卒 業 研 究 報 告

題　　目

乗算器に関する基本検討

指 導 教 員

矢野　政顯　教授

報 告 者

中村　基継

平成 13 年 2 月 9 日

高知工科大学 電子・光システム工学科

目次

第 1 章　はじめに

今日，爆発的な勢いで普及している携帯電話をはじめ，無線通信やオーディオ・ビデオ処理などのデジタル信号処理技術の分野で DSP（Digital Signal Processor）は使用されている．DSP はデジタル信号処理に特化したマイクロプロセッサの一つで，乗算器と加算器を実装することで，繰り返しの数値演算を効率よく行うよう，ある特定用途向けに設計されたものである．乗算を頻繁に行う計算では，ソフトウェアによる処理を行うと処理速度が遅くなるため，どのような乗算器を使用するか，つまり乗算器の基本性能そのものが直接，システム全体の処理性能に大きく影響をあたえる．そのため，様々な乗算アルゴリズムが今日まで考案されている．本論文では，乗算器にはどのような構造をしたものがあるのかを把握することを目的とし，調査してまとめたものである．

第 2 章　乗算器の基本構造

2.1　乗算アルゴリズム

2.1.1　乗算アルゴリズムの第 1 バージョン

この方式の乗算器では，k ビットの乗数レジスタと 2k ビットの被乗数レジスタ，積レジスタおよび加算器，そして乗算制御部を必要とする（図 2.1）．ただし，ここでは乗算制御部は記述していない．動作手順は次のようになる．乗数 Y は k ビットの乗数レジスタに格納され，2k ビットの積レジスタには初期値として 0 を設定する．手計算による筆算と同様，被乗数 X を乗算ステップごとに 1 ビットずつ左シフトする．したがって， k ステップ後には k ビットの被乗数 X は左へ k 桁ずらされている状態になるので，被乗数レジスタには 2k ビットが必要となる．被乗数レジスタの初期設定では，右半分に k ビットの被乗数 X を格納し，左半分に 0 を格納する．このレジスタをステップごとに 1 ビット左シフトすることで，2k ビットの積レジスタに加算される部分積の桁位置を合わせる．

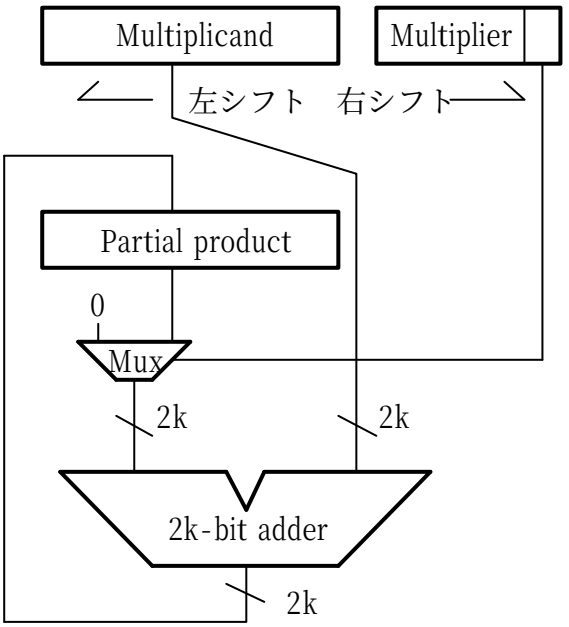


図 2.1　乗算アルゴリズムの第 1 バージョンの回路図

図 2.2 に示すフローチャートのように，ビットごとに 3 つの基本的なステップを繰り返す．ステップ 1 では乗数 Y の最下位ビット（LSB）の値に応じて，被乗数を積レジスタに加えるかどうかを判定する．ステップ 2 の左シフトは部分積を左へずらす働きをする．また，ステップ 3 の右シフトによって，次の処理サイクルで対象となる乗数 Y の次のビットが得られる．この 3 ステップの処理サイクルを k 回繰り返すことで最終的に積が得られる．

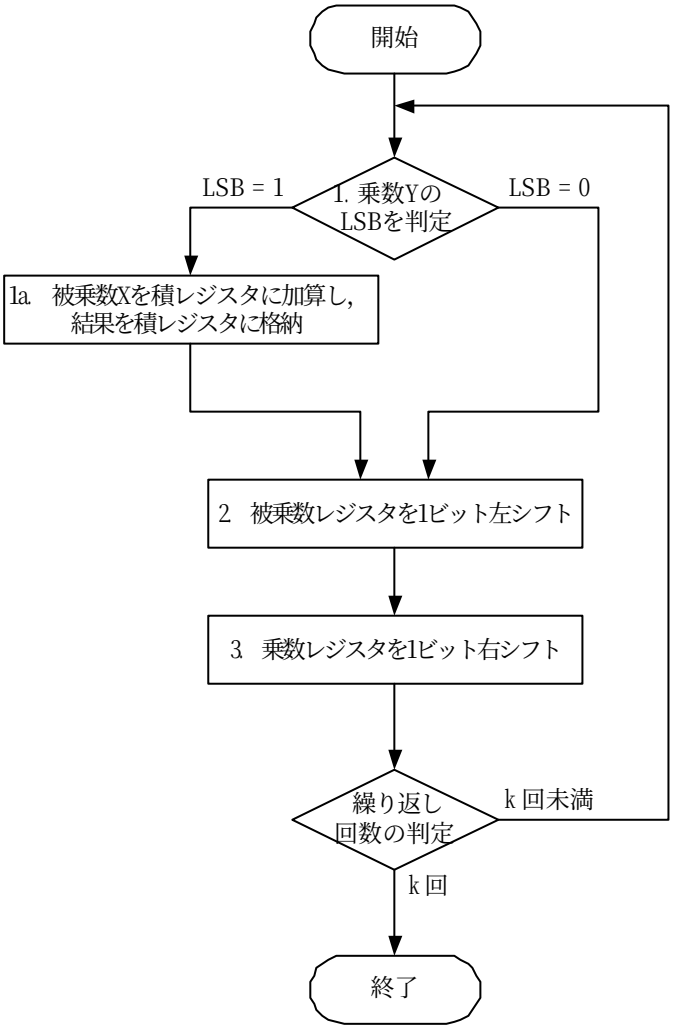


図 2.2　乗算アルゴリズムの第 1 バージョンのフローチャート

2.1.2　乗算アルゴリズムの第 2 バージョン

乗算アルゴリズムの第 1 バージョンでは，被乗数 X を左シフトし，空いた位置には 0 を挿入していたので，積の LSB が設定された後は，被乗数 X の影響を受けることはなかった．そこで被乗数 X を左シフトさせるのではなく，積を右シフトさせることで，被乗数 X を固定することが可能となり，さらに部分積に加算する被乗数 X は k ビットであるので，加算器も k ビットで済む．したがって，図 2.3 に示すように，被乗数レジスタと加算器のビット長を 50 パーセント削減できる．

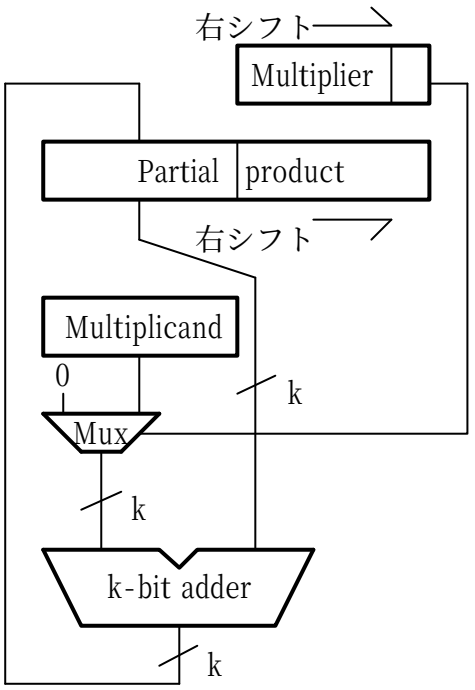


図 2.3　乗算アルゴリズムの第 2 バージョンの回路図

これらから導き出される乗算アルゴリズムは図 2.4 に示すとおりである．このアルゴリズムでは，演算前に k ビットの被乗数レジスタと乗数レジスタそれぞれの値を設定し，2k ビットの積レジスタには初期値として 0 を設定する．このアルゴリズムで得られる和は k ビットであり，加算ステップ中には 2k ビットの積レジスタの左半分だけが変更される．

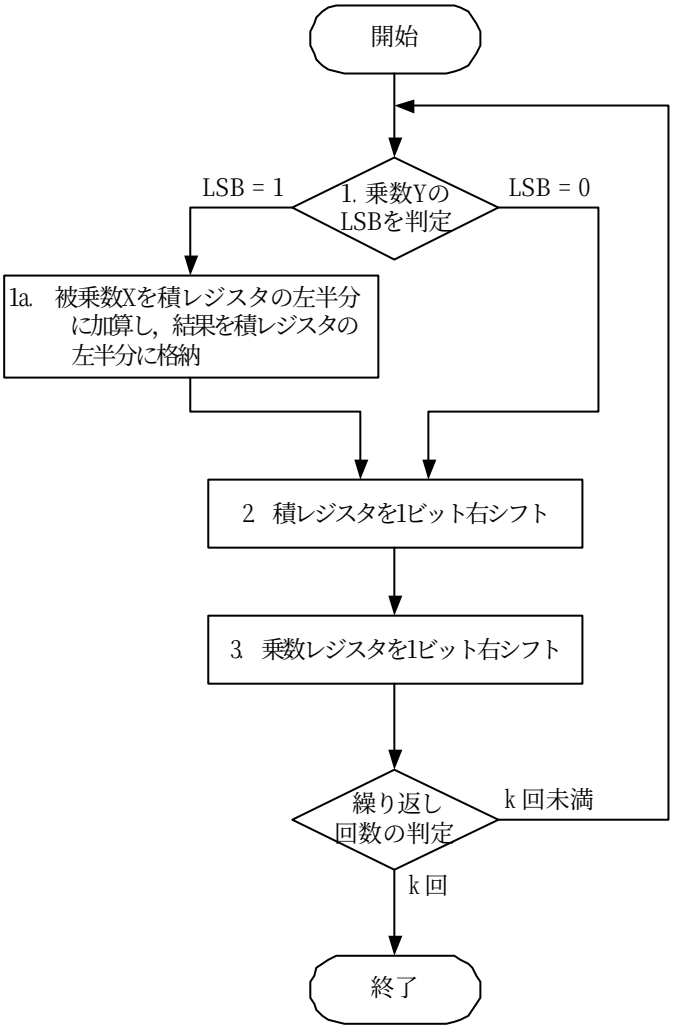


図 2.4　乗算アルゴリズムの第 2 バージョンのフローチャート

2.1.3　乗算アルゴリズムの第 3 バージョン

この方法では積レジスタの右半分を乗算レジスタとすることで，レジスタの数を必要最小限に抑えたものである．その回路を図 2.5 に示す．ここでは 2k ビットの積レジスタの LSB を対象に，乗数 Y が 0 か否かの判断を行う．このアルゴリズムでは，演算前に積レジスタの右半分に乗数 Y を設定し，左半分に 0 を初期値として設定する．計算手順を図 2.6 に記述する．

