2003－ARC－155 (2)
2003／11／27

[招待論文] 算術演算回路のアルゴリズム

高木 直史†

† 名古屋大学大学院情報科学研究科 〒464-8603 名古屋市千種区不老町
E-mail: †ntakagi@is.nagoya-u.ac.jp

あらまし ディジタル回路の設計においては，その回路の要件に合った優れた算術演算回路を設計することが重要である。本稿では，加算器および乗算器，除算器の構成の基礎となるアルゴリズムを紹介する。
キーワード 算術演算回路，ハードウェアアルゴリズム，加算器，乗算器，除算器

Hardware Algorithms for Arithmetic Circuits

Naofumi TAKAGI†

† Department of Information Engineering, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 464-8603 Japan
E-mail: †ntakagi@is.nagoya-u.ac.jp

Abstract Designing good arithmetic circuits which meet specific requirements is one of the keys in design of digital circuits. In this article, hardware algorithms for addition, multiplication and division are explained.
Key words Arithmetic circuits, hardware algorithms, adder, multiplier, divider.

1. はじめに

回路の自動合成ツールの高機能化により，回路設計者にとって，加算器や乗算器はブラックボックスになりつつある。しかし，より高品質の回路を設計するためには，これらの演算回路の構成法およびその基となる回路のアルゴリズムについて十分に理解しておくことが重要である。算術演算回路の合成ツールの開発者やＩＰの設計者が，演算回路のアルゴリズムを理解しておかなければならないことは言うまでもない。

本稿では，加算器および乗算器，除算器の構成の基礎となるアルゴリズムを紹介する。

2. 加算器のアルゴリズム

n ビット符号なし２進整数の加算について考える。被加数，加数，和をそれぞれ $X = [x_{n-1} \cdots x_0]$, $Y = [y_{n-1} \cdots y_0]$, $S = [s_n s_{n-1} \cdots s_0]$ $(x_i, y_i, s_i \in \{0,1\})$ とする。

2.1 順次桁上げ加算

加算は，下位桁から順に，各桁において，その桁の演算数 x_i, y_i と下位からの桁上げ c_i から，その桁の和 s_i と上位への桁上げ c_{i+1} を計算することにより行える。この一桁分の計算を行う回路を全加算器（ＦＡ）と呼ぶ。

全加算器を直列に接続して構成したのが順次桁上げ加算器である。各全加算器の桁上げ出力が一つ上位の全加算器の桁上げ入力になる。最悪の計算時間（回路の段数）は n に比例する。

2.2 桁上げ飛び越し加算

順次桁上げ加算において，第 i 桁を，下位からの桁上げが通過して上位に伝搬するのは，$p_i (= x_i \oplus y_i)$ が 1 のときである。連続する桁のブロックを考えると，ブロック内のすべての桁で p_i が 1 であれば，下位からの桁上げがブロックを通過して上位に伝搬する。そこで，計算をいくつかのブロックに分け，各ブロックで桁上げ伝搬条件が成立するかどうかを求めておき，条件が成立する場合，下位からの桁上げがブロックを飛び越して上位に伝搬するようにするのが，桁上げ飛び越し加算器である[1]。各ブロック内では，桁上げが順次伝搬する。

第 j 桁からの k 個の桁からなるブロック h の桁上げ伝搬条件は，$P_h = p_j p_{j+1} \cdots p_{j+k-1}$ が 1 になることであり，ブロック h からの桁上げ C_{h+1} は，$C_{h+1} = c_{j+k} + P_h C_h$ となる。ここに c_{j+k} はブロックの最上位の全加算器からの桁上げ，C_h は一つ下位のブロックからの桁上げである。桁上げ伝搬条件は，全ブロックで独立に計算できる。

図１に示すように，桁上げ飛び越し加算器は，順次桁上げ加算器に桁上げ飛び越し回路を付加した構成になる。(p_i は全加算器内で計算されるものとしている。)

ある桁で発生した桁上げがいくつか上位のブロックの桁まで伝搬する場合，桁上げは，発生したブロック内を順次上位へ伝搬し，いくつかのブロックを飛び越し，さらに伝搬先の桁が属するブロック内を順次伝搬する。したがって，桁上げ伝搬に要する時間は，桁上げが二つのブロック内を順次伝搬する時間といくつかのブロックを飛び越す時間の合計になる。よって，回

