# 第6章 浮動小数点演算の作法

本書の詳細はこちら：
https://www.cqpub.co.jp/hanbai/books/52/52531.htm

「浮動小数点演算」とか「浮動小数点形式」ということばを聞いたことがある方は少なくないと思います．しかし，実際に浮動小数点形式でデータを扱うシステム，あるいは回路を手がけたことのある方となると，意外に少ないのではないでしょうか．筆者も若かりしころは，浮動小数点演算と聞くだけで腰が引けてしまい，「そこは担当したくないなあ」などと思ったものです．

たしかに浮動小数点演算は固定小数点演算と比べ，なにかめんどうな感じがするのは事実です．少し知識があるハードウェア設計者であれば，「加減算がめんどうだ」，「丸めがなにやら複雑だ」，「特有の作法が良くわからない」など，避けて通るに十分な理由を数多く挙げることでしょう．

しかし，浮動小数点はごくわずかの作法を押さえておけば，それほど難しくありません．逆に，多くの恩恵を与えてくれるものです．

また，浮動小数点演算というと，「IEEE 754の規格にあるとおりに扱わなくてはならない」と信じ込んでしまい，それゆえにめんどうな感じがして敬遠している方も多いようです．しかし，規格に従わなくてはならないのは，そのフォーマットで外部とデータを共有する場合であると筆者は考えます．浮動小数点演算部が局部で閉じていて，浮動小数点形式のままで外部とデータのやり取りがなければ，かならずしも規格どおりの扱いは必要ありません．単精度や倍精度のフォーマットに従う必要もありませんし，丸めにしても必要な精度が保持されるのであれば，規格どおりでなくてもいいのです．こうした自由度により，必要最低限の回路規模で浮動小数点の恩恵にあずかれるのも，ハードウェアならではといえるでしょう．

さあ，自由な発想で浮動小数点を使い倒しましょう．

## 6-1 浮動小数点形式とは

まず，浮動小数点形式を理解するところから始めましょう．

10進法において，例えば$123456_{10}$を以下のように表記することがあるのは，よくご存じだと思います．

$$123456 = 1.23456 \times 10^5 \quad \cdots\cdots (6\text{-}1)$$

これは科学記数法（scientific notion）と呼ばれる表記法です．詳細に表現すると次式になります．

$$123456 = (1 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} + 5 \times 10^{-4} + 6 \times 10^{-5}) \times 10^5 \quad \cdots\cdots (6\text{-}2)$$

これがまさに10進法による浮動小数点表現になります．式での表現で理解しにくければ，**図6-1**を参



見本

照してください．最上位のけたをかならず$10^0$に固定し，実際にその数がもつ大きさを$10^5$として分離して表現しています．結果的に小数点の位置が移動しているのがわかります．とくに難しいことをやっているわけではないことがわかるでしょう．

### ● 仮数部の最上位けたをかならず‘1’とする正規化数を用いる

10進法と同じような考えで，2進法の場合も例えば次式のように表現できます．

$$101101_2 = (1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}) \times 2^5 \quad \cdots\cdots\cdots\cdots (6\text{-}3)$$

式(6-3)を**図6-2**に図示します．式(6-3)において，カッコの中の数を仮数部(significand)，カッコの右外にある2の指数を指数部(exponent)と呼びます．これに符号(sign)を付けてやれば，2進法による浮動小数点形式は基本的に完成です．

仮数部を$n$ビットとして一般化した2進数$a$は，

$$a = (-1) \times a_s \times (a_{n-1} \times 2^0 + a_{n-2} \times 2^{-1} + \cdots + a_1 \times 2^{-(n-2)} + a_0 \times 2^{-(n-1)}) \times 2^e \quad \cdots\cdots\cdots\cdots (6\text{-}4)$$