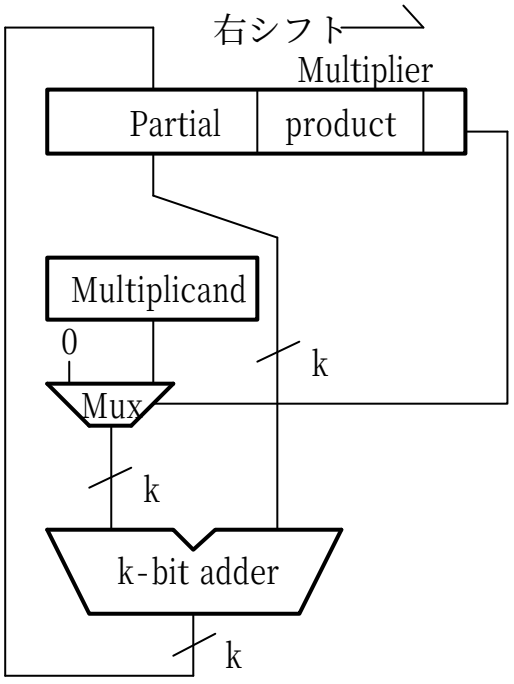


図 2.5　乗算アルゴリズムの第 3 バージョンの回路図

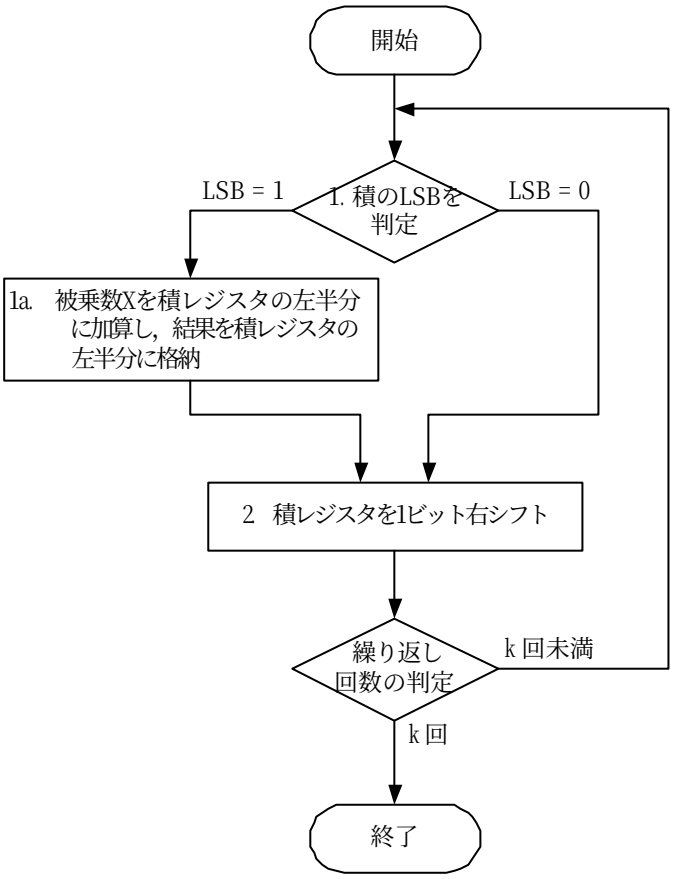


図 2.6　乗算アルゴリズムの第 3 バージョンのフローチャート

2.2　ハードウェア構成

乗算器は部分積生成，部分積の加算，被乗数，乗数，部分積のシフトなどの回路が必要である．この節では加算器とシフタについて説明する．

2.2.1　並列加算器

- リプルキャリー加算器（Ripple Carry Adder）

全加算器の桁上げをカスケード接続したもので, n ビット並列加算器のような回路に使用するとき適用される．非常に単純な構成のためよく使用されるが，桁上げの最下位ビット（LSB）から最上位ビットへの桁上げ信号の伝播遅延がクリティカルパスとなるため，高速回路には向いていない．図 2.7 に 4 ビットリプルキャリー加算器を示す．これは，n を段数，T_c を 1 段当たりの桁上げ遅延，そして T_a を全加算時間としたときに，加算の遅延は $T_a = n \cdot T_c$ となる．なお，図 2.8 に半加算器および全加算器を記す．

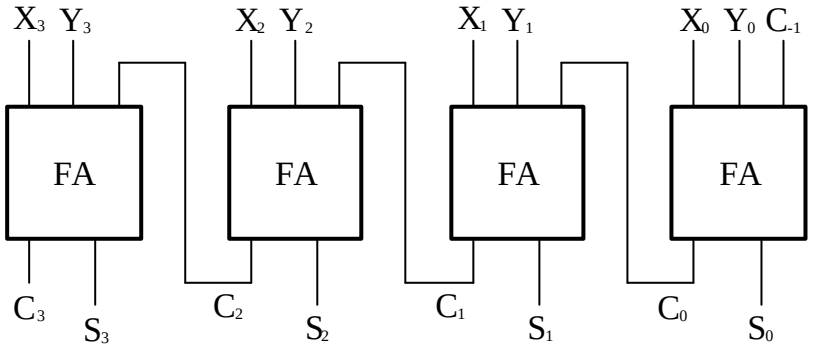


図 2.7　リプルキャリー加算器

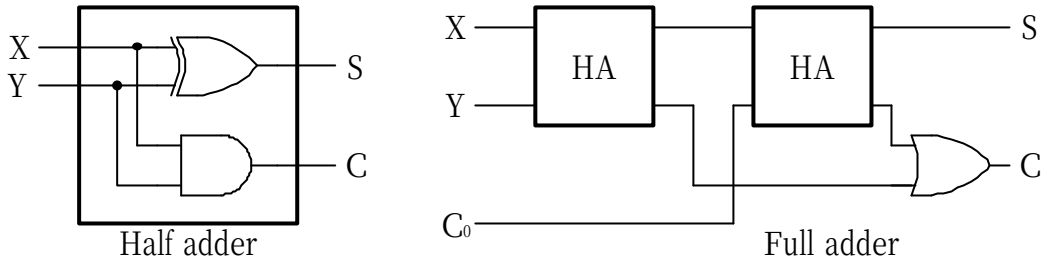


図 2.8　半加算器および全加算器

- **桁上げ先見加算器**（Carry Look-ahead Adder）

加算器の伝播遅延を短くする手法の一つ. 桁数が 16 桁近くにもなってくると，個々の全加算器の工夫に頼るだけでは高速化に限界があるため，16 ビットアダー全体で高速化を考える必要がある．加算器の演算速度は桁上げ信号の伝播速度により規定を受けるため，下位ビットからの桁上げ信号を持たずに個々の桁上げ信号が決定できれば高速化が図れる．ここで，下位桁からの桁上げ信号と個々の桁上げ信号の関係について述べる：

入力 X，Y が共に“ 1 ”のときは下の桁からの桁上げ信号 C_{i-1} の値にかかわらず桁上げ信号 C_i が発生し，X，Y のどちらかが“ 1 ”のときは C_{i-1} の値で C_i が決定することがわかる．この関係を論理式で表すと，次のようになる．

$$C_i = G_i + P_i \cdot C_{i-1} \quad \text{ただし} \quad G_i = X_i \cdot Y_i, \quad P_i = X_i \oplus Y_i$$

ここで，G_i は桁上げ生成信号，P_i は桁上げ伝播信号を指している．機能的には半加算器の C および S と同じである．各桁の桁上げ信号 C_i を表すと

$$C_0 = G_0 + P_0 \cdot C_{-1}$$
$$C_1 = G_1 + P_1 \cdot C_0 = G_1 + G_0 \cdot P_1 + P_0 \cdot P_1 \cdot C_{-1}$$
$$C_2 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + P_0 \cdot P_1 \cdot P_2 \cdot C_{-1}$$
$$C_3 = C_3 + C_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + P_0 \cdot P_1 \cdot P_2 \cdot P_3 \cdot C_{-1}$$
$$\cdots\cdots$$
$$C_n = G_n + G_{n-1} \cdot P_n + G_{n-2} \cdot P_{n-1} \cdot P_n + \ldots + G_0 \cdot P_1 \cdot P_2 \cdot \ldots \cdot P_n + P_0 \cdot P_1 \cdot P_2 \cdot \ldots \cdot C_{-1}$$