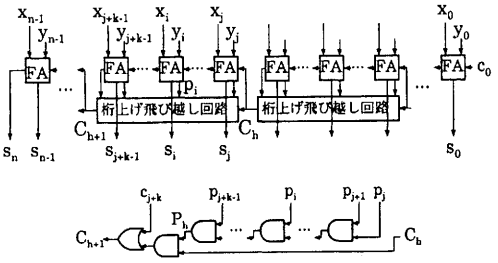

図1　桁上げ飛び越し加算器（上）と桁上げ飛び越し回路（下）

路の計算時間は，ブロックの大きさと個数に依存する。各ブロックの大きさを $\sqrt{n}$ に比例する一定値にすると，ブロックの個数も $\sqrt{n}$ に比例し，計算時間は $\sqrt{n}$ に比例する。最も高速にするには，桁上げがブロック内を順次伝搬する時間を考慮し，中央のブロックほど線形に大きくすればよい。どのようにブロック化しても，回路の素子数は n に比例する。見掛け上，回路の段数は順次桁上げ加算器より若干大きくなるが，桁上げ信号がバイパスするので，計算時間は小さくなる。すなわち，最長経路はフォールスパスになる。

いくつかの連続するブロックをグループ化し，桁上げ飛び越しの機構を多段化して，さらに高速化することができる。

2.3　桁上げ選択加算

順次桁上げ加算において，隣り合う全加算器の間で受け渡される情報は桁上げであり，0 か 1 の二通りしかない。そこで，計算を上位と下位に分け，上位では，下位での計算と並行して，下位からの桁上げが 0 の場合と 1 の場合の二通りを同時に計算しておき，下位からの桁上げが確定した時点で，上位の正しい結果を選択するようにすれば高速化が可能である。この考えに基づくのが，桁上げ選択加算器である [2]。図 2 に示すように，計算をいくつかのブロックに分け，最下位を除く各ブロックで，下位からの桁上げが 0 の場合と 1 の場合の両方について和と桁上げを計算しておき，下位からの桁上げによりいずれかを選択する。選択され確定した桁上げが，次のブロックのセレクタの制御入力になる。

ある桁で発生した桁上げは，発生したブロック内を順次上位へ伝搬し，一つ上位のブロックのセレクタの制御入力になる。以後，より上位のブロックの和および桁上げが次々と確定する。したがって，加算に要する時間は，桁上げが一つのブロック内を順次伝搬する時間とセレクタを何段か通過する時間になる。

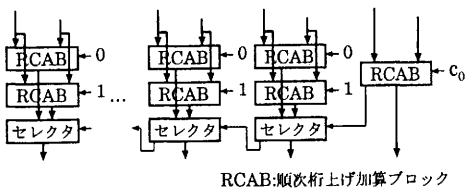

図2　桁上げ選択加算器

よって，回路の計算時間は，ブロックの大きさと個数に依存する。各ブロックの大きさを $\sqrt{n}$ に比例する一定値にすると，ブロックの個数も $\sqrt{n}$ に比例し，計算時間は $\sqrt{n}$ に比例する。最も高速にするには，桁上げがブロック内を順次伝搬する時間を考慮し，ブロックを上位ほど線形に大きくすればよい。どのようにブロック化しても，回路の素子数は n に比例する。

各ブロックで，下位ブロックからの桁上げが 1 の場合の桁上げを計算する代りに，ブロック内での桁上げ伝搬条件を計算しておくことにより，ハードウェア量を削減することができる [3]。

桁上げ選択の考えを再帰的に適用し，多段構成にすることにより計算を高速化できる。この考えを進めていくと木状の構成になり，条件求和加算器になる [4]。

2.4　桁上げ先見加算

加算において，各桁での桁上げは，演算数のそれより下位の桁だけから定まる。したがって，原理的には，演算数が入力された時点ですべての桁で並列に桁上げを計算できる。この考えに基づくのが，桁上げ先見加算器である。

第 i 桁において，$g_i(= x_i \cdot y_i)$ が 1 のとき桁上げが発生し，p_i が 1 のとき下位からの桁上げが上位に伝搬する。すなわち，上位への桁上げは，$c_{i+1} = g_i + p_i c_i$ と表される。これに $c_i = g_{i-1} + p_{i-1}c_{i-1}$，$c_{i-1} = \ldots$ と代入を繰り返すと，$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_0 c_0$ となる。これは，ある桁で上位への桁上げが 1 となるのは，その桁で新たに桁上げが発生する場合と，下位のどこかの桁で発生した桁上げがその桁まで伝搬しさらに上位に伝搬する場合のいずれかであることを示している。すべての桁でこの計算を行えば，すべての桁の桁上げが求まる。しかし，後者の計算は，桁上げの発生箇所が下位の任意の桁である場合について行う必要があり，この計算を各桁で別々に並列に行う純粋な桁上げ先見加算器は，論理素子の入力数（ファンイン）やハードウェア量の観点から，数桁程度のものしか構成できない。

