{H}$	$p5 - pp8\mathrm{L}$	$p8, pp8\mathrm{H}$	$p7 + pp8\mathrm{L}$
36	–	$p6, pp9\mathrm{L}$	$p6 - pp8\mathrm{H}$	$p7, pp9\mathrm{L}$	$p8 + pp8\mathrm{H}$
37	–	$p7, pp9\mathrm{H}$	$p6 - pp9\mathrm{L}$	$p8, pp9\mathrm{H}$	$p7 + pp9\mathrm{L}$
38	–	$p7, pp10\mathrm{L}$	$p7 - pp9\mathrm{H}$	$p8, pp10\mathrm{L}$	$p8 + pp9\mathrm{H}$
39	–	$p8, pp10\mathrm{H}$	$p7 + pp10\mathrm{L}$	$p9, pp10\mathrm{H}$	$p8 - pp10\mathrm{L}$
40	–	$p7, pp11\mathrm{L}$	$p8 + pp10\mathrm{H}$	$p8, pp11\mathrm{L}$	$p9 - pp10\mathrm{H}$
41	–	$p8, pp11\mathrm{H}$	$p7 + pp11\mathrm{L}$	$p10, pp11\mathrm{H}$	$p9 - pp11\mathrm{L}$
42	–	$p6, pp12\mathrm{L}$	$p8 + pp11\mathrm{H}$	$p7, pp12\mathrm{L}$	$p10 - pp11\mathrm{H}$
43	–	$p7, pp12\mathrm{H}$	$p6 - pp12\mathrm{L}$	$p8, pp12\mathrm{H}$	$p7 + pp12\mathrm{L}$
44	–	$p7, pp13\mathrm{L}$	$p7 - pp12\mathrm{H}$	$p8, pp13\mathrm{L}$	$p8 + pp12\mathrm{H}$
45	–	$p8, pp13\mathrm{H}$	$p7 + pp13\mathrm{L}$	$p9, pp13\mathrm{H}$	$p8 - pp13\mathrm{L}$
46	–	$p9, pp14\mathrm{L}$	$p8 + pp13\mathrm{H}$	$p10, pp14\mathrm{L}$	$p9 - pp13\mathrm{H}$
47	–	$p10, pp14\mathrm{H}$	$p9 + pp14\mathrm{L}$	$p12, pp14\mathrm{H}$	$p11 - pp14\mathrm{L}$
48	–	$p8, pp15\mathrm{L}$	$p10 + pp14\mathrm{H}$	$p9, pp15\mathrm{L}$	$p12 - pp14\mathrm{H}$
49	–	$p9, pp15\mathrm{H}$	$p8 + pp15\mathrm{L}$	$p10, pp15\mathrm{H}$	$p9 - pp15\mathrm{L}$
50	–	$p10, pp16\mathrm{L}$	$p9 + pp15\mathrm{H}$	$p11, pp16\mathrm{L}$	$p10 - pp15\mathrm{H}$
51	–	$p11, pp16\mathrm{H}$	$p10 + pp16\mathrm{L}$	$p12, pp16\mathrm{H}$	$p11 - pp16\mathrm{L}$
52	–	$p9, pp17\mathrm{L}$	$p11 + pp16\mathrm{H}$	$p10, pp17\mathrm{L}$	$p12 - pp16\mathrm{H}$
53	–	$p10, pp17\mathrm{H}$	$p9 - pp17\mathrm{L}$	$p11, pp17\mathrm{H}$	$p10 + pp17\mathrm{L}$
54	–	$p11, pp18\mathrm{L}$	$p10 - pp17\mathrm{H}$	$p12, pp18\mathrm{L}$	$p11 + pp17\mathrm{H}$
55	–	$p12, pp18\mathrm{H}$	$p11 - pp18\mathrm{L}$	$p13, pp18\mathrm{H}$	$p12 + pp18\mathrm{L}$
56	–	$p11, pp19\mathrm{L}$	$p12 - pp18\mathrm{H}$	$p13, pp19\mathrm{L}$	$p13 + pp18\mathrm{H}$
57	–	$p12, pp19\mathrm{H}$	$p11 - pp19\mathrm{L}$	$p14, pp19\mathrm{H}$	$p13 + pp19\mathrm{L}$
58	–	$p4, pp20\mathrm{H}$	$p12 - pp19\mathrm{H}$	$p8, pp20\mathrm{H}$	$p14 + pp19\mathrm{H}$
59	–	$p6, pp21\mathrm{H}$	$p4 + pp20\mathrm{H}$	$p8, pp21\mathrm{H}$	$p8 - pp20\mathrm{H}$
60	–	$p8, pp22\mathrm{H}$	$p6 + pp21\mathrm{H}$	–	$p8 - pp21\mathrm{H}$
61	–	$p9, pp23\mathrm{H}$	$p8 - pp22\mathrm{H}$	–	–
62	–	–	$p9 + pp23\mathrm{H}$	–	–
63	–	$p8, pp24\mathrm{H}$	–	$p10, pp24\mathrm{H}$	–
64	–	$p8, pp25\mathrm{H}$	$p8 - pp24\mathrm{H}$	$p12, pp25\mathrm{H}$	$p10 + pp24\mathrm{H}$
65	–	–	$p8 - pp25\mathrm{H}$	–	$p12 + pp25\mathrm{H}$
66	–	–	–	–	–

関連論文

- 矢崎俊志, 阿部公輝, “FFT 多倍長乗算器の VLSI 設計,” 日本応用数理学会論文誌, Vol.15, No.3, pp.385-401, Sep. 2005.
  (第 3 章の内容に関連)

- S. Yazaki and K. Abe ,“VLSI Implementation of Karatsuba Algorithm and Its Evaluation,” Proceedings of The International Workshop on Modern Science and Technology 2006, pp.378-383, May 2006.
  (第 4 章の内容に関連)

- S. Yazaki and K. Abe, “VLSI Design of Iterative Karatsuba Multiplier and Its Evaluation,” Proceedings of The 4th IASTED International Conference on Circuits, Signals, and Systems, San Francisco, pp.313-318, Nov. 2006.
  (第 4 章の内容に関連)

参考論文

- S. Yazaki and K. Abe, “An Optimum Design of FFT Multi-Digit Multiplier and Its VLSI Implementation,” Bulletin of the University of Electro-Communications, Vol.18, No.1 and 2, pp.39-46, Jan. 2006.

- 矢崎俊志, 阿部公輝, “FFT 乗算器の最適化実装,” 電子情報通信学会技術報告 (VLSI 設計技術研究会), Vol.104, No.477, pp.163-168, Dec. 2004.

- 矢崎俊志, 阿部公輝, “高速 Fourier 変換を用いた多倍長乗算器の設計と評価および VLSI への実装,” 電子情報通信学会技術報告 (VLSI 設計技術研究会), Vol.103, No.476, pp.253-258, Nov. 2003.

- 矢崎俊志, 阿部公輝, “高速 Fourier 変換を用いた多倍長乗算器の構成法とハードウェア実装法の検討,” 情報処理学会第 65 回全国大会講演予稿集, Vol.65, No.1, pp105-106, Mar. 2003.

著者略歴

矢崎 俊志 (やざき しゅんじ)

1978 年 6 月 12 日　香川県に生まれる

2000 年 3 月　東京工科大学 工学部 情報工学科 卒業

2002 年 3 月　電気通信大学 大学院 電気通信学研究科 博士前期課程情報工学専攻 修了

2003 年 4 月　電気通信大学 大学院 電気通信学研究科 博士後期課程情報工学専攻 入学