3. Sequential Algorithms for Multiplication and Division

基本の直列型アルゴリズムを紹介する

- 乗算
- 除算
- 平方根演算

3.1 直列乗算

X と A をそれぞれ乗数と被乗数とする。

$$X = x_{n-1}x_{n-2}\cdots x_1x_0,$$
$$A = a_{n-1}a_{n-2}\cdots a_1a_0$$

符号 - 絶対値や補数方法の場合、x_{n-1} と a_{n-1} が符号ビットになる。

直列の乗算アルゴリズムでは、次のような処理ステープを $n-1$ 回行う。

ステープ j の時、乗数のビット x_j を調べ、積 x_jA を累加された部分積 $P^{(j)}$ に加える。

この繰り返しな手順を式で表すと

$$P^{(j+1)} = \left(P^{(j)} + x_j \cdot A\right) \cdot 2^{-1}; j = 0, 1, 2, \cdots, n-2 \quad (1)$$

最初に $P^{(0)} = 0$。

$$
\begin{aligned}
P^{(n-1)} &= \left(P^{(n-2)} + x_{n-2} \cdot A\right) \cdot 2^{-1} \\
&= \Big(\left(P^{(n-3)} + x_{n-3} \cdot A\right) \cdot \\
&\quad 2^{-1} + x_{n-2} \cdot A\Big) \cdot 2^{-1} \\
&= \cdots \\
&= \left(x_{n-2}2^{-1} + x_{n-3}2^{-2} + \cdots\right. \\
&\quad \left. + x_0 2^{-(n-1)}\right) \cdot A \\
&= \left(\sum_{j=0}^{n-2} x_j 2^{-(n-1-j)}\right) \cdot A
\end{aligned}
$$

絶対値の積：

もし、演算数が両方とも正であれば、（すなわち、$x_{n-1} = a_{n-1} = 0$), 積 U が次のように求められる。

$$U = 2^{n-1} \cdot P^{(n-1)} = (\sum_{j=0}^{n-2} x_j 2^j) \cdot A = X \cdot A \quad (2)$$

結果はその絶対値とした $2(n-1)$ ビットの積になる。

積の語長：

最大の絶対値は次の式で示す。

$$\begin{aligned} U_{max} &= (2^{n-1}-1)(2^{n-1}-1) \\ &= 2^{2n-2}-2^n+1 \\ &= 2^{2n-3}+(2^{2n-3}-2^n+1) \end{aligned} \tag{3}$$

最後項が $n \geq 3$ の場合正であるから、

$$2^{2n-3} < U_{max} \leq 2^{2n-2}; n \geq 3 \tag{4}$$

$(2n-2)$ ビットが必要で、符号ビットを含めば $(2n-1)$ ビットが要る。

絶対 - 値符号数のため、上のアルゴリズムを用いて、２つの絶対値を乗算し、符号はその結果とは別々に得られる（２つの演算数が同じ符号だったら正、そうでない場合は負）。

２の補数法と１の補数法においては、負の乗数 X の乗算と負の被乗数 A の乗算を区別しないといけない。

乗数 X が正数の場合：

被乗数 A だけが負の場合は、上記のアルゴリズムを変更する必要はない。

１の補数も２の補数もただ負数の倍数を加算すればいい。

[例 3.1]

次の乗算に、被乗数 A は 2 の補数法で表している負数で、乗数 X は正である。 両方とも 4 ビットである。

最終の乗算結果は符号ビットを含む 7 ビットになる。 4 ビットの演算ユニットにおいて、すべてのレジスターの長さが 4 ビットで、最終積を記憶するには倍長さのレジスタが必要である。

A		1 0 1 1	-5
X	$\times$	0 0 1 1	3
$P^{(0)} = 0$		0 0 0 0	
$x_0 = 1 \Rightarrow Add\ A$		1 0 1 1	
		1 0 1 1	
$Shift$		1 1 0 1	1
$x_1 = 1 \Rightarrow Add\ A$	$+$	1 0 1 1	
		1 0 0 0	1
$Shift$		1 1 0 0	0 1
$x_2 = 0 \Rightarrow Shift\ only$		1 1 1 0	0 0 1 $\quad -15$