第 j 桁からの k 個の桁からなるブロック h の桁上げ生成条件，すなわち，ブロック内で発生した桁上げがブロックから出力される条件は，$G_h = g_{j+k-1} + p_{j+k-1}g_{j+k-2} + \cdots + p_{j+k-1}p_{j+k-2}\cdots p_{j+1}g_j$ が 1 になることである。また，桁上げ伝搬条件は，P_h が 1 になることである。G_h と P_h の計算を行う回路をブロック桁上げ先見回路という。ブロック内の第 $j+i$ 桁での桁上げは，$c_{j+i+1} = g_{j+i} + p_{j+i}g_{j+i-1} + \cdots + p_{j+i}\cdots p_{j+1}g_j + p_{j+i}\cdots p_{j+1}p_j C_h$ となる。この計算を行う回路を上記の回路と組み合わせたものをブロック桁上げ先見生成回路という。4 桁（$k = 4$）程度のブロックが一般的である。

ブロック桁上げ先見生成回路を直列に接続することにより，加算器を構成できる。しかし，この構成では，桁上げがブロックを順次伝搬するので，計算時間は順次桁上げ加算器に対して定数倍しか改善されない。

桁上げ先見の考え方は，連続する桁のブロックの間でも成り立つ。すなわち，あるブロックで上位への桁上げが 1 となるのは，そのブロック内で桁上げが発生する場合と，下位のどこかのブロックで発生した桁上げがそのブロックまで伝搬しさらに上位に伝搬する場合のいずれかである。したがって，いくつか

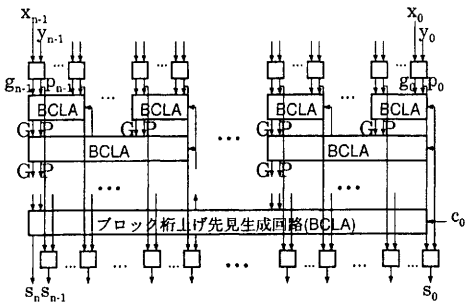

図3　木状構造の桁上げ先見加算器

の連続するブロックのグループを考えると，上記と全く同じ計算により，ブロックの桁上げ生成／伝搬条件から，グループにおける桁上げ生成／伝搬条件を計算できる。また，グループへの桁上げ入力から，グループ内の各ブロック（の最上位桁）での桁上げも上記と同様に計算できる。これは，さらに，グループのグループにも適用できる。したがって，図3に示すように，数桁分のブロック桁上げ先見生成回路を多段に木状に接続することにより，高速の加算器を構成できる。加算器は同一のブロックから構成され，回路の素子数は n に比例する。回路の段数は $\log n$ に比例する。しかし，配線などは複雑になる。

2.5　入力に時間差がある加算

入力の到達時刻がビット位置によって異なる場合には，入力の時間差を利用して，高速で小面積の加算器を構成できる。

入力が最下位ビットから順に，一定の時間間隔で到着する場合は，順次桁上げ加算器，または，ブロック桁上げ先見回路を直列に接続した加算器を用いればよい。到着間隔が全加算器での桁上げの計算時間よりも長い場合は，順次桁上げ加算器で十分である。到着間隔が短い場合は，数ビットを一つのブロックとし，ブロックでの1ビット当りの桁上げ計算時間が到着間隔と等しくなるようにすればよい。入力の最上位ビットの到着後，直ちに加算を完了できる。

入力が最上位ビットから順に，一定の時間間隔で到着する場合は，桁上げ選択加算器を用いればよい。ブロックの大きさが，上位ほど等比数列的に大きくなるようにする。ブロックの大きさの比は，到着間隔と全加算器での桁上げの計算時間の比によって定まる。入力の最下位ビットの到着後，$\log n$ に比例する遅延で加算を完了できる。

2.6　繰り返し加算

いくつかの数を加え合わせる場合，加算が繰り返される。このとき，個々の加算では桁上げを伝搬させずに，桁上げの処理を次の加算に持ち越せば，個々の加算を高速に行える。すなわち，ある加算において各桁で発生した桁上げを上位に伝搬させずに，次の加算において各桁の桁上げ入力として扱う。これを桁上げ保存加算と呼ぶ。和を桁上げと中間和からなる桁上げ保存形で表すことになる。各加算では，前の加算の出力である桁上げと中間和と新たな演算数を加え合わせ，新たな桁上げと中間和を求めることになる。桁上げ保存加算では，各桁で独立に，

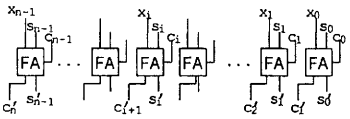

図4　桁上げ保存加算器

全桁で並列に桁上げと中間和を計算できる。桁上げ保存加算器は，図4に示すように，全加算器を並列に独立に並べることにより構成できる。回路の段数は，演算数の桁数に関係なく，全加算器一段分である。