となり，すべての桁の桁上げ信号は自桁より下位の入力値と最下位の桁上げ信号だけで生成できる．図 2.9 に 4 ビット桁上げ先見方式の回路図を示す．

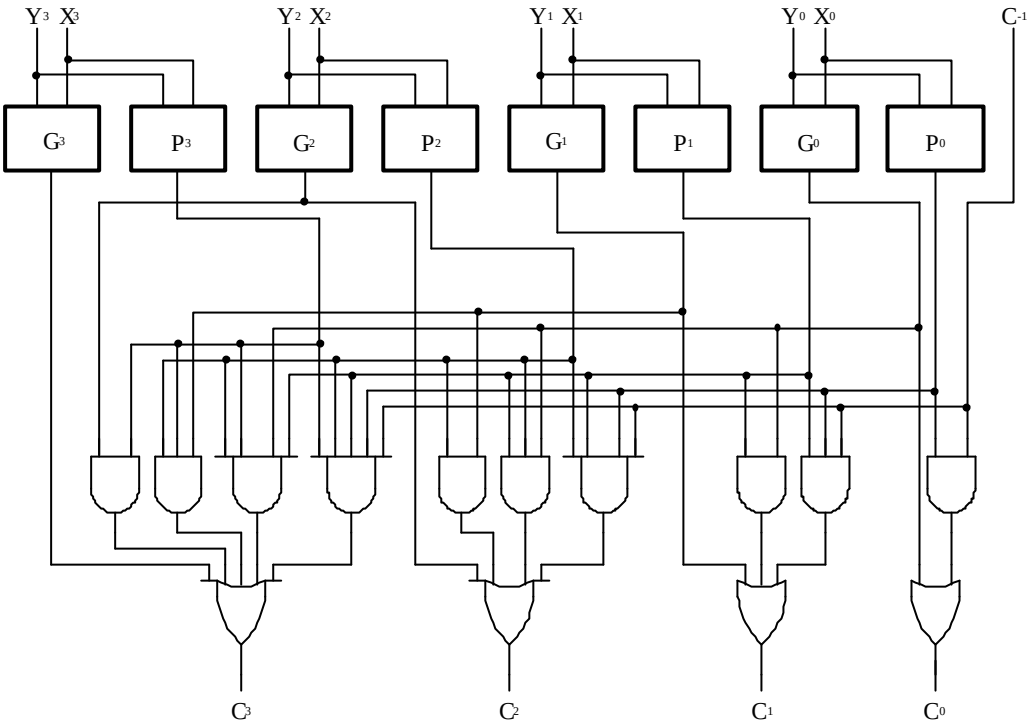


図 2.9　4 ビット桁上げ先見回路

この回路図からもわかるように，先見する桁数が増加するにしたがって回路が複雑化し，素子数も著しく増加するばかりでなく，ゲートのファンアウトが増加し，高速化の効果が得にくくなる．実際に実装する場合，ハードウェア量や効率を考慮してすべての桁上げ信号を CLA で生成することはなく，例えば 4 ビットを 1 つのブロックとしてその単位で CLA を用いて桁上げ信号を伝播させ，ブロック内はリプルで桁上げ信号を伝えるという方式をとる場合が多い．この方式をブロック CLA と呼び，更に桁数が 32 ビット程度まで大きくなると CLA を 2 重にして桁上げ信号の伝播の高速化を図る．

- **桁上げ保存加算器**（Carry Save Adder）

桁上げ保存加算器（CSA）**は，加算を繰り返して実行し，結果を累計していく場合には，桁上げは次の加算の際に上位の桁に渡せばよく，そうすることによって桁上げに要する時間を節約する方法. n 個の全加算器を単純に並列に並べ, ビット間での桁上げは行わず，各ビットからの桁上げをそのまま出力させたものである. したがって, 入力から出力への伝播はビット数によらず一定であり, リプルキャリー加算器より構造上はるかに高速である．図** 2.10 **に** 4 **ビット桁上げ保存加算器を示す．これは乗算を加算器の繰り返し使用で実行する場合と, 乗算を全加算器セルの** 2 **次元配列（アレイ）によって実行する場合によく使用される方法である.**

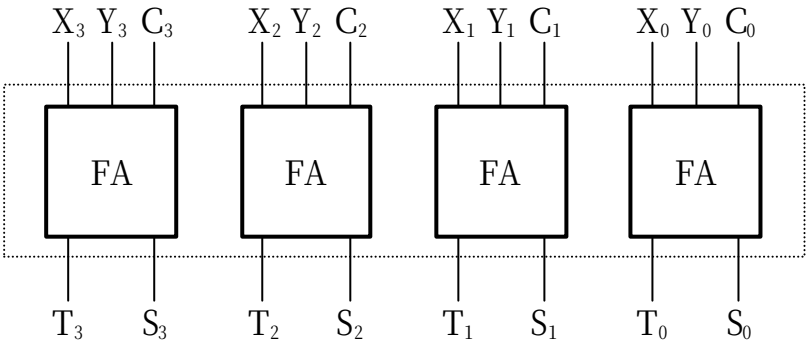


図 2.10　4 **ビット桁上げ保存加算器**

2.2.2　シフトレジスタ

- **直列入力シフトレジスタ**

図 2.11 は，D フリップフロップを用いたシフトレジスタである．n ビットのレジスタ i＋1 の出力を，ビット i の入力に接続することで，保持している値を 1 ビットだけ下位へシフトした値が次の状態となる. n ビットのデータを下位ビットから順番に 1 ビットずつ SI に入力して n 回シフトすると，データをシフトレジスタの出力 $Q_{n-1:0}$ から取り出すことができる．この直列－並列変換は，伝送路を 1 ビットずつ直列に送られてくるデータの受信のための基本的な操作である.

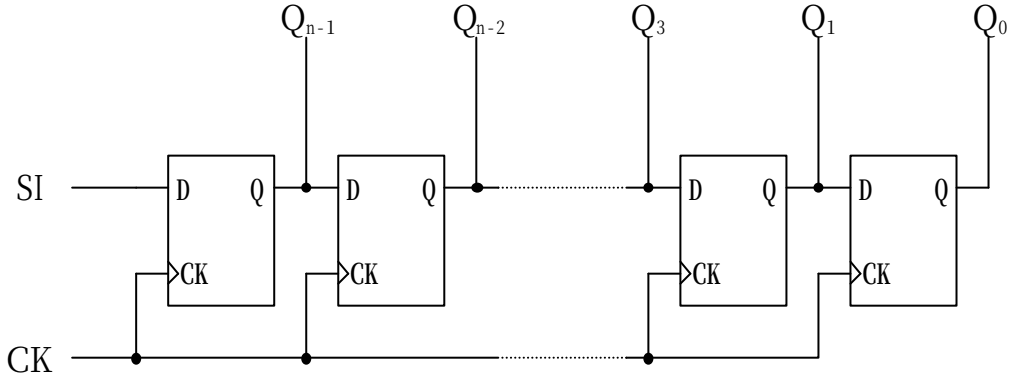


図 2.11　直列入力シフトレジスタ

- **並列入力シフトレジスタ**

図 2.12 のシフトレジスタは各ビットへのデータ入力 $D_{n-1:0}$ を設けることで，シフトの際の入力と，2-to-1 セレクタで選択することにより，制御信号 $\overline{L}$ が 0 のときデータをシフトレジスタにロードする機能を付加することができる．これにより，送信するデータをロードし，それを n 回シフトしながら各ビットを Q_0 から取り出して直列に送信する，並列－直列変換を可能とする.

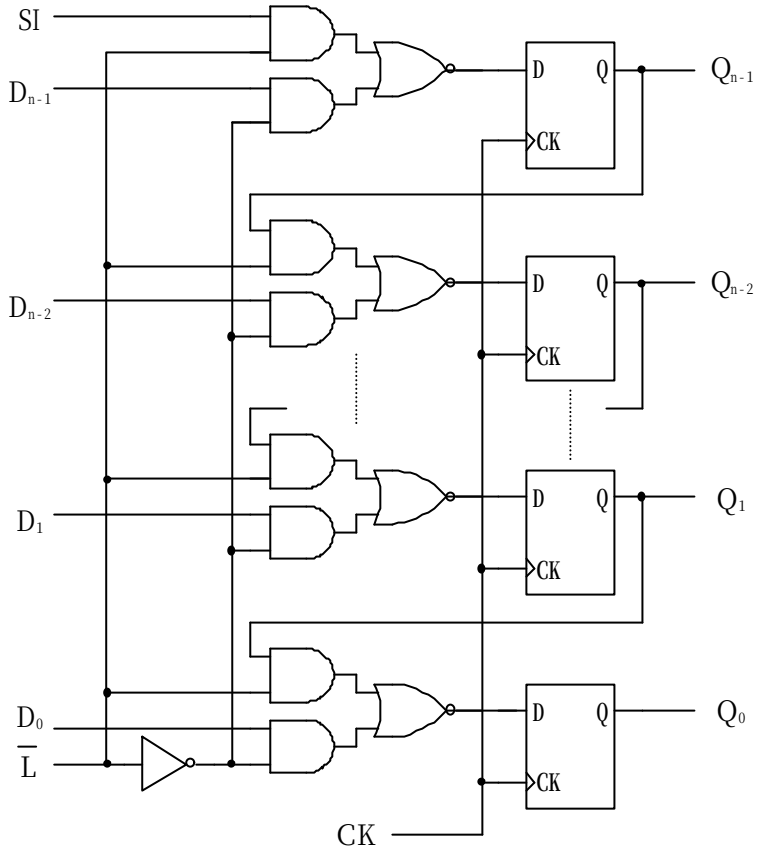


図 2.12　並列入力シフトレジスタ

2.3　Booth アルゴリズム

n ビットの乗算を n/2 クロックで行う最も簡単な方法として，乗数 Y を 2 ビットずつ処理し，その値に応じた被乗数 X の倍数を生成し，その部分積に加えて 2 ビットずつシフトするものが考えられる．すなわち n が偶数である時

$$XY = \sum_{i=0}^{n-1} Xy_i 2^i = \sum_{i=0}^{n/2-1} X(2y_{2i+1} + y_{2i})2^{2i} \qquad (2.1)$$