最下位ビット x_0 から、乗数の 3 ビット、
$x_2, x_1,$ と $x_0,$ 一回一ビットずつ検出する。

すなわち、 add-and-shift 或は shift-only の演算が実行される。

最終結果は負数で、２の補数法により正しく表示されている。

積の下位半分のビットが加算されない。一番のレジスタの４ビットが全て使われており、２番目のレジスタの３ビットだけが使われている。

２番目のレジスタの符号ビットついて

（１）積の最下位に積符号に関わらずいつも０とセットする；

（２）１つめのレジスタの符号ビットに等しくする。

もう１つの方法として、２つめのレジスタの全ての四ビットを積の下位部分の４ビットに使われ、１つめのレジスタの右側２ビットを符号ビットとする。

負の乗数の場合：

乗数 X が負のとき、符号ビット（負の重み付き)がほかのビットと同じく扱えない。

まず、２の補数について考える。

$$X = -x_{n-1}2^{n-1} + \tilde{X} \tag{5}$$

$$\tilde{X} = \sum_{j=0}^{n-2} x_j 2^j$$

乗数 X の符号ビットを無視すれば、

$$\begin{aligned} U &= \tilde{X} \cdot A \\ &= (X + x_{n-1} \cdot 2^{n-1}) \cdot A \\ &= X \cdot A + A \cdot x_{n-1} \cdot 2^{n-1} \end{aligned} \tag{6}$$

$X \cdot A$ 項は求める積である。

したがって、 $x_{n-1} = 1$ の場合、次の補正が必要：

$$X \cdot A = U - A \cdot x_{n-1} \cdot 2^{n-1} \tag{7}$$

すなわち、もし $x_{n-1} = 1$、U の上位半分から被乗数 A を引くべきである。

[例 3.2]

乗数と被乗数が 2 の補数で表している負数である。

A		1 0 1 1		-5
X	$\times$	1 1 0 1		-3
$x_0 = 1 \Rightarrow Add\ A$		1 0 1 1		
$Shift$		1 1 0 1	1	
$x_1 = 0 \Rightarrow Shift\ only$		1 1 1 0	1 1	
$x_2 = 1 \Rightarrow Add\ A$	$+$	1 0 1 1		
		1 0 0 1	1 1	
$Shift$		1 1 0 0	1 1 1	
$x_3 = 1 \Rightarrow Correct$	$+$	0 1 0 1		
		0 0 0 1	1 1 1	$+15$

補正ステープでの減算は、その２の補数との加算で実行される。

１の補数表現の乗算：

同様に、１の補数での乗算の場合は、

$$X = -x_{n-1}(2^{n-1} - ulp) + \tilde{X} \qquad (8)$$

$$X \cdot A = U - x_{n-1} \cdot 2^{n-1} \cdot A + x_{n-1} \cdot ulp \cdot A \quad (9)$$

$x_{n-1} = 1$ のとき、 $P^{(0)} = A$ としてアルゴリズムを実行し始める。それが２項めの補正項 $x_{n-1} \cdot ulp \cdot A$ となる。

最後に１項めの補正項 $A \cdot x_{n-1} \cdot 2^{n-1}$ を引く。

[例 3.3]

1 の補数で 5 と −3 の積を求める。

A		0 1 0 1		5
X	×	1 1 0 0		−3
$x_3 = 1 \Rightarrow P^{(0)} = A$		0 1 0 1		
$x_0 = 0 \Rightarrow Shift$		0 0 1 0	1	
$x_1 = 0 \Rightarrow Shift$		0 0 0 1	0 1	
$x_2 = 1 \Rightarrow Add\ A$	+	0 1 0 1		
		0 1 1 0	0 1	
$Shift$		0 0 1 1	0 0 1	
$x_3 = 1 \Rightarrow Correct$	+	1 0 1 0	1 1 1	
		1 1 1 0	0 0 0	−15