総和が桁上げ保存形で得られるので，最後に，これを通常の2進表現に変換する。この変換は，二つの2進数の通常の加算であり，桁上げの伝搬を伴う。

3.　乗算器のアルゴリズム

n ビット符号なし2進整数の乗算について考える。被乗数と乗数をそれぞれ $X = [x_{n-1} \cdots x_0]$，$Y = [y_{n-1} \cdots y_0]$ とする。ここでは，並列乗算について考える。乗数の各ビット y_j に対して部分積 $X \cdot y_j \cdot 2^j$ を生成し，これらを累算して積を得る。

3.1　配列型乗算

配列型乗算では，部分積を順次，桁上げ保存加算で加え合わせていく[5]。配列型乗算器は，図5に示すように，全加算器を2次元配列状に並べた構成になる。回路の段数は n に比例し，素子数は n^2 に比例する。規則正しいセル配列構造になる。

2ビット Booth の方法[6]を用いて，部分積の個数をおよそ半分にすることにより，高速化が可能である。2ビット Booth の方法では，乗数を桁集合が $\{-2,-1,0,1,2\}$ の4進SD表現にリコードする。乗数を2ビットずつのグループに分け，各グループで y_{2j+1}，y_{2j} と一つ下位のグループの上位ビット y_{2j-1} より，リコードされた乗数桁 $\hat{y}_j$ を $\hat{y}_j := -2 \cdot y_{2j+1} + y_{2j} + y_{2j-1}$ という計算によって求める。この変換は，各グループで独立に行える。部分積の生成において，2倍は1ビットシフト，-1 倍はビット反転で行える。

3.2　累算系列を二本にした配列型乗算

加算は結合則および交換則が成り立つので，これを利用して累算の系列を複数にして，計算を高速化することができる。た

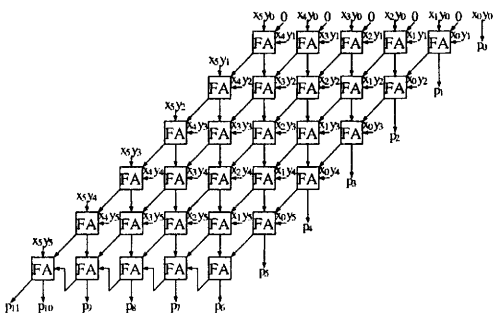

図5　配列型乗算器

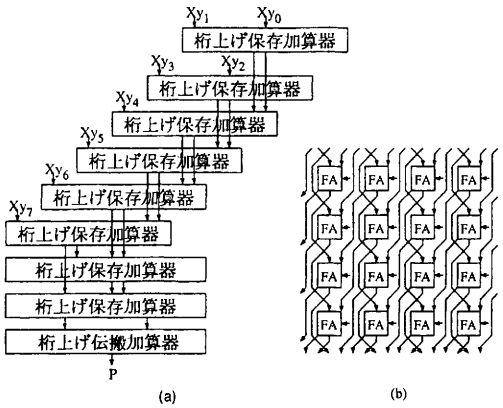

図6　累算系列を二本にした配列型乗算器

とえば，累算の系列を二本に並列化することが考えられる。部分積を，乗数の下位と上位に対応する二つのグループに分け，累算を二つ並列に行い，最後に二つの累算結果を加え合わせれば，配列部での計算時間をおよそ半分にすることができる。この乗算器では，下位の部分積の累算結果が，上位の部分積を累算する配列部を越えて，最終累算のための加算器に伝えられる。このため，一部の配線が他より極端に長くなり，また，下位と上位で配列部のレイアウトが異なる。

配列型乗算器の回路の規則性をできるだけ損なわずに高速化を実現する方法として，累算の二本の系列の一方は奇数段目の桁上げ保存加算器のみを伝わり，他方は偶数段目のみを伝わるようにすることが考えられる[7]。図6(a)にこの乗算器の回路構成を示す。配列部は(b)に示すように，同じパターンの繰り返しになる。

3.3　Wallace 木を用いた乗算

累算の系列を複数にして計算を高速化する考えをさらに進めれば，部分積を木状に加え合わせることになる。

桁上げ保存加算では，和を保存して三つの2進数を二つの2進数に変換していることになる。すなわち，A，B，C という三つの2進数を桁上げ保存加算器に入力すれば，R，S という二つの2進数が得られ，$A+B+C=R+S$ となる。この計算が，ビット長に関係なく，全加算器一段分の計算時間で行える。

n 個の部分積に対して，$\lfloor n/3 \rfloor$ 個の桁上げ保存加算器を用い，上記の計算を並列に行えば，部分積の個数が $\lceil 2n/3 \rceil$ になる。この計算をおよそ $\log_{3/2} n$ 回繰り返せば，部分積が二数になる。最後に，この二数を桁上げ先見加算器などで加え合わせる。これが Wallace 木を用いた乗算である[6]。図7に Wallace 木を用いた乗算器の構成を示す。配列型乗算器と同じく全加算器を基本セルとして構成される。回路の段数は $\log n$ に比例する。素子数は配列型乗算器と大差ない。しかし，配線が複雑で，レイアウトが複雑になる。