であるので，$2y_{2i+1}+y_{2i}$ の値に応じて被乗数 X の 0，1，2，3 倍の倍数を用意しておけばよい．しかし，問題は 3 倍の生成にはシフトと加算が必要であり，演算開始前に準備しておくにしても，そのためのレジスタが必要となる．また，この方式を拡張して k ビットずつまとめて処理する場合には，0 から 2^k-1 まで

の 2^k 種類の倍数が必要である．

これに対して Booth 法では，2 ビットずつまとめて処理する場合には被乗数 X の 0，±1，±2 の倍数のみがあればよく，比較的簡単な回路で実現できる．また 2 の補数表現の数を補正なしに乗算できることも特徴の 1 つである．Booth 法の基本は，2 の補数表現の乗数 Y を

$$\begin{aligned} Y &= -Y^S + Y^V = -y_{n-1}2^{n-1} + \sum_{i=0}^{n-2} y_i 2^i \\ &= -y_{n-1}2^{n-1} + 2\sum_{i=0}^{n-2} y_i 2^i - \sum_{i=0}^{n-2} y_i 2^i \\ &= -\sum_{i=0}^{n-1} y_i 2^i + 2\sum_{i=0}^{n-2} y_i 2^i \\ &= -\sum_{i=0}^{n-1} y_i 2^i + \sum_{i=0}^{n-1} y_{i-1} 2^i = \sum_{i=0}^{n-1} (-y_i + y_{i-1})2^i \end{aligned} \tag{2.2}$$

と変形することにある（ただし $y_{-1} \equiv 0$）．式 (2.2)の各項は 0 か±1 であるので，乗数 Y を 1 ビットずつ右にシフトしながら最下位の 2 ビットを検査し，その値に応じて加減算を行えば 2 の補数表現の乗算を補正なしに実行できる．nが偶数の時，式(2.2)を 2 項ずつまとめると

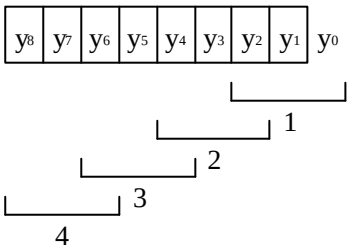
$$\begin{aligned} Y &= \sum_{i=0}^{n-1} (-y_i + y_{i-1})2^i \\ &= \sum_{i=0}^{n/2-1} (2(-y_{2i+1} + y_{2i}) + (-y_{2i} + y_{2i-1}))2^{2i} \\ &= \sum_{i=0}^{n/2-1} (-2y_{2i+1} + y_{2i} + y_{2i-1})2^{2i} \end{aligned} \tag{2.3}$$

となり，各項の値は表 2.1 に示すように 0，±1，±2 のいずれかとなる．したがって乗数 Y の最下位に $y_{-1} = 0$ を付加し，それを 2 ビットずつ右にシフトしながら最下位の 3 ビットをスキャンして，その値に応じて被乗数 X の 1 ビット左シフトを伴う加減算を行えば，n/2 クロックの乗算回路が構成できる．図 2.13 に 2 次の Booth アルゴリズムによる乗算を示す．

表 2.1　2 次の Booth アルゴリズム

y_{i+2}	y_{i+1}	y_i	演算
0	0	0	0
0	0	1	X
0	1	0	X
0	1	1	2X
1	0	0	-2X
1	0	1	-X
1	1	0	-X
1	1	1	0

Booth 法では一般に k ビットずつまとめて処理する場合，必要な被乗数の倍数 mX は $-2^{k-1} \leq m \leq 2^{k-1}$ の範囲であり，加減算器を用いれば $2^{k-1}+1$ 種類で済む．ただし，倍数の選択のために乗数の連続する $(k+1)$ ビットを検査する必要がある．



							1	1	1	0	1	1	0	1	被乗数	(X)
					×	)	0	1	1	0	1	0	0	1	乗数	(Y)
1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	部分積	1
0	0	0	0	0	0	0	1	0	0	1	1	0				2
0	0	0	0	0	0	1	0	0	1	1						3
1	1	1	0	1	1	0	1	0								4
1	1	1	1	0	0	0	0	0	1	1	0	1	0	1		

図 2.13　2 次の Booth アルゴリズムを用いた乗算

第 3 章　代表的な乗算器の分類

3.1　逐次型乗算器

乗算とは被乗数を乗数桁分加算することであるが，筆算と同様，桁ごとに被乗数と乗数の 1 桁の乗算と加算およびシフトを乗数の桁だけ行う. 2 進数での乗数は 1 か 0 であるため，乗数が 1 のときは加算とシフトを行い，0 のときはシフトのみを行う．したがって，逐次型乗算回路は被乗数 X，乗数 Y，積 P を格納する 3 つのレジスタおよび, 加減算回路, シフト制御回路からの構成となる. 積 P は, 被乗数 X と乗数 Y を合わせた桁の長さの倍長レジスタを必要とするが, 乗数 Y の使用済みビットの場所を積に使用することで，積 P のレジスタの下位と乗数 Y を共有できる.

3.1.1　Robertson 法

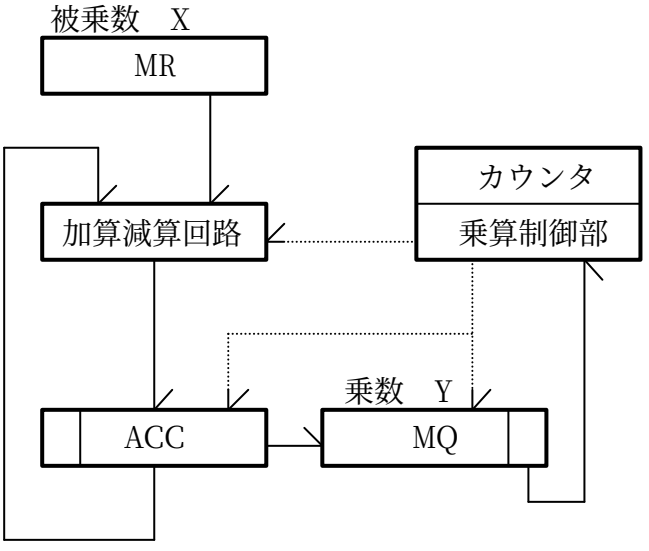


図 3.1　逐次型乗算器（Robertson 法）のブロック図

図 3.1 は逐次型乗算器（Robertson 法）の構成である．被乗数 X はレジスタ MR に，乗数 Y はレジスタ MQ に格納する．アキュムレータ ACC の初期値は 0 である．積 P は上位が ACC，下位が MQ に結果として残される．数値は最上位ビット（MSB）が符号ビットである 2 の補数表現とする．乗数 Y が正の場合の演算は MQ（乗数 Y）の最下位桁をスキャンすることから始まる．最下位桁が 1 のとき，ACC に MR（被乗数 X）を加算し，ACC と MR レジスタを連結して右シフトを行う．このときのシフトでは，最上位桁にはシフト前と同じビットが挿入される．これは乗数が負の値を 2 の補数によって表現をしているため，数値の正負にかわらず正しい結果を得るためである．この操作を乗数 Y の桁数だけ実行することで ACC と MQ レジスタに積 P の上位と下位が残される．乗数 Y が正数の場合には，このままで正しい結果が出力されるのだが，負数の場合には，最後の桁が符号ビットで 1 となっているので，これを下位の桁と同様に計算するのではなく，減算，つまり 2 の補数を加算する．これによって，2 進数の性質から正しい結果を出力できる．上記をまとめると次のようになる．

【X＞0， Y＞0 の場合】
X と Y_{-i} との部分積は，i 桁分符号桁 0 の拡張を行って求める．部分積を加算する．

【X＜0， Y＞0 の場合】
X と Y_{-i} との部分積は，i 桁分符号桁 1 の拡張を行って求める．部分積を 2 の補数で加算する．〔例 a〕

【X＞0， Y＜0 または X＜0， Y＜0 の場合】
先と同様，符号拡張行った部分積を 2 の補数で加算し，X の補数を加算する．〔例 b〕

〔例 a〕

```
                1 1 1 0 1
          ×  )  0 0 1 1 0
-------------------------
1 1 1 1 1 1 0 1
1 1 1 1 1 0 1
-------------------------
1 1 1 1 0 1 1 1 0
```

〔例 b〕

```
                      1 1 1 0 1
                ×  )  1 0 1 0 1
-----------------------------
      1 1 1 1 1 1 1 0 1
      1 1 1 1 1 0 1
+     0 0 0 1 1
-----------------------------
1 0 0 0 1 0 0 0 0 1
```

3.1.2　Booth 法

第 2 章で述べた Booth アルゴリズムを用いた逐次型乗算器である. 図 3.2 は 1 次の Booth アルゴリズムを用いた逐次型乗算器のブロック図である.

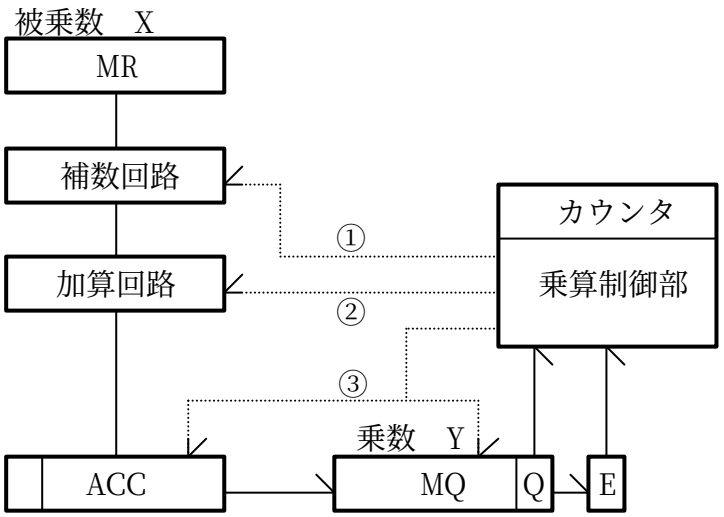


図 3.2　逐次型乗算器（Booth 法）のブロック図

3.3.1 と同様，被乗数 X を MR へ，乗数 Y を MQ へ格納する．E はフリップフロップであり，ACC，MQ および E は連結し，シフトする．ACC と E の初期値は 0 である．乗算制御部は Booth アルゴリズムに基づき，次のように処理される.