補正項を引くことは１の補数を加算することにより実行される。

１の補数が２倍サイズの符号ビットで表さなければならないから (1.6 章を参考する）二倍長さの加算器が必要である。

3.2 直列除算

除算は４つの基本演算の中でもっとも複雑で、一番手間がかかる演算である。除算結果は２つの部分（商と余り）からなる。

被除数 X と除数 D、商 Q、 R は余数として計算式は次の通り

$$X = Q \cdot D + R \; with \; R < D \qquad (10)$$

X と D、結果 Q と余数 R を正数とする。

データ保存用のレジスタ：

多くの固定小数点演算ユニットにおいて、二倍長さの乗算結果が得られるので、一般に、乗算の結果を除算に使われるものになる。

したがって、X のため、２個分のレジスタを使う。商 Q は１つのレジスタに記憶できる数となる。

レジスタの長さを n とすると、レジスタにある数が 2^{n-1} より小さい。

整数の場合：

$Q < 2^{n-1}$ を保証するため、次の式が必要となる。

$$X < 2^{n-1}D$$

この条件を満たさない場合、オーバーフローを検出する必要がある。

X 或は D(両方）中の一つを予めシフトすることにより上の条件が常に成り立つことができる。

$D = 0$ ついて検出する必要である。

小数の場合：

商が小数であることを保証するためオーバフローの制約が $X < D$ に変わる。

小数の商

$$Q = 0 \cdot q_1 \cdots q_m \quad (m = n - 1)$$

を得るために、除算を数列の減算とシフトにより実行する。

商の i 桁めの決定方法：

ステープ i の時、余数が除数 D に比較され、もし余数が大きかったら、商のビット $q_i = 1$、そうでなければ $q_i = 1$。

$$r_i = 2r_{i-1} - q_i \cdot D; \qquad i = 1, 2, \cdots, m \quad (11)$$

r_i は新しい余数、 r_{i-1} は前の余数。最初の余数

$$r_0 = X$$

とする。 q_i は $2r_{i-1}$ と D の比較によって決められる。除算の中、比較演算は一番複雑である。

[証明]:

一番最後の r_m は次のように表す。

$$\begin{aligned} r_m &= 2r_{m-1} - q_m \cdot D \\ &= 2(2r_{m-2} - q_{m-1} \cdot D) - q_m \cdot D = \cdots \\ &= 2^m r_0 - (q_m + 2q_{m-1} + \cdots + 2^{m-1} q_1) \cdot D \end{aligned}$$

$r_0 = X$ と代入して、両側に 2^m で割って結果、

$$r_m 2^{-m} = X - (q_1 2^{-1} + q_2 2^{-2} + \cdots + q_m 2^{-m}) \cdot D$$

$$r_m 2^{-m} = X - Q \cdot D \qquad (12)$$

[例 3.4]

$X = (0.100000)_2 = 1/2$ そして
$D = (0.110)_2 = 3/4$ として。被除数が二倍長さレジスタを占める。$X < D$ という条件が明らかに成立する。

$r_0 = X$		0 .1 0 0	0 0 0	
$2r_0$		0 1 .0 0 0	0 0	$set\ q_1 = 1$
$Add - D$	$+$	1 1 .0 1 0		
$r_1 = 2r_0 - D$		0 0 .0 1 0	0 0	
$2r_1$		0 .1 0 0	0	$set\ q_2 = 0$
$r_2 = 2r_1$		0 .1 0 0	0	
$2r_2$		0 1 .0 0 0		$set\ q_3 = 1$
$Add - D$	$+$	1 1 .0 1 0		
$r_3 = 2r_2 - D$		0 0 .0 1 0		

$2r_0$ の生成はオーバフローを引き起こさないようにするため、算術ユニットに特別なビット設ける必要である。

最終結果は $Q = (0.101)_2 = 5/8$ と
$R = r_m 2^{-m} = r_3 2^{-3} = 1/4 \cdot 2^{-3} = 1/32$

(正確な結果は無限循環小数
$2/3 = 0.1010101\cdots$)。

商と最終余数は、 $X = Q \cdot D + R =$
$5/8 \cdot 3/4 + 1/32 = 16/32 = 1/2$ を満たす。

整数除算への適用：

もちろんこの手順は演算子と結果が整数の場合にも適用できる。

$$2^{2n-2}X_F = 2^{n-1}Q_F \cdot 2^{n-1}D_F + 2^{n-1}R_F \quad (13)$$

X_F, D_F, Q_F, R_F は小数である。 (2^{2n-2} で割ると

$$X_F = Q_F \cdot D_F + 2^{-(n-1)}R_F \quad (14)$$

上に述べた条件 $X < 2^{n-1}D$,2^{2n-2} で割ると
$X_F < D_F$ の形になる。

[例 3.5]