となります($a_s$は符号を，$e$は指数部を表す)．

しかし，これでは不十分な点があります．例えば，$n = 8$として，$a$の仮数部が$00010110_2$であったとしましょう．これでは，仮数部が8ビットであるにもかかわらず，実際の有効けたは5ビット(上位3ビットは‘0’が詰まっているため)であり，せっかくの浮動小数点形式が生かされているとはいえません．浮動小数点形式において仮数部が$n$ビットならば，有効けたも$n$ビットでなくてはなりません．したがって先の例において，$a$の仮数部は最上位のけたが‘1’となるように3ビット分シフト・アップして$10110xxx_2$と表現し，指数部を－3します(**図6-3**)．xxxのけたにはしかるべき値が入って，8ビット分の有効けたとなります．これにより，式(6-4)は次のように書き換えられます．

$$a = (-1) \times a_s \times (1 \times 2^0 + a_{n-2} \times 2^{-1} + \cdots + a_1 \times 2^{-(n-2)} + a_0 \times 2^{-(n-1)}) \times 2^e \quad \cdots\cdots\cdots\cdots (6\text{-}5)$$

式(6-5)では，$2^0$のけたはかならず‘1’となります．このような表現形式を正規化数(normalized number)と呼びます．$00010110_2$のように仮数部の先頭に‘0’が入っているものは，正規化数ではあり

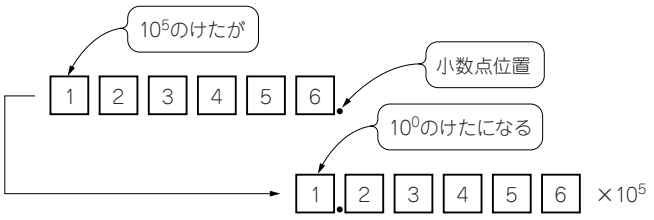



**図6-1**
**10進法の浮動小数点表示**
最上位のけたをかならず$10^0$に固定し，実際にその数がもつ大きさを$10^5$として分離して表現している

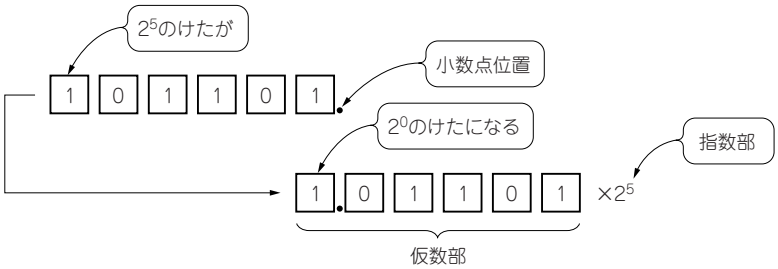



**図6-2　2進法の浮動小数点表示**
式(6-3)において，カッコの中の数を仮数部(significand)，カッコの右外にある2の指数を指数部(exponent)と呼ぶ



見本

ません．

さらに，係数$a_{n-1}$がつねに‘1’であるという前提であれば，これを「暗黙の‘1’」とし，表現に加える必要がなくなってきます．そこで，表現からは削除した表現形式が用いられます（仮数部の先頭の‘1’が実際になくなってしまうわけではなく，表示から消えるだけなので，誤解のないように）．実際の浮動小数点形式の例は後ほど説明することにして，まずその恩恵について述べます．

### ● 浮動小数点演算の最大の恩恵は演算結果の有効けたを保持できること

浮動小数点演算はどのようなときに便利なのかについて考えてみましょう．

第4章において，除数，被除数ともに正規化しておくことで，余分なけたの演算を回避できることを説明しました．これだけでも十分なのですが，浮動小数点演算のもたらす恩恵はこんなものではありません．

例えば，254 ÷ 255 × 255という演算を行うとします．最初の除算において整数部のみを求めると，商$q = 0$となるのは明らかです（このとき，小数第1位を求めて四捨五入するという操作は行わない）．そのため，この商に255を掛けたとしても最終的には0となります．しかし，255の乗算を先に行った後に除算を行うと，結果が254となることは説明の必要もないでしょう．このように，固定小数点演算では演算手順を変えると答えが変わってしまいます．「最初の除算で小数部を考慮しないからでしょう？」という声が聞こえてきそうですね．しかし，

$(a \div b) \times c = (a \times c) \div b$（$a$，$b$，$c$は8ビット）……(6-6)

が成立するには，式(6-6)の左辺において最低でも8ビットの小数部を求める必要があります．そうすると，商は整数部をあわせて16ビットのけたとなり，乗算後は24ビットのけたとなります．求めたいのが254であったとすれば，ずいぶんむだなけたが費やされることになります．