【Q＝0，E＝0 の場合】・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・処理③
右シフトのみ行う.
【Q＝0，E＝1の場合】・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・処理②
被乗数 *X* を ACC に加算後，右シフトさせる.
【Q＝1，E＝0 の場合】・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・処理①
被乗数 *X* の補数をとり，ACC に加算後，右シフトさせる
【Q＝1，E＝1の場合】・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・処理③
右シフトのみ行う.

これらを制御カウンタで示す回数だけ繰り返す．最終的には ACC の最上位桁には積 P の符号が格納される．

3.2　完全並列型乗算器

並列型乗算器は多数の加算器を結合した構成となっている．特長として次のようなことがあげられる．

① シフト操作をなくした高速な乗算器．
② 逐次型乗算器の制御部に相当するものがなく，高速である．
③ 多段に配置された加算器群にはフィールドバックがなく，パイプライン構成に適している．

3.2.1　アレイ乗算器

絶対値表示の乗算は以下のように m 個の部分積の和を求めることで得られる．並列乗算器は１チップの n 倍精度には n^2 個の接続で乗算の演算精度が拡張可能である．筆算手順において乗算の演算方式は，乗数 Y が 0 の場合は部分積が 0 となり，乗数 Y が１の場合は部分積が被乗数の値となる．被乗数 X と乗数 Y が 4 ビットとして積 P を 8 ビットとした場合を図 3.3 に示す．

また 1 ビット部分積和セルを図 3.4 に示す．この並列乗算方式は多数の加算器を乗算時間は素子の遅延時間によって決まるので非常に高速である．しかし，部分積の加算回数に等しい数の並列加算器が必要となり，精度拡張時においては精度増に伴いファンイン／ファンアウトおよび大幅なゲート数増等の問題が起こる．

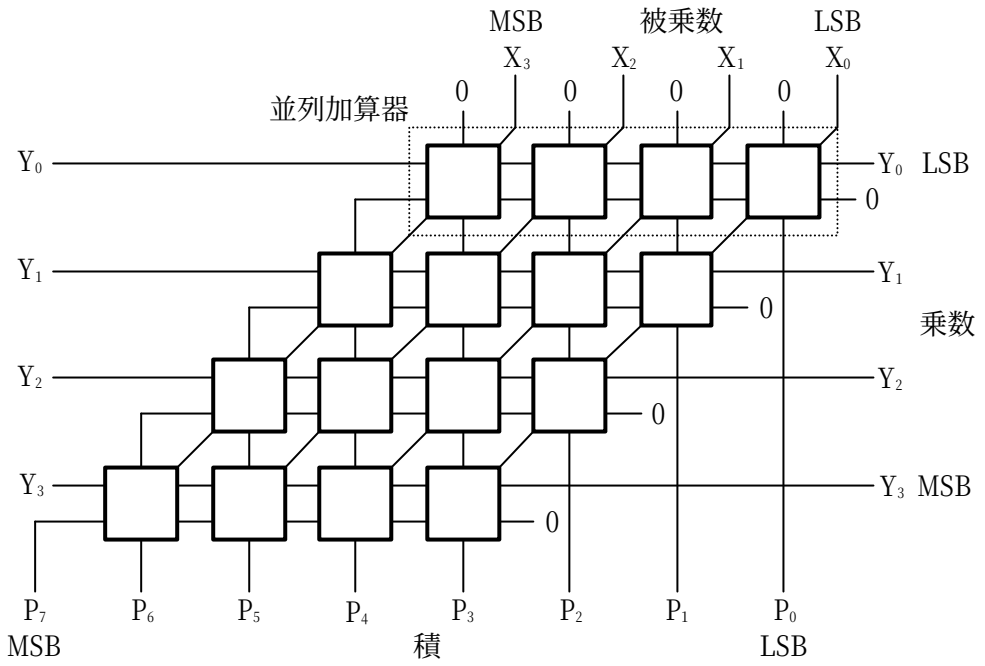


図 3.3　完全並列型アレイ乗算器

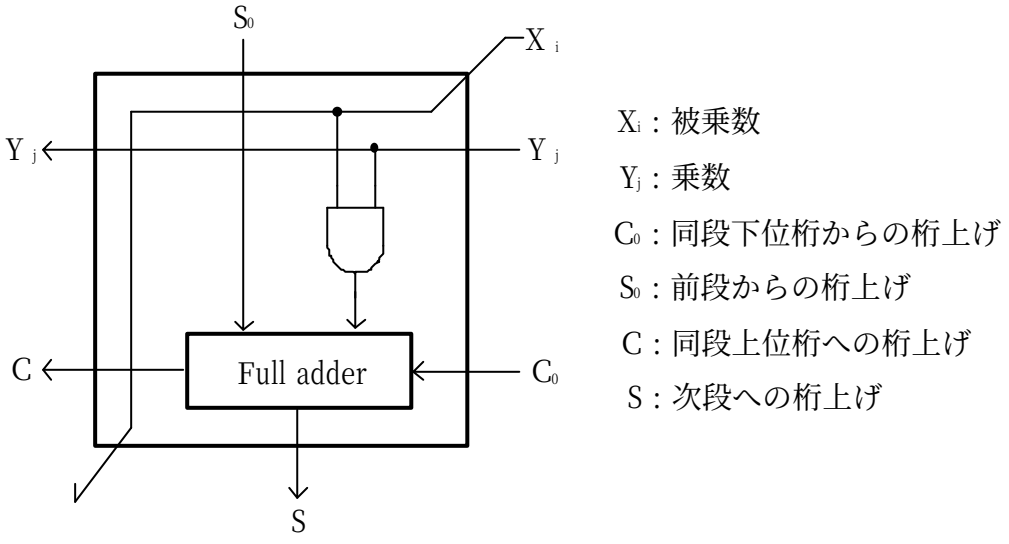


図 3.4　1 ビット部分積和セル

3.2.2　Wallace tree 乗算器

Wallace tree 乗算器とは桁上げ保存加算器（以下，CSA）による加算の段数を減らすために，CSA を tree 状に配置したものである．加算の回数は同じ複数個の CSA を用いるアレイ乗算器と変わらないが，CSA を並列に配置し，並列に演算をさせることにより結果が求まるまでの論理段数，つまり，遅延時間を減らそうとしたものである．通常の全加算器は 3 入力 2 出力なので，並列度を最大にすると最短で $\log_{3/2} n$ 段の全加算器を通すことにより，すべての部分積の加算が終了する．乗算演算において乗数 Y の各ビットにおける加算を行う順序を変えたとしても最終結果は変わらないので，このようなアルゴリズムが可能である．なお，Wallace tree 乗算器は桁間および段間をわたる桁上げ信号や和信号の配線に規則性がなくアレイ化に難があるため，16 ビット程度までならばむしろアレイ方式が実用的である．それ以上では加算段数の増加が log の関数で収まる Wallace tree が望ましい．

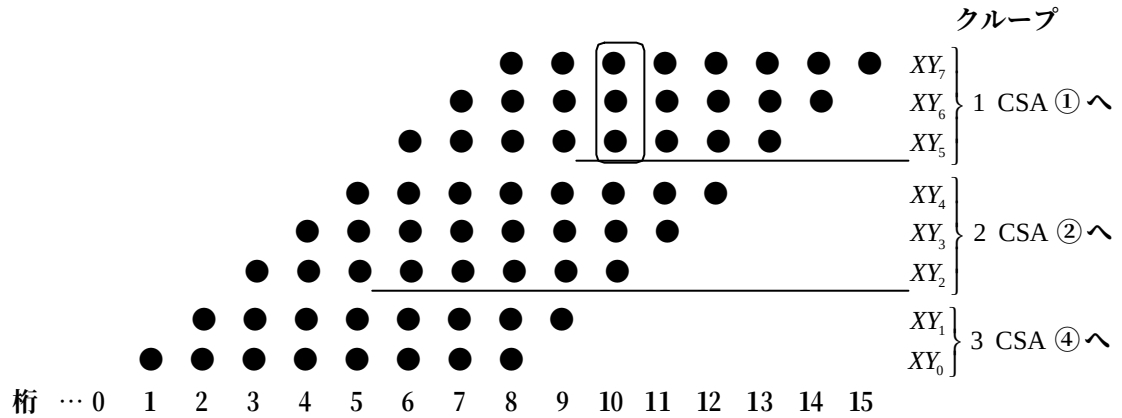


図 3.5　8×8 乗算における部分積のドット表示

図 3.5 は 8 ビット×8 ビット乗算の部分積のドット表示である．8 個の部分積を 3 つずつのグループに分割し，グループ内の各行に対応する 3 つの数は CSA で計算される．CSA の和を S，キャリーを C とする．