Wallace 木を用いた乗算法にも，2ビット Booth の方法を採用できる。しかし，部分積の個数が約半分になっても，桁上げ保存加算の段数は一段ないし二段減るだけである。2ビット Booth の方法の採用にあたっては，計算時間およびハードウェア量について，十分に検討する必要がある。

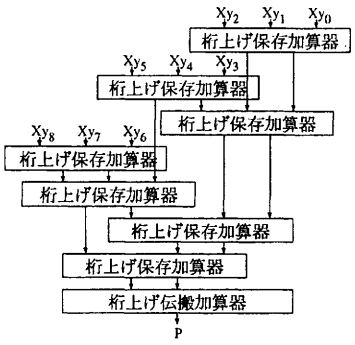

図7　Wallace 木を用いた乗算器

3.4　4-2 加算木を用いた乗算

Wallace 木を用いた乗算法では，基本計算が三数を二数に変換する桁上げ保存加算であるため，レイアウトが複雑になった。桁上げ保存加算を直列に2回行えば，和を保存して四数を二数に変換できる。すなわち，一段目の桁上げ保存加算で四数のうちの三数を加え合わせて二数を得，この二数と残りの一数を二段目で加え合わせて二数を得る。この計算を 4－2 加算と呼ぶ。4－2 加算器は，図8に示すように，全加算器二つからなる基本セルを一列に並べることにより構成できる。

4－2 加算を基本計算として部分積を木状に加え合わせれば，図9に示すように，回路は 4－2 加算器を二分木状に接続した構造になる。この乗算器を 4－2 加算木を用いた乗算器と呼ぶ[8]。回路の段数は，$\log n$ に比例するが，Wallace 木を用いた乗算器に比べ若干増加する。Wallace 木に比べ，回路の規則性に優れ，レイアウトが容易である。

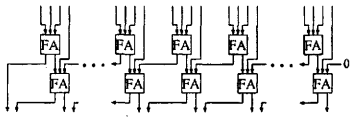

図8　4－2 加算器

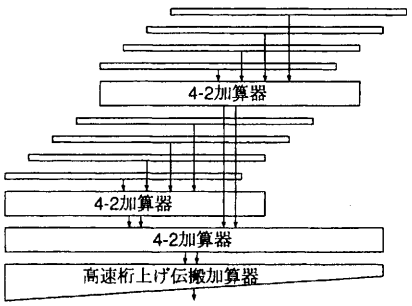

図9　4－2 加算木を用いた乗算器

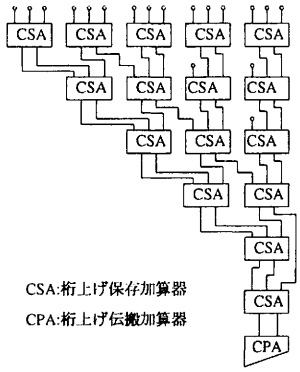

図 10　Overturned-Stairs 加算木を用いた乗算器の構成

3.5　Overturned-Stairs 加算木を用いた乗算

比較的高速で，かつ，規則性の高い回路構造をもつ乗算器として，バランス木を用いた乗算器がある。この乗算器では，部分積をいくつかのグループに分けて累算を並列に行い，さらに，各グループの累算結果を順次累算する。グループの大きさを後段ほど線形に大きくし，各グループでの累算と累算結果の累算の遅延がバランスするようにする。回路の段数は $\sqrt{n}$ に比例し，配列型乗算器に比べ高速である。グループを飛び越す配線は，ビットスライス当たり最大２本になる。

バランス木における累算結果の累算部を改良して，回路段数を若干小さくした乗算器として，図 10 に示す Overturned-Stairs（ＯＴＳ）加算木を用いた乗算器がある [9]。バランス木では，累算結果の累算に系列当たり桁上げ保存加算を二段用いていたのに対し，ＯＴＳ加算木では一段になる。ブロックを飛び越す配線は，ビットスライス当たり最大３本になる。