これに対し，浮動小数点演算において最終的に欲しい有効けたが8ビットであるとすれば，8ビットのほかに精度を確保するための数ビットを加えればよいのです．これによって回路規模の抑制や速度の向上につながる面もありますが，なんといっても最大の恩恵は有効けたを保持できるということです．乗除算が連続するような場合で有効けたが重要な場合，非常に効果的といえます．精度を保つことに注意しつつ，最終的な演算結果に要求する有効けたに合わせてフォーマットを決めてしまえば，先ほどの演算手順の問題からは開放されます．またフォーマットを統一してしまえば，演算回路そのものはまっ

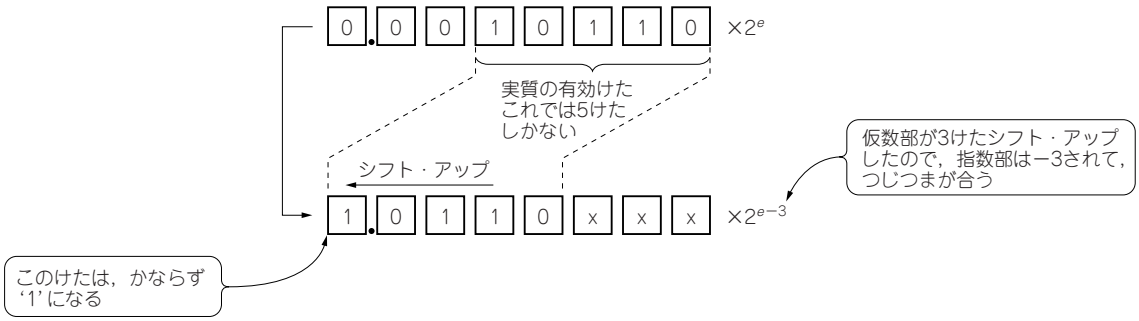



**図6-3　仮数部の正規化**
いちばん左にある最初の‘1’を左端のけたとなるようにシフト・アップし，下位にはしかるべき値を入れる



見本

たく同一のものを利用できます．

例えば，コンピュータ・グラフィックスなどで多用される座標変換に，透視変換というものがあります．そこで使用される座標の変換式は，2点透視変換の場合で式(6-7)のようになります．

$$\begin{cases} x' = \dfrac{a_x x + b_x y + c_x}{dx + ey + f} \\ y' = \dfrac{a_y x + b_y y + c_y}{dx + ey + f} \end{cases} \quad \cdots\cdots (6\text{-}7)$$

$x$，$y$は表示される直交座標値，$x'$，$y'$はレンダリングされるテクスチャの直交座標値です．いずれも有限な値で，多くの場合同等の大きさ(広さ)をもっています．

これがどういう意味をもつかといえば，分母，分子ともに元の$x$，$y$からかけ離れた値となったとしても，除算を実行するとおよそ元の範囲に値が戻ってくるということです．つまり，分母と分子の比は，特定の有限な範囲にあるということになります．

このような場合に固定小数点演算を採用すると，まず分母と分子が元の$x$，$y$の範囲から一度大きく離れてしまいます．除算後に元の範囲に戻るような係数が与えられたときは，非常に多くのけたがむだな演算に費やされることになります．これに対して，浮動小数点はこうしたケースではきわめて有効です．最終的に$x'$，$y'$に必要なけた数を確保できるような浮動小数点フォーマットを定義して，これですべて演算すれば，有効けたは保持され，むだなけたの演算は必要ありません．

一方，良いことばかりではなく，めんどうなことも伴います．加減算がその典型で，数のけた(大きさ)を示す部分を指数部として分離しているので，加減算を行う場合はけたを合わせるという操作が必要になります．また，固定小数点でデータを受け渡す場合，データの変換が必要になります．

おおざっぱに浮動小数点のメリット，デメリットについて説明しました．固定小数点と浮動小数点の住み分けの例は第7章で解説します．

### ● 効率的な浮動小数点演算を行うためのフォーマット「IEEE 754」

浮動小数点形式の具体例として，IEEE 754に取り決められたフォーマットを見てみましょう．**図6-4**に単精度と倍精度のフォーマットを示します．ここでは，単精度を例にとり説明します．