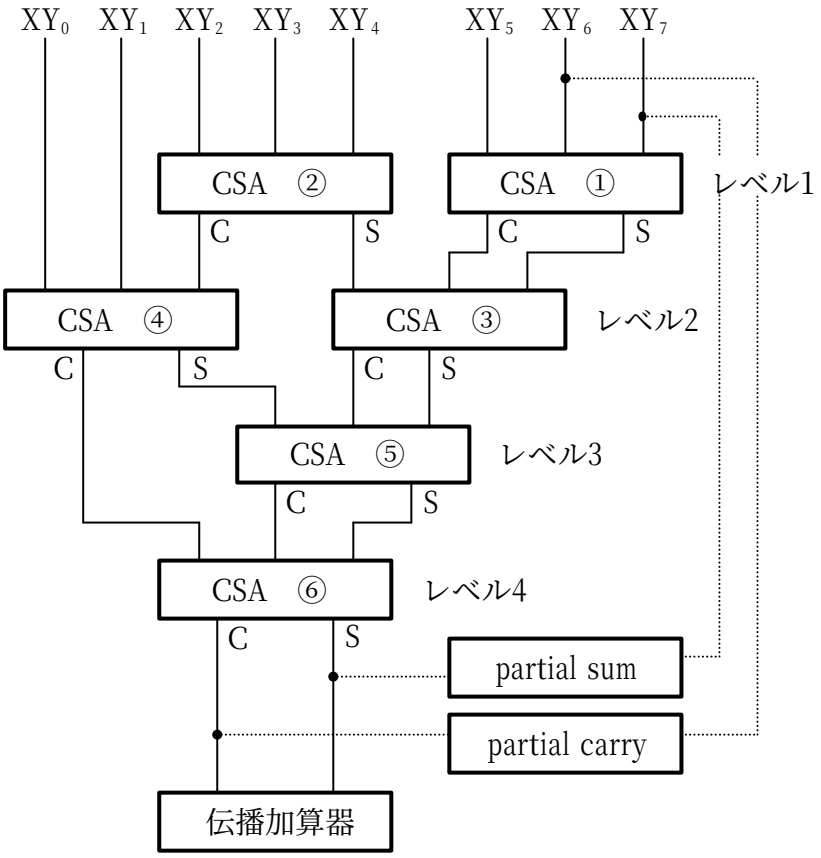


図 3.6　Wallace-tree 乗算器

8 ビット×8 ビットの Wallace tree 乗算器を図 3.6 に示す．CSA ① と ②で同時にグループ 1, 2 に対して加算する．キャリーは保存しておく．次に CSA ① の S と C，CSA ② の S を CSA ③ へ，XY_0，XY_1 と CSA ② の C を CSA ④ に入力する．以下，図 3.6 のようにレベル 4 まで CSA で加算される．最終段はリップルキャリーあるいは，桁上げ先見加算器で構成される．レベルの数は次のように求められる．加算すべき部分積の個数 m がレベル 1 後に $(2/3)m$ 個，レベル 2 後に $(2/3)^2 m$ 個となり，最終的に 2（すなわち，$(2/m)m$）個となる．したがってレベル数を L とすると，$2/m = (2/3)^L$ より

$$L = (\log_2 m - 1)/(\log_2 3 - 1) = 1.71(\log_2 m - 1)$$

となる．最終段を桁上げ先見方式で行うものとすると，計算時間は $O(\log_2 m)$ となる．

ここでは 8 ビットの乗数 Y を示しているが，乗数 Y のビット数が更に長い場合には, 最終段の桁上げ伝播加算器に入る前から仮の partial sum と carry の両方をラッチに入れて，図 3.6 の破線のようにレベル 1 の CSA へ戻してやり，所要の回数まわしてやる必要がある．このとき，必要となるシフト動作はシフト桁数が常に一定であるので，sum/carry を戻すときの配線でシフトを行えばよい．無論，シフトによって飛びで出してしまうビットは，他の適当なレジスタに保持させる必要がある．

乗算は加算と並んで計算機の基本的は演算の一つであるので，これを高速に実行することが重要となる．Wallace tree 構造での要点は，tree をなす CSA を何個・何段使用し，これを何回繰り返して処理するかという点にある．例えば，64 ビット×64 ビットの乗算の場合, 図 3.6 の乗数 8 ビットの Wallace tree では約 11 回の繰り返し演算が必要となる. partial sum/carry が図の破線で示したようにラッチ経由で入力に回ってくるので，1 回の Wallace tree の使用で実質は乗数 6 ビット分の部分積しか求められないからである．そこでより高速化を目指す場合，次のような手法がとられている．

- 1 回の繰り返しで Wallace tree が処理する乗数 Y のビット数 k をできるだけ大きくする．
- k ビットの中を更に複数ビットずつに分け，乗数 Y をデコードし，それぞれに対応する被乗数 X の倍数を予め生成し，Wallace tree の CSA へ被乗数として，図 3.7 のように供給させる．
- 桁移動度は固定長（k ビット）の桁移動ですみ，前途のような配線を行う．

この方式の図 3.7 では図 3.6 のような 64 ビットの CSA を 6 個使用した Wallace tree を 1 回使用することにより，実質倍増の 12 ビット分の乗数部の部分積の sum/carry を求めることができるようになる．つまり，乗数 Y を m ビットずつに分割して，これをデコードして使用することにより，繰り返し回数を 1/m に減らすことを狙っている．そのためには 2m 個の被乗数 X の倍数をあらかじめ用意しておく必要があるが，これは m が大きな数値であるときには現実的ではない．その代わりに数組の加算器（CSA）を利用して毎回これを作り出している．これに相当する操作が Wallace tree による乗算である．Wallace tree は m ビットの乗算を高速に行うものとして活用されている．

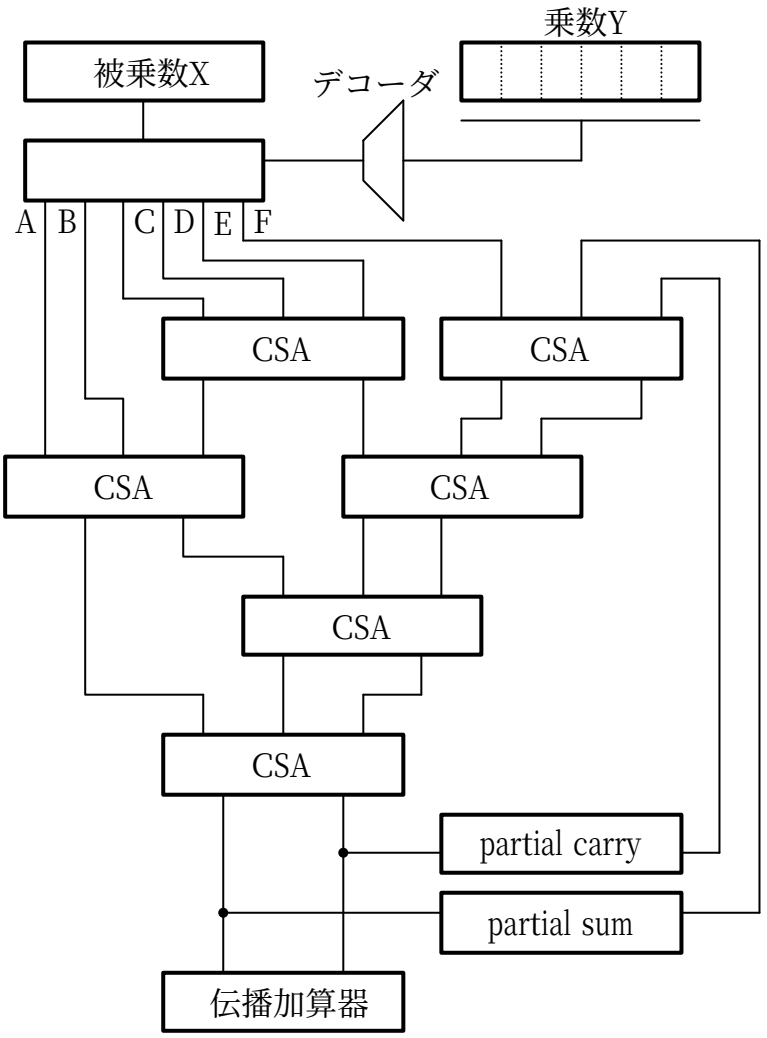


図 3.7　加速 Wallace tree 乗算器

3.3　準並列型乗算器

準並列型方式では，完全並列型乗算器と繰り返し演算機構としての逐次型乗算器の両機構を取り入れた形となっている．1 チップの n 倍精度には同チップを n 個カスケード接続し，n－1 回の繰り返し演算を行うことにより拡張した乗算結果を得られる．その繰り返し演算用としての部分積ラッチ機構を完全並列型乗算器に付加して，この乗算器を 1 モジュールとする．この乗算モジュールの演算精度を 4 ビットとして 2 倍の 8 ビット精度に拡張したブロック図を図 4 に示す．

1 モジュールは被乗数 X×乗数 Y の 4 ビット乗算器で 8 ビットの積 P を出力する．モジュール内部ではラッチ機構 4 個と 1 ビット部分積和セル 4 個で構成されており，前者はポジティブエッジ型の D フリップフロップを使用し，後者は図 3.4 と同様に部分積用としての AND ゲートと前段の部分積との和をとるための加算器を使用している．