3.6　冗長２進加算木を用いた乗算

桁上げ保存加算と同様に繰り返し加算を高速化する方法として，冗長２進表現（基数２，桁集合 $\{-1,0,1\}$ の符号付きディジット表現 [10]）の利用がある。二数の加算が桁上げの伝搬なしに高速に行える。冗長２進加算木を用いた乗算 [11] では，部分積を冗長２進数とみなし，冗長２進加算を基本計算として，二分木状に加え合わせる。最後に積を２進表現に変換する。変換に高速の加算器を用いれば，乗算器全体の回路段数は $\log n$ に比例する。$4-2$ 加算木を用いた乗算器と同様の回路構造になり，高速でかつレイアウトが比較的容易である。

4.　除算器のアルゴリズム

n ビット符号なし２進小数の除算について考える。被除数 X と除数 Y はともに正規化されており，$1/2 \leqq X, Y < 1$ を満たすものとする。$|X/Y - Z| < 2^{-n}$ を満たす商 Z を小数点以下 n ビット目まで求めるものとする。

4.1　減算シフト型除算

減算シフト型除算では，中間結果と剰余に関する等式 $R_j = r^j(X - Y \cdot Q_j)$ が基となる。r は商の基数，$Q_j = \sum_{i=1}^{j} q_i r^{-i}$ は商の小数点以下 j 桁目まで（中間結果），R_j は商を r 進で小数点以下 j 桁目まで求めた時の剰余の r^j 倍である。この等式から，漸化式，$R_{j+1} := rR_j - q_{j+1}Y$ を得る。また，$Q_{j+1} := Q_j + q_{j+1}r^{-j-1}$ である。これらに従い，商を上位から一桁ずつ求めていく。商の基数 r をいくつにするか（Z の基数である 2 とは異なってもよい），q_{j+1} をどのような桁集合から選ぶか，R_j の表現に冗長表現を用いるかどうかなどにより種々のバリエーションがある [12]。

基数が 2（$r = 2$）の場合は，$R_0 := X$，$Q_0 := 0$ とし，$R_{j+1} := 2R_j - q_{j+1}Y$，$Q_{j+1} := Q_j + q_{j+1}2^{-j-1}$ という漸化式に従って計算を行う。ここでは，説明を簡単にするため $X < Y$ と仮定する。

最も基本的な計算法は，回復型除算である。$q_{j+1} \in \{0,1\}$ とし，$2R_j - Y$ が非負なら q_{j+1} を 1，負なら 0 とする。すなわち，まず $R'_{j+1} := 2R_j - Y$ を計算し，$R'_{j+1} \geqq 0$ なら $q_{j+1} := 1$，$R_{j+1} := R'_{j+1}$ とし，$R'_{j+1} < 0$ なら $q_{j+1} := 0$，$R_{j+1} := R'_{j+1} + Y$ とする。常に $0 \leqq R_j < Y$ が成り立つ。途中で $R_{j+1} = 0$ になれば，計算を終了するようにしてもよい。Q_n は２進表現になり，そのままで商 Z になる。

非回復型除算では，$2R_j - q_{j+1}Y$ が負になっても，これをそのまま R_{j+1} とする。回復型除算に比べ，加減算の回数が少なくなる。$q_{j+1} \in \{-1,1\}$ とし，$2R_j$ が非負なら q_{j+1} を 1，負なら -1 とする。常に $-Y \leqq R_j < Y$ が成り立つ。商を通常の２進表現に変換する必要があるが，Z の j 桁目 z_j は q_{j+1} が 1 なら 1，-1 なら 0 となり，特に計算の必要はない。商の変換を漸化式に組み込むこともできる。$z_1 := 1$，$R_1 := 2X - Y$，$R_2 := 2R_1 - Y$ とし，$j \geqq 2$ について，$2R_j \geqq 0$ なら $z_j := 1$，$R_{j+1} := 2R_j - Y$，$2R_j < 0$ なら $z_j := 0$，$R_{j+1} := 2R_j + Y$ とする。

Sweeney，Robertson，Tocher により同時期に独立に考案されたＳＲＴ除算 [13] では，$q_{j+1} \in \{-1,0,1\}$ とし，$q_{j+1} = 0$ のとき R_j の計算に加減算を不要にしている。常に $-Y \leqq R_j < Y$ が成り立つように q_{j+1} を選ぶ。そのためには，$2R_j < 0$ なら $q_{j+1} := -1$，$R_{j+1} := 2R_j + Y$，$-Y \leqq 2R_j < Y$ なら $q_{j+1} := 0$，$R_{j+1} := 2R_j$，$0 \leqq 2R_j$ なら $q_{j+1} := 1$，$R_{j+1} := 2R_j - Y$ とすればよい。すなわち，$-Y \leqq 2R_j < 0$ のときは，q_{j+1} を -1 としても 0 としてもよく，$0 \leqq 2R_j < Y$ のときは，q_{j+1} を 0 としても 1 としてもよい。そこで，$2R_j < -1/2$ なら $q_{j+1} := -1$，$-1/2 \leqq 2R_j < 1/2$ なら $q_{j+1} := 0$，$1/2 \leqq 2R_j$ なら $q_{j+1} := 1$ とする。すなわち，$2R_j$ と $\pm 1/2$ との比較により q_{j+1} を決定する。