前の例の演算子と結果を整数とする。

2 倍長の被除数は $X = 0100000_2 = 32$ で、除数は $D = 0110_2 = 6$ である。

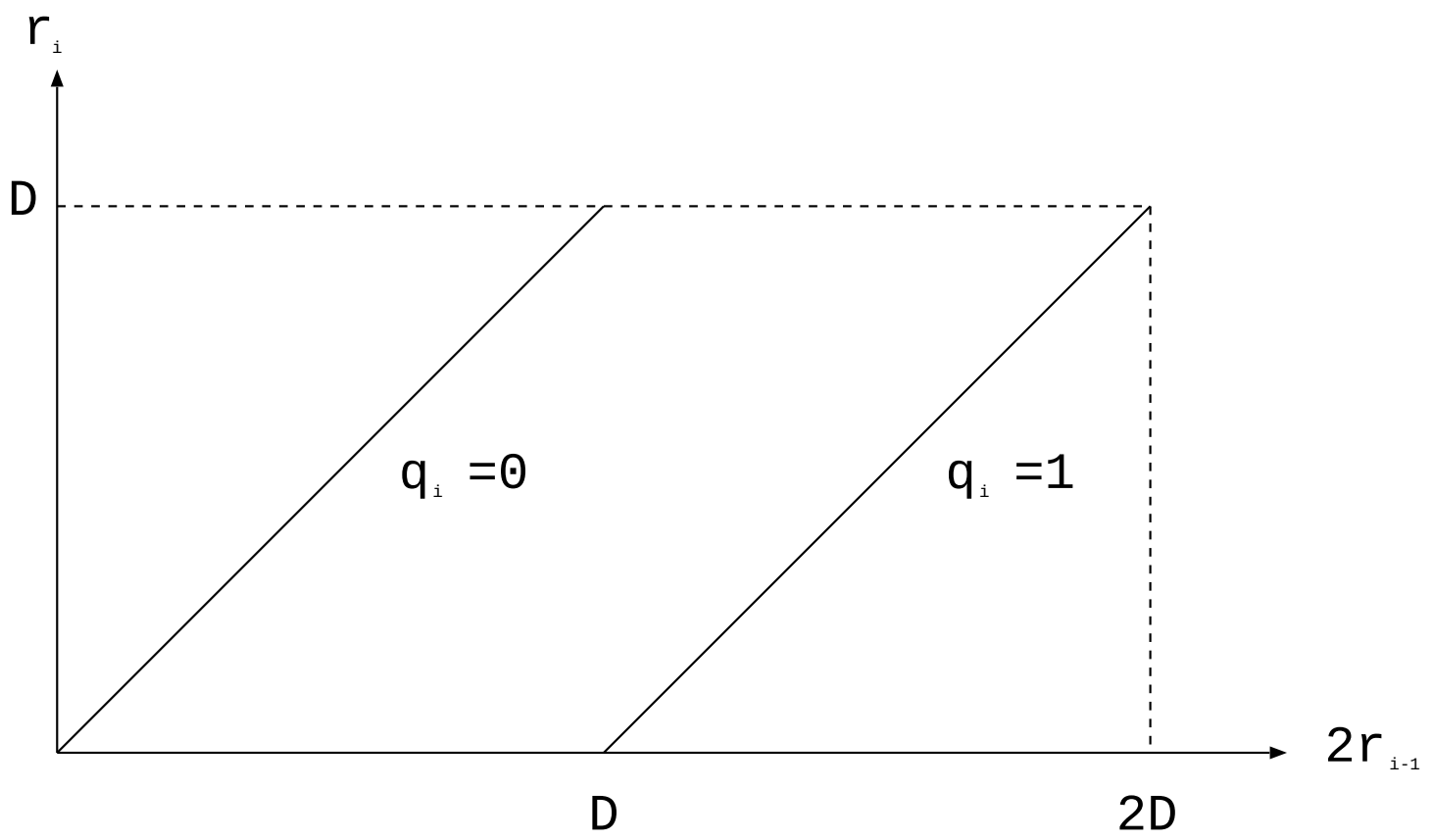
overflow 条件 $X < 2^{n-1}D$ は X の上位ビット 0100 と D (0110) とを比較することにより検証する。