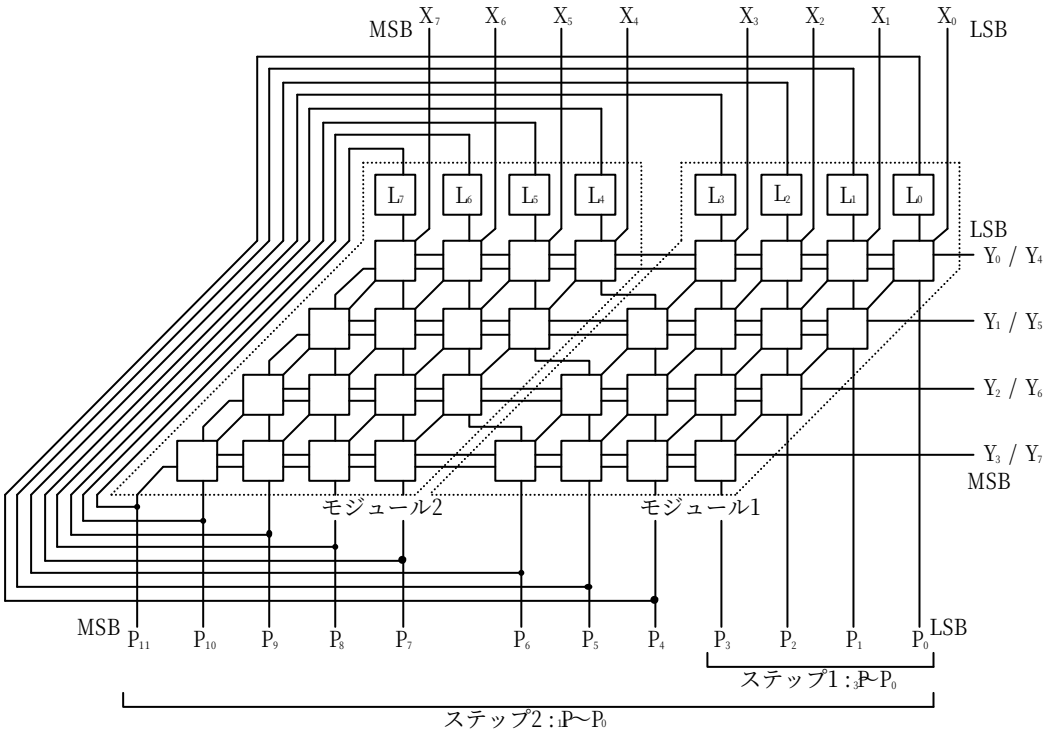


図 3.8　準並列型アレイ乗算器のブロック図

この乗算器では D フリップフロップを使用して演算を行っていることから，クロックによる状態遷移が生じるのでステップ毎での演算動作を以下に示す．

≪ステップ 1≫

① ラッチ機構の内容をクリア（初期化）する．

② 被乗数 X の上位 4 ビット（X_7〜X_4）を上位側のモジュール 2 に与える．
被乗数 X の下位 4 ビット（X_3〜X_0）を下位側のモジュール 1 に与える．
乗数 Y の下位ビット（Y_3〜Y_0）を両モジュールに与える．

③ 乗算結果 P の下位 4 ビット（P_3〜P_0）が得られる．

④ 乗算結果 P の上位 8 ビット（P_{11}〜P_4）は部分積としてラッチ機構（L_7〜L_0）へ伝播する．
その際に，モジュール２では P_{11}〜P_8 を自モジュールの L_7〜L_4 へ伝播し，P_7 をモジュール 1 の L_3 へ伝播する．
また，モジュール 1 では P_6〜P_4 を自モジュールの L_2〜L_0 へ伝播する．

≪ステップ 2≫

⑤ ラッチ機構へクロックを与え，乗算結果 P の上位 8 ビット（P_{11}〜P_4）を記憶する．

⑥ 被乗数 X の上位 4 ビット（X_7〜X_4）を上位側のモジュール 2 に与える．
被乗数 X の下位 4 ビット（X_3〜X_0）を下位側のモジュール 1 に与える．
乗数 Y の下位ビット（Y_7〜Y_3）を両モジュールに与える．

⑦ 乗算結果 P の残り上位 12 ビット（P_{11}〜P_0）が得られる．

3.4　高基数乗算器

3.4.1　基数 4 の乗算

決められた大きさの数を高基数で表現すると少ない桁数で済む．つまり，桁ごとに処理する乗算アルゴリズムでは，高基数に変換することで，より少ないサイクルで処理することが可能である．

2 進数 k ビットの数値は，4 進数$\lceil k/2 \rceil$桁，8 進数$\lceil k/3 \rceil$桁の数値に変換することができ，高基数乗算を行えば本質的に各サイクルにおいて乗数 Y の複数ビットを取り扱える．まず，逐次型乗算の一般的な基数 r バージョンを提示することから始める：

$$P^{(j+1)} = (P^{(j)} + XY_j\, r^k)r^{-1} \quad \text{ただし} \quad P^{(0)} = 0,\ P^{(k)} = P$$

$$|\text{—— add ——}|$$

$$|\text{— shift right —}|$$

$$P^{(j+1)} = rP^{(j)} + XY_{k-j-1} \quad \text{ただし} \quad P^{(0)} = 0,\ P^{(k)} = P$$

$$\left|\begin{matrix}\text{shift}\\ \text{left}\end{matrix}\right|$$

$$|\text{—— add ——}|$$

r^{-1} もしくは r による乗算は，共に，桁による右シフトか左シフトを引き起こす．高基数乗算と基数 2 の乗算との唯一の違いは項 XY_j を構成する過程で，多くの計算を必要とすることである．例えば，乗算を基数 4 で行うならば，各ステップにおいて部分積項 $X(Y_{i+1}Y_i)_2$ が構成され，累積部分積に加算する必要がある．図 3.9 にドット表記での乗算プロセスを記す．

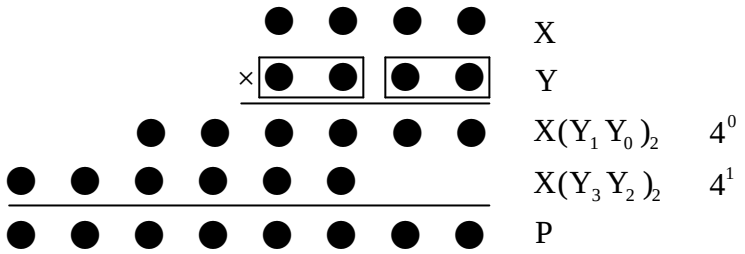


図 3.9　基数 4（2 ビット同時処理）の乗算のドット表記

この方法は次のような問題を抱える．基数 2 の乗算の部分積におけるドット列は 0 もしくは X をシフトしたものを表示するに対し，ここでは倍数 0X，1X，2X，3X を必要とする．0X，1X，2X の生成においては問題ないが（2X は X をシフトさせたもの），3X は少なくても加算演算（3X = 2X + X）が必要である．基数 4 の乗算での上記の問題の解決法をいくつか述べる．

最初の動作としてまず 3X を計算し，それを後に使用するためレジスタに保持させる．乗算のハードウェアは 2-way マルチプレクサと図 3.10 で示す 4-way マルチプレクサを取り替えることを除いては，図 2.3 で示したハードウェア乗算器とほぼ同じである．なお，図 3.10 は倍数 3X を用いた基数 4 の乗算の一例である．

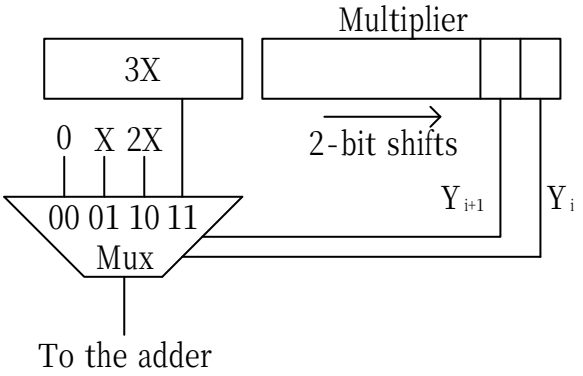


図 3.10　基数 4 の乗算器の倍数生成部と 3X の計算法

3X を必要とする場合の乗算方法は，上記に記した方法の他にも存在する．-X を加算し，乗数の次の基数 4 の桁に 1 の桁上げを送り出す（図 3.10）．桁上げ入力

を含め，各サイクルで必要な倍数は [0, 4]である．倍数 3，4 は桁上げ出力を 1 として，それぞれ-1 と 0 に変換するが，倍数 0，1，2 はダイレクトに出現する．最終的に余分なサイクルが桁上げのために必要となる場合がある．図 3.10 と図 3.12 で表される乗算方式は基数 8，16 へと拡張することができる．しかし，倍数生成ハードウェアは高基数にすることで，より複雑になってしまい，少ないサイクルによって支払われるべき処理速度の利得を犠牲にしてしまう．たとえば，基数 8 の乗算では，倍数 3X，5X，7X を前もって計算しておくか，3X だけを計算しておいて図 3.12 と同じような方法で倍数 5X, 6X, 7X をそれぞれ-3X, -2X，-X と桁上げ 1 に変換する．

X			0	1	1	0				
3X	0	1	0	0	1	0				
Y			1	1	1	0				
$P^{(0)}$			0	0	0	0				
$+X(Y_1 Y_0)_2$	0	0	1	1	0	0				
$4P^{(1)}$	0	0	1	1	0	0				
$P^{(1)}$			0	0	1	1	0	0		
$+X(Y_3 Y_2)_2$	0	1	0	0	1	0				
$4P^{(2)}$	0	1	0	1	0	1	0	0		
$P^{(2)}$			0	1	0	1	0	1	0	0

図 3.11　倍数 3X を用いた基数 4 の乗算例

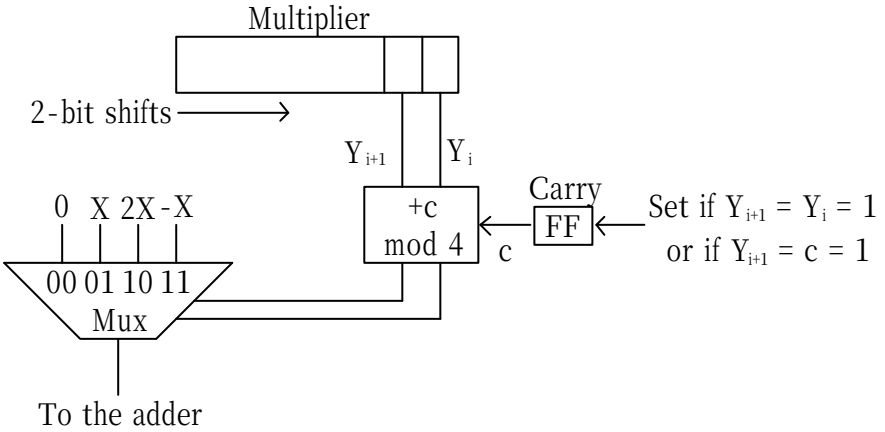


図 3.12　基数 4 の乗算に基づき 3X を 4X，-X へ置換する倍数生成部

3.4.2　2 次の Booth アルゴリズム

基数 2 の Booth デコードは現在の演算回路では直接利用されない．しかし，高基数の Booth デコードを理解するための方法として役立つ．表 3.1 を使って再コード化した場合，連続した 1 もしくは-1 が見受けられないことが容易にわかる．したがって，基数 4 の乗算を再コード化した乗数で実行するならば，被乗数の倍数±X と±2X だけを準備すればよい．それらは被乗数をシフトすることや補数をとることで，容易に得られる．Z_{i+1} は Y_{i+1} と Y_i，Z_i は Y_i と Y_{i-1} から決まるので，基数 4 の数値 $Z'_{i/2} = (Z_{i+1}\ Z_i)_2$（i は偶数）は Y_{i+1}，Y_i と Y_{i-1} から直接得られる．（表 3.2）