減算シフト型除算では，$R_{j+1} := 2R_j - q_{j+1}Y$ という計算を繰り返す。この計算を高速に行うには，部分剰余 R_j を桁上げ保存形で表し，加減算を桁上げの伝搬なしに行うことが有効である。しかし，桁上げ保存形では，正確な正負判定や大小比較のためには最悪の場合全桁を調べる必要がある。したがって，商の選択に正確な正負判定や大小比較が不要になるように工夫する必要がある。上述のように，商の桁集合を $\{-1,0,1\}$ とすれば，$-Y \leqq 2R_j < 0$ のときは，q_{j+1} を -1 としても 0 としてもよく，$0 \leqq 2R_j < Y$ のときは，q_{j+1} を 0 としても 1 としてもよい。したがって，商の選択において，$2R_j$ のおおまかな

値がわかればよい。すなわち，商の選択の隣合う領域に重なりを設けることにより，商の選択において正確な大小比較を不要にすることができる。桁上げ保存形で表された $2R_j$ の小数点以下1桁目まで（符号桁を含む上位3桁）の値を $2\hat{R}_j$ として，$2\hat{R}_j \leqq -1$ ならば $q_{j+1} := -1$，$2\hat{R}_j = -1/2$ ならば $q_{j+1} := 0$，$2\hat{R}_j \geqq 0$ ならば $q_{j+1} := 1$ とする。$2\hat{R}_j \leqq 2R_j < 2\hat{R}_j + 1$ であるから，q_{j+1} として -1 が選択されたときは $2R_j < 0$，0 のときは $-Y \leqq -1/2 \leqq 2R_j < 1/2 \leqq Y$，1 のときは $2R_j \geqq 0$ が保証される。これにより，1ステップにかかる計算時間（回路の段数）は n に無関係な一定値になる。桁上げ保存形の代りに冗長2進表現を用いることもできる[14]。

商の桁集合が $\{-1, 0, 1\}$ の場合は，商を通常の2進表現に変換する必要がある。冗長2進表現の商がすべて求まってから2進表現に変換するようにすると，変換に要する時間が計算時間に上積みされることになる。そこで，この変換を商の選択と並行して 'on-the-fly' に行う手法が提案されている[15]。

最近のマイクロプロセッサで減算シフト型除算の専用回路をもつものは，基数が4，商の桁集合が $\{-2, -1, 0, 1, 2\}$ で，部分剰余の表現に桁上げ保存形を用いたアルゴリズムを採用し，1クロックで1ステップ分の計算を行うものが多い。

4.2 乗算型除算

乗算型除算法は，乗算の繰り返しにより除算を行うもので，高速乗算器をもつシステムで用いられている。Newton 法に基づくものや Goldschmidt 法がある。

Newton 法に基づく除算法では，まず，Newton 法により除数 Y の逆数を求める。$1/Y$ の粗い近似値 P を初期近似値として用い，$U_0 := P$ とし，$U_{i+1} := U_i \cdot (2 - U_i \cdot Y)$ という計算を繰り返す。1回の繰り返しで，解の精度の桁数が2倍になり，二次の収束をする。1回の繰り返しに，乗算が2回必要である。$2 - (U_i \cdot Y)$ は $U_i \cdot Y$ の補数を取ればよい。必要な精度の逆数が求まれば，これに X を乗じて商を得る。

Goldschmidt の除算法では，分数の分母と分子に同じ数を乗じても値が変わらないことを利用する。$X/Y = (X \cdot D_0 \cdot D_1 \cdot D_2 \cdots)/(Y \cdot D_0 \cdot D_1 \cdot D_2 \cdots)$ であり，分母が 1 に近付くように D_i を定めると，分子は X/Y に近付く。具体的には，$X_0 := X$，$Y_0 := Y$ とし，$D_i := 2 - Y_i$，$X_{i+1} := X_i \cdot D_i$，$Y_{i+1} := Y_i \cdot D_i$ という計算を繰り返す。1回の繰り返しに，乗算が2回必要である。$2 - Y_i$ は Y_i の補数を取ればよい。二次の収束をする。計算を速めるため，D_0 として $1/Y$ の粗い近似値 P を用いる。必要な演算は，Newton 法に基づく除算法とほぼ同じであるが，1回の繰り返しにおける二つの乗算の独立性が高く，パイプライン処理などにより複数の乗算が高速に実行できる場合は，より有利である。