最終のステップにおいて、本当の余り R が生成され、 余りと $2^{-(n-1)}$ との乗算を行う必要がない。

除算手続きにおいてもっとも難しいステップは、比較である。

$2r_{i-1}$ から D を引くと、結果が負の場合、$q_i = 0$ として、その前の値を余りとして保存しなければならない。

この方法は restoring division （復元法）という。



復元除算法

3.6 非復元除算

除算を行って結果を得るもう一つの方法は、非復元除算アルゴリズムである。

- 余りが負の場合、すぐに復元させることをしない。
- 修正は次のステップを後回しにすることによって行うことができる。

復元法の特徴：

復元法において、$2r_{i-1} - D$ が負の時、余りは $2r_{i-1}$ である。

それをシフトさせ、再び D を引くと、$4r_{i-1} - D$ を得る。

この処理は、余りが負となるまで繰り返される。

非復元法：

非復元方法では、負の余り $2r_{i-1} - D < 0$ になった場合、それをシフトし、D を加えることにより復元演算を避ける。

$$2(2r_{i-1} - D) + D = 4r_{i-1} - D$$

従って、復元除算方法と同じ余りを得る。

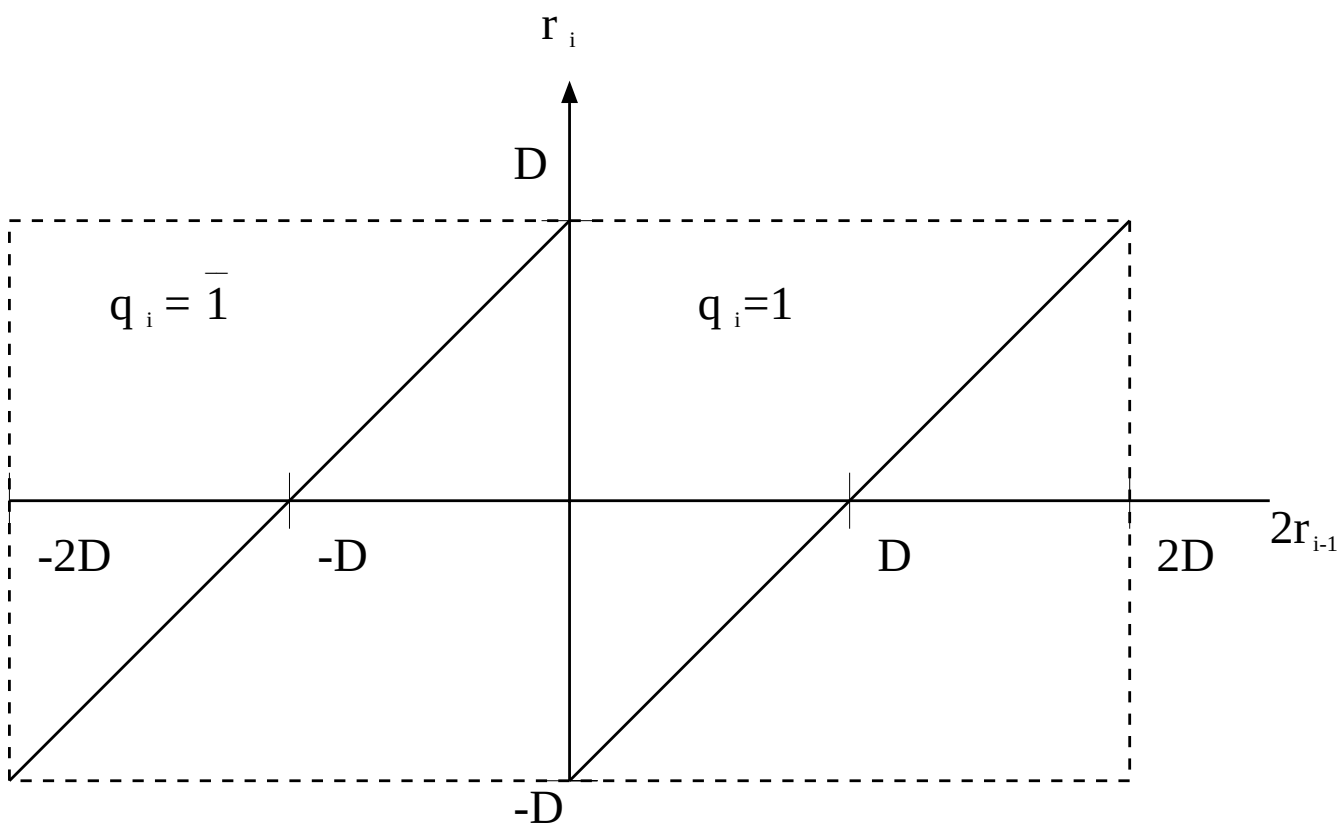
商の結果について

ステップ i で「誤った」商の選択の訂正を可能にするために、負と仮定した商 q_{i-1} の存在を許す。q_i は 1 と $\bar{1}(=-1)$ を取る。