表 3.1　基数 2 の Booth デコーディング

Y_i	Y_{i-1}	Z_i
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

表 3.2　基数 4 の Booth デコーディング

Y_{i+1}	Y_i	Y_{i-1}	Z_{i+1}	Z_i	$Z'_{i/2}$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	$\bar{1}$	1
0	1	1	1	0	2
1	0	0	$\bar{1}$	0	$\bar{2}$
1	0	1	$\bar{1}$	1	$\bar{1}$
1	1	0	0	$\bar{1}$	$\bar{1}$
1	1	1	0	0	0

基数 2 の Booth デコーディングと同様に，基数 4 のデコーディングも数値変換とみなすことができる．すなわち，基数 4 の数値である[0, 3]が，数値[-2, 2]に変換される．一例として，表 3.2 は以下のような符号なし数から符号付き桁数への変換を実行するために用いられる：

$$(21\ 31\ 22\ 32)_4 = (10\ 01\ 11\ 01\ 10\ 10\ 11\ 10)_2 = (1\ \text{-}2\ 2\ \text{-}1\ 2\ \text{-}1\ \text{-}1\ 0\ \text{-}2)_4$$

16 ビットの符号なし数が基数 4 の 9 桁の数に変わることに注目する．通常，k ビット符号なし 2 進数の基数 4 の符号付き桁表現は MSB が 1 のとき，$\lfloor k/2 \rfloor + 1 = \lceil (k+1)/2 \rceil$桁を必要とする．ここで，$Y_{-1} = Y_k = Y_{k+1} = 0$と仮定していることに注意する．前述の例での 2 進数が 2 の補数形式で表されると解釈すると，基数 4 の桁を無視するだけで正しく再コード化された数値が得られる．

$$(1001\ 1101\ 1010\ 1110)_{2\text{'s-compl}} = (\text{-}2\ 2\ \text{-}1\ 2\ \text{-}1\ \text{-}1\ 0\ \text{-}2)_4$$

このように，2 の補数形式での k ビット 2 進数の場合，Booth デコードされた基数 4 の数値は$\lceil k/2 \rceil$桁になる．k が奇数のとき，正しく再コード化するため，$Y_k = Y_{k-1}$と仮定する．いずれにせよ，$Y_{-1} = 0$ である．
基数 4 の Booth デコーディングでの数値変換では，キャリー伝播を引き起こさない．1 つのビットを重ね合わせながら連続する 3 ビットを調べていけば，基数 4 の場合の各桁の数[-2, 2]は独立に決まる．この理由により，基数 4 の Booth デコーディングは乗数の連続した 3 ビットを重ね合わせながらスキャンする方法であると言われている．この考え方はより基数に対する多重ビット・スキャン方式に拡張できる．Booth デコーディングを用いた基数 4 の乗算例を図 3.13 で示す．

X			0	1	1	0						
Y			1	0	1	0						
Z'				$\bar{1}$		$\bar{2}$						
$P^{(0)}$	0	0	0	0	0	0						
$+XZ'_0$	1	1	0	1	0	0						
$4P^{(1)}$	1	1	0	1	0	0						
$P^{(1)}$	1	1	1	1	0	1		0	0			
$+XZ'_1$	1	1	1	0	1	0						
$4P^{(2)}$	1	1	0	1	1	1		0	0			
$P^{(2)}$			1	1	0	1		1	1	0	0	

図 3.13　Booth 法を用いた基数 4 の乗算例

2 の補数表示された 4 ビットの乗数 $Y = (1010)_2$ は 2 桁の基数 4 $Z' = (-1\ -2)_4$ に変換し，それらを 2 サイクルで累積部分積に加算するため，倍数 $XZ'_0 = -2X$ と $XZ'_1 = -X$ を生成するこれらの計算過程で負の数として適切な処理を行うために部分積の上位ビットに 0 を符号拡張して 4 ビットから 6 ビットとする．

また，右シフトの間の符号拡張が $4P^{(1)}$ から $P^{(1)}$ を得るときに行われることに注意する．図 3.14 に基数 2 の Booth デコーディングに基づく倍数生成回路を示す．

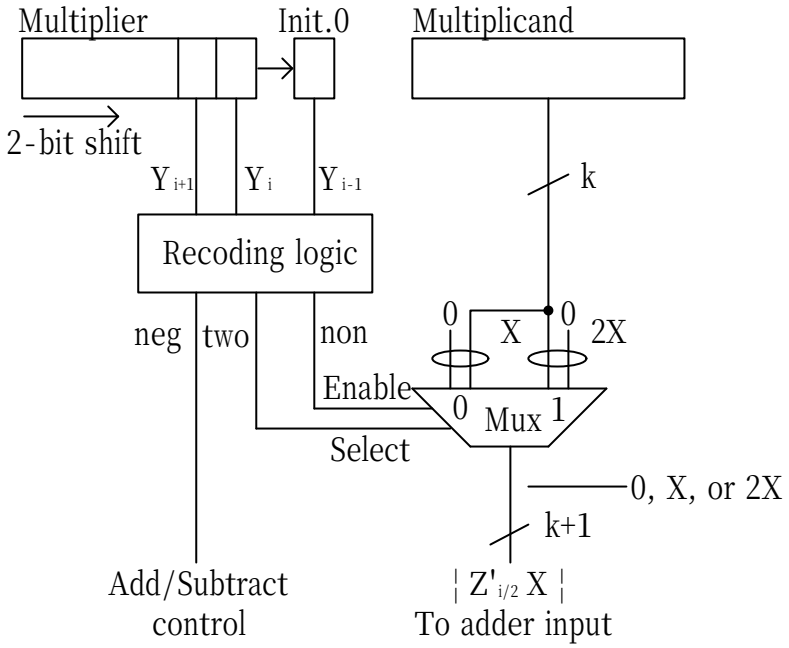


図 3.14　Booth 法を用いた基数 4 の倍数生成部

X は桁（0，±1，±2）の 5 つの倍数を必要とするので，要求する倍数をエンコードするためには少なくとも 3 ビットが必要である．容易かつ効率的なデコードを行うには，0 と nonzero 桁を区別する 1 ビット，nonzero 桁の符号への 1 ビット，nonzero 桁の絶対値への 1 ビットを当てることである．デコード回路は，このように 3 つの入力（Y_{i+1}, Y_i, Y_{i-1}）により，3 つの出力を生み出す．

「neg」が 0 のとき，倍数を加算し，1 のとき，倍数を減算する．
「non0」が 1 のとき，倍数が 0 でないことを示す．
「two」が 0 のとき，「non0」の倍数は 1 倍であり，1 のとき，「non0」の倍数は 2 倍である．

図 3.14 の Booth デコーディングを，図 3.12 の方式との比較してみると，図 3.12 のデコードは直列であり，右から左へと逐次処理しなければならないのに対して図 3.14 のデコードは完全に並列であるので，桁上げが生じないことに注目されたい．Booth デコーディングは逐次型乗算器のみならず，全ての倍数を同時に必要とする Wallace tree 乗算器やアレイ乗算器などの並列型乗算器の設計にも利用されている．

第 4 章　むすび

本論文では，乗算器の基本構成とその動作原理についての調査をおこなった．乗算器はディジタル信号処理システムの基本的構成要素であり，ハードウェアアルゴリズムによって，演算処理速度や使用面積に差が出てくる．演算処理の高速化と面積の低減化という相反する課題をどのようにまとめ，ある特定用途にあった乗算アーキテクチャの考案に近づけるための基礎学習として本論文を位置付ける．

謝辞

日頃から懇切丁寧な御指導と御助言を賜りました高知工科大学工学部電子・光システム工学科矢野政顯教授に心から感謝致します.

全過程を通じ，懇切丁寧な御指導と御助言を賜りました高知工科大学工学部電子･光システム工学科学科長原央教授ならびに，高知工科大学工学部電子･光システム工学科橘　昌良助教授に厚く御礼申し上げます.

また終始，適切なる御助言，御助力を頂きました矢野研究室，原研究室，橘研究室の諸氏に心から感謝致します.

参考文献

電子情報通信学会：“電子情報通信ハンドブック”，オーム社（1988）

中村次男：“ディジタル回路設計法　―ワンチップ化の実例集―”，日本理工出版会（1990）

渡邉勝正：“コンピュータ概論”，丸善（1992）

Neil H. E. Weste, Kamran Eshraghian : “PRINCIPLES OF CMOS VLSI DESIGN”, ADDISON WESLEY (1992)

柴山　潔：“コンピュータアーキテクチャの基礎”，近代科学社（1993）

電子情報通信学会：“ディジタル信号処理ハンドブック”，オーム社（1993）

富田眞治：“コンピュータアーキテクチャⅠ”，丸善（1994）

中澤喜三郎：“計算機アーキテクチャと構成方式”，朝倉書店（1995）

高橋　茂，工藤知宏：“計算機工学概論”，共立出版（1996）

富田眞治，中島　浩：“コンピュータハードウェア”，昭晃堂（1998）

電子情報通信学会：“エンサンクロペディア　電子情報通信ハンドブック”，オーム社（1998）

Behrooz Parhami : ”Computer Arithmetic”, Oxford University Press (1999)

John L. Hennessy, David A. Patterson：“コンピュータの構成と設計　第2版”，日経 BP 社（1999）