5. おわりに

算術演算回路のアルゴリズムについて，さらに詳しく知りたい方は，算術演算全般については[16]〜[19]など，減算シフト型の除算および開平については[12]，三角関数などの初等関数計算については[20]などをお読み頂きたい。3Dベクトルのユークリッドノルム計算など，より複雑な算術演算のためのハードウェアアルゴリズムについては，[21]〜[23] などを参照頂きたい。本分野の新しいアイデアは，隔年開催の IEEE Symposium on Computer Arithmetic で発表されることが多い。

文　献

[1] M. Lehman and N. Burla: 'Skip techniques for high-speed carry propagation in binary arithmetic units', IRE Trans. Elec. Comput., vol. EC-10, pp. 691–698, Dec. 1961.

[2] O. J. Bedrij: 'Carry-select adder,' IRE Trans. Elec. Comput., vol. EC-11, pp. 340–346, June 1962.

[3] A. Tyagi: 'A reduced-area scheme for carry-select adders,' IEEE Trans. Comput. vol. 42, pp. 1163–1170, Oct. 1993.

[4] J. Sklansky: 'Conditional sum addition logic,' IRE Trans. Elec. Comput., vol. EC-9, pp. 226-231, June 1960.

[5] C. R. Baugh and B. A. Wooly: 'A two's complement parallel array multiplier', IEEE Trans. Computers, vol. C-22, no. 12, pp. 1045–1047, Dec. 1973.

[6] C. S. Wallace: 'A suggestion for a fast multiplier', IEEE Trans. Elec. Comput., vol. EC-13, no. 1, pp. 14–17, Feb. 1964.

[7] J. Iwamura et al.: 'A 16-bit CMOS/SOS multiplier-accumulator', Proc. IEEE Intnl. Conf. on Circuits and Computers 1982, 12.3, Sep. 1982.

[8] J. E. Vuillemin: 'A very fast multiplication algorithm for VLSI implemantation', Integration, VLSI journal, vol. 1, no. 1, pp. 39–52, Apr. 1983.

[9] Z.-J. Mou and F. Jutand: 'Overturned Stairs Adder Trees and Multiplier Design', IEEE Trans. Comput., vol. 41, no. 8, pp. 940–948, Aug. 1992.

[10] A. Avizienis: 'Signed-digit number representations for fast parallel arithmetic,' IRE Trans. Elec. Comput., vol. EC-10, pp. 389–400, Sep. 1961.

[11] N. Takagi, H. Yasuura and S. Yajima: 'High-speed VLSI multiplication algorithm with a redundant binary addition tree', IEEE Trans. Comput., vol. C-34, no. 9, pp. 789–796, Sep. 1985.

[12] M. D. Ercegovac and T. Lang: "Division and square root – Digit-recurrence algorithms and implementations", Kluwer Academic Publishers, 1994.

[13] J. E. Robertson: 'A new class of digital division methods', IRE Trans. Elec. Comput., vol. EC-7, pp. 218–222, 1958.

[14] 高木，安浦，矢島：「冗長2進表現を利用したVLSI向き高速除算器」，電子通信学会論文誌，vol. J67-D, no. 4, pp. 450–457, Apr. 1984.

[15] M. D. Ercegovac and T. Lang: 'On-the-fly conversion of redundant into conventional representations', IEEE Trans. Comput., vol. C-36, no. 7, pp. 895–897, July 1987.

[16] K. Hwang: "Computer Arithmetic – Principles, Architecture, and Design", John Wiley & Sons, 1979.
堀越監訳：「コンピュータの高速演算方式」，近代科学社，1980.

[17] A. Omondi: "Computer Arithmetic Systems – Algorithms, Architecture and Implementations", Prentice Hall, 1994.

[18] I. Koren: "Computer Arithmetic Algorithms, 2nd Edithion", A K Peters, 2002.

[19] 高木：「連載講座：算術演算回路のアルゴリズム」，情報処理，vol. 37, no. 1〜no. 5, Jan.〜May 1996.

[20] J.-M. Muller: "Elementary Functions – Algorithms and Implementation", Birkhauser, 1997.

[21] N. Takagi and S. Kuwahara: 'A VLSI algorithm for computing the Euclidean norm of a 3D vector', IEEE Trans. Comput., vol. 49, no. 10, pp. 1074–1082, Oct. 2000.

[22] N. Takagi: 'A digit-recurrence algorithm for cube rooting', IEICE Trans. Fundamentals, vol. E84-A, no. 5, pp. 1309–1314, May 2001.

[23] N. Takagi, D. Matsuoka and K. Takagi: 'Digit-recurrence algorithm for computing reciprocal square root', IEICE Trans. Fundamentals, vol. E86-A, no. 1, pp. 221–228, Jan 2003.