非復元法における商は次の式で決定される。

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq 0 \\ \bar{1} & \text{if } 2r_{i-1} < 0 \end{cases}$$

$$r_i = 2r_{i-1} - q_i \cdot D$$



非復元乗算法

[例 3.6]

[例 3.4] において
$X = (0.100)_2 = 1/2, D = (0.110)_2 = 3/4$
とする。

$r_0 = X$			0	.1	0	0	
$2r_0$		0	1	.0	0	0	set $q_1 = 1$
Add $-D$	+	1	1	.0	1	0	
r_1		0	0	.0	1	0	
$2r_1$			0	.1	0	0	set $q_2 = 1$
Add $-D$	+		1	.0	1	0	
r_2			1	.1	1	0	
$2r_2$			1	.1	0	0	set $q_3 = \bar{1}$
Add D	+		0	.1	1	0	
r_3			0	.0	1	0	

最後の余りは前の例のものと同じで、商は

$Q = 0.11\bar{1} = 0.101_2 = 5/8$ である。

非復元除算において、もっとも重要なのは、２の補数表示の負数に簡単に展開できることである。

$$q_i = \begin{cases} 1 & 2r_{i-1} \text{ と } D \text{ が同符号} \\ \bar{1} & 2r_{i-1} \text{ と } D \text{ が異符号} \end{cases}$$

計算過程中で、余りの符号も変化するので、負の被除数 X について特別な措置は必要ない。

[例 3.7]

$X = (0.100)_2 = 1/2, D = (1.010)_2 = -3/4$ であるとする。

$r_0 = X$			0	.1	0	0	
$2r_0$		0	1	.0	0	0	set $q_1 = \bar{1}$
Add D		1	1	.0	1	0	
r_1		0	0	.0	1	0	
$2r_1$			0	.1	0	0	set $q_2 = \bar{1}$
Add D	+		1	.0	1	0	
r_2			1	.1	1	0	
$2r_2$			1	.1	0	0	set $q_3 = 1$
Add $-D$	+		0	.1	1	0	
r_3			0	.0	1	0	

最終的に、

$Q = 0.\bar{1}\bar{1}1 = 0.\bar{1}0\bar{1} = -(0.101)_2 = -5/8$。

２の補数表示は 1.011 である。最後の余りは 1/32 であって、被除数 X と同じ符号である。

余りの符号：

最終的な余りの符号と被除数の符号と等しくなることを定義する。

例えば、 5 を 3 での割る時、商 1、余り 2 を得る。商 2、余り -1 は $|R| < D$ を満たしているがこの値を取らない。

もし余りの符号が被除数と異なるならば、余りと商を訂正しなければならない。

[例 3.8]

$X = (0.101)_2 = 5/8, D = (0.110)_2 = 3/4$
とする。

$r_0 = X$			0	.1	0	1	
$2r_0$		0	1	.0	1	0	set $q_1 = 1$
Add$-D$	$+$	1	1	.0	1	0	
r_1			0	.1	0	0	
$2r_1$		0	1	.0	0	0	set $q_2 = 1$
Add$-D$	$+$	1	1	.0	1	0	
r_2			0	.0	1	0	
$2r_2$			0	.1	0	0	set $q_3 = 1$
Add$-D$	$+$		1	.0	1	0	
r_3			1	.1	1	0	

被除数が正であるのに、余り (r_3) は負である。 r_3 に D を加えること
$(1.110 + 0.110 = 0.100$ ということ$)$ で余りを訂正するべきであり、訂正した商は以下に示す通りである。

$$Q_{corrected} = Q - ulp$$

ここで、 $Q = 0.111$ であって、
$Q_{corrected} = 0.110_2 = 3/4$ である。

結果の訂正：

一般に、余りと被除数の符号が異なる時、以下のように訂正することが必要となる。

- 被除数と除数が同じ符号であるならば、余り r_m は D を加えることによって訂正され、商は ulp を引くことによって訂正される。
- もし、被除数と除数が異なる符号である時、余りは r_m から D を引くことによって、商は ulp を加えることによって訂正される。

余りが 0 の時を考える必要がある。

[例 3.9]

$X = (1.101)_2 = -3/8, D = (0.110)_2 = 3/4$ とする。この除算の正しい結果は $Q = -1/2$ で、余り 0 を持つ。

$r_0 = X$			1	.1	0	1	
$2r_0$			1	.0	1	0	set $q_1 = \bar{1}$
Add$D$	+		0	.1	1	0	
r_1			0	0	0	0	zero remainder
$2r_1$			0	.0	0	0	set $q_2 = 1$
Add$-D$	+		1	.0	1	0	
r_2			1	.0	1	0	
$2r_2$		1	0	.1	0	0	set $q_3 = \bar{1}$
Add$D$	+	0	0	.1	1	0	
r_3			1	.0	1	0	

余り r_3 と被除数 X は同じ符号であるが、
訂正ステップは必要である。なぜなら、
$-1/2$ である商の代わりに
$Q = 0.\bar{1}1\bar{1} = 0.\bar{1}01_2 = -3/8$ を得るからで
ある。したがって、途中の余り 0 の発生の
発見と余りの訂正（余り 0 を得るため）が
必要である。

$$r_3(corrected) = r_3 + D = 1.010 + 0.110 = 0.000$$

ulp を引くことにより $Q = 0.\bar{1}1\bar{1} = 0.\bar{1}01$
を訂正して、 $Q_{corrected} = 0.\bar{1}00_2 = -1/2$

3.3.1 ２の補数商の生成

非復元除算法は 1 と $\bar{1}$ の桁を使用する商を生成するため、被除数と除数との表現が一致しない。

もし、 XとD が 2 の補数表現とすると、商の 1 と $\bar{1}$ というような表現から、 2 の補数表現への変換が必要となる。

$q_i \in \{\bar{1}, 1\}\ (q_i \neq 0)$ を表現するには 1 ビットが十分であるので、 0 と 1 を用いてそれぞれ値 $\bar{1}$ と 1 を割り当てる。

変換アルゴリズム：

結果として、２進数は $(0.p_1 \cdots p_m)$ で表現される。ここで、$p_i = \frac{1}{2}(q_i + 1)$ である。この数は以下のアルゴリズムを用いて 2 の補数表現に変換できる。

Step 1. 与えられた数を 1 ビット左へシフトする。

Step 2. 最上位ビットの補をとる。

Step 3. 最下位に 1 を追加する。

このアルゴリズムの結果、以下の数列を得る。

$$(1 - p_1) \,.\, p_2 p_3 \cdots p_m 1$$

同じ値の証明：

２の補数表現として値を判断する時、上記の数列が元の商 Q と同じ値を持つことを証明する。

２の補数表現として上記の数列：

$$-(1-p_i)+\sum_{i=2}^{m} p_i 2^{-i+1}+2^{-m}$$

$p_i = \frac{1}{2}(q_i+1)$ を代入すると、

$$q_1 2^{-1}-2^{-1}+\sum_{i=2}^{m}(q_i+1)2^{-i}+2^{-m}$$

$$= q_1 2^{-1} - \left(2^{-1} - 2^{-m}\right) + \sum_{i=2}^{m} q_i 2^{-i} + \sum_{i=2}^{m} 2^{-i}$$

最後の項は $\left(2^{-1} - 2^{-m}\right)$ と等しいから、

$$= q_1 2^{-1} + \sum_{i=2}^{m} q_i 2^{-i} = \sum_{i=1}^{m} q_i 2^{-i} = Q$$

上記の変換アルゴリズムは並列にビット処理を実行する可能である。

これは、２の補数表現の時、非復元除算の各々のステップにおいて適当な商のビットを生成する。

例えば、 $X = 1.101$ と $D = 0.110$ とする除算で商 $0.\bar{1}1\bar{1}$ を生成する代わりに、
$(1 - 0).101 = 1.101$ を生成する。訂正ステップの後、 $(Q - ulp) = 1.100$ を得る。これは、 2 の補数表示において $-1/2$ の正しい表現である。

3.4 平方根演算

平方根の抽出には、従来の「完全平方」の方法は復元除算と概念的に似ている。

X を正の小数、平方根を $Q = (0.q_1q_2 \cdots q_m)$ とする。

１ステップで１ビットが求まれ、 m のステップで、 Q のすべてのビットは生成される。

中間根と余りの表現：

ステップ i において、部分的に中間根として $Q_i = \Sigma_{k-1}^{i} q_k 2^{-k}$ を用いる。従って、$Q_m = Q$。

また、ステップ i における余りを r_i とする。

一般に、その次の余りは以下の式から求められる。

$$r_i = 2r_{i-1} - q_i \cdot (2Q_{i-1} + q_i 2^{-i})$$

平方根の計算手順：

初めのステップの余りは X で、$Q_0 = 0$ である。

$$r_1 = 2r_0 - q_1(0 + q_1 2^{-1}) = 2X - q_1(0 + q_1 2^{-1})$$

復元法で平方根の桁 q_i を決定するために仮の余りを計算する。

$$2r_{i-1} - (2Q_{i-1} + 2^{-i})$$

もし、この仮の余りが正であるならば、r_i にこの値を代入し、$q_i = 1$ とする。さもなければ、$r_i = 2r_{i-1}, q_i = 0$ とする。

上記の手続きの証明：

$$
\begin{aligned}
r_m &= 2r_{m-1} - q_m(2Q_{m-1} + q_m 2^{-m}) \\
&= 2^2 r_{m-2} - 2q_{m-1}(2Q_{m-2} + q_{m-1}2^{-(m-1)}) \\
&\quad -q_m(2Q_{m-1} + q_m 2^{-m}) \\
&\quad \vdots \\
&= 2^m \cdot r_0 - 2^m[(q_1 2^{-1})^2 + (q_2 2^{-2})^2 + \cdots + (q_m 2^{-m})^2] \\
&\quad -2^m \left[2q_2 2^{-2} q_1 2^{-1} + \cdots + 2q_m 2^{-m} \sum_{i=1}^{m-1} q_i 2^{-i} \right] \\
&= 2^m X - 2^m \left(\sum_{i=1}^{m} q_i 2^i \right)^2 = 2^m (X - Q^2)
\end{aligned}
$$

2^m で割ったものが最終の余りとして期待された結果 $r_m 2^{-m}$ である。

[例 3.10]

$X = 0.1011_2 = 11/16 = 176/256$ とする。これの平方根の計算は次のように行われる。

$r_0 = X$			0	.1	0	1	1	
$2r_0$		0	1	.0	1	1	0	
$-(0 + 2^{-1})$	$-$	0	0	.1	0	0	0	
r_1		0	0	.1	1	1	0	set $q_1 = 1, Q_1 = 0.1$
$2r_1$		0	1	.1	1	0	0	
$-(2Q_1 + 2^{-2})$	$-$	0	1	.0	1	0	0	
r_2		0	0	.1	0	0	0	set $q_2 = 1, Q_2 = 0.11$
$2r_2$		0	1	.0	0	0	0	$(2Q_2 + 2^{-3}) = 1.101$ より小さい
$r_3 = r_2$		0	1	.0	0	0	0	set $q_3 = 0, Q_3 = 0.110$
$2r_3$		1	0	.0	0	0	0	まだ正数
$-(2Q_3 + 2^{-4})$	$-$	0	1	.1	0	0	1	
r_4		0	0	.0	1	1	1	set $q_4 = 1, Q_4 = 0.1101$

$Q = 0.1101_2 = 13/16$ と余り $2^{-4}r_4 = 7/256 = X - Q^2 = (176 - 169)/256$ である。

上記の手続きは 復元除算のアルゴリズムに似ている。

非復元除算のアルゴリズムに似せる方法は、次の q_i に対する選択を用いる点である。

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq 0 \\ \bar{1} & \text{if } 2r_{i-1} < 0 \end{cases} \tag{3.21}$$

[例 3.11]

$X = 0.011001_2 = 25/64$ とする。

$r_0 = X$			0	.0	1	1	0	0	1	
$2r_0$			0	.1	1	0	0	1	0	set $q_1 = 1$,
										$Q_1 = 0.1$
$-(0 + 2^{-1})$	$-$		0	.1	0	0	0	0	0	
r_1			0	.0	1	0	0	1	0	
$2r_1$			0	.1	0	0	1	0	0	set $q_2 = 1$,
										$Q_2 = 0.11$
$-(2Q_1 + 2^{-2})$	$-$	0	1	.0	1	0	0	0	0	
r_2		1	1	.0	1	0	1	0	0	
$2r_2$		1	0	.1	0	1	0	0	0	set $q_3 = \bar{1}$,
										$Q_3 = 0.11\bar{1}$
$+(2Q_2 - 2^{-3})$	$+$	0	1	.1	0	$\bar{1}$	0	0	0	
r_3		0	0	.0	0	0	0	0	0	

従って、平方根は

$Q = 0.11\bar{1} = 0.101_2 = 5/8$

結果の平方根 Q の表現は 非復元除算法で商に施した操作によって２の補数表現に変換できる。

[**第３章 終**]