まず，仮数部には暗黙の‘1’は含まれていませんので，この1ビットを加えて実質24ビットになります(**図6-5**)．したがって，暗黙の‘1’を加えた仮数部がとる値の範囲は1≦仮数部＜2になります．仮数部の最小位ビットをulp(unit in the last place)と呼び，演算の精度を表す単位として用いられます．固定小数点ではLSB(least significant bit)は決められた特定の大きさを示しますが，浮動小数点では指数部によって仮数部の最小位ビットの大きさが変化します．あくまで有効けたの最小位を単位として精度を表し，実際の大きさは問いません．

指数部は，－126≦指数部≦127の範囲をとることになっています．しかし，表現形式は2の補数ではなく，ゲタばき表現(biased representation)といわれる127のゲタ(bias)[注6-1]を加えた形で，符号を伴わない形式になっています．したがって127を指数部から引くことで，2の補数に変換されます．

なぜこのような形式を採用しているかといえば，符号を伴わない形式をとることで，整数と同じよう

注6-1：倍精度の場合，ゲタは1023.



見本

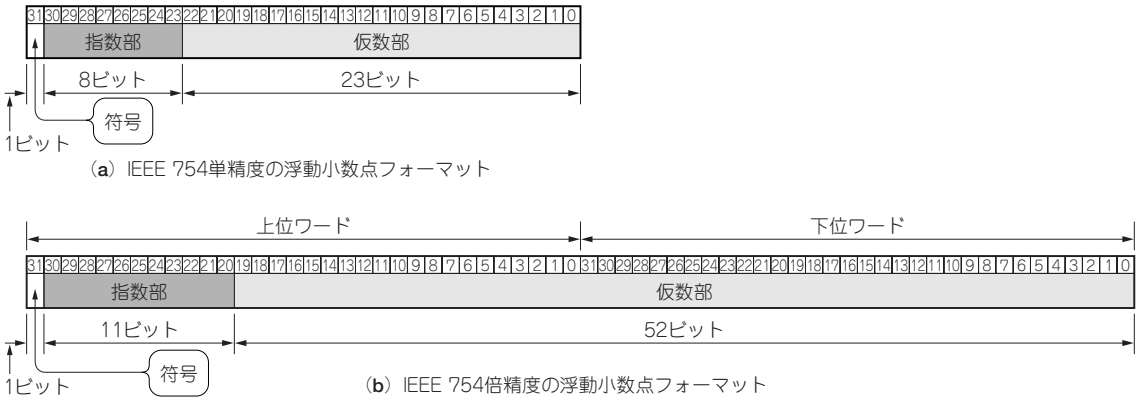



**図6-4　浮動小数点形式**
(a) に単精度の，(b) に倍精度のIEEE 754浮動小数点フォーマットを示す

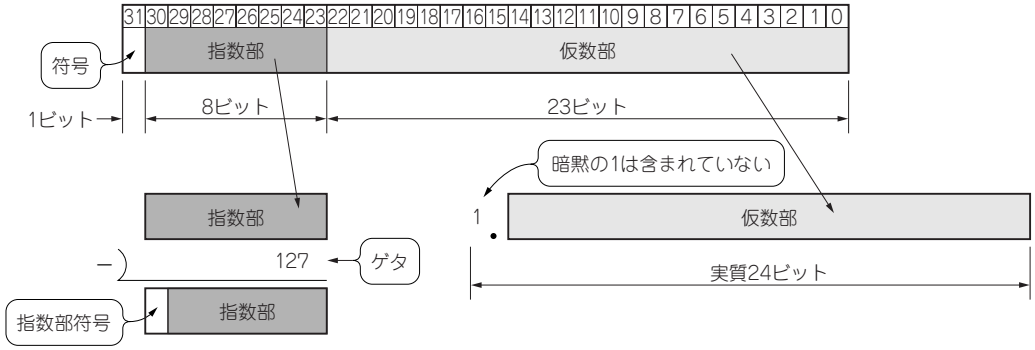



**図6-5　IEEE 754単精度の浮動小数点フォーマット**
単精度の浮動小数点において仮数部は，暗黙の‘1’を加えて実質24ビットの仮数部となり，指数部は＋127のゲタばき表現となる

な比較演算で大小比較を行えるからです．指数部が符号付きであるとすれば，指数部と仮数部は分離して考えなければなりません．つまり，整数比較演算では大小が判定できず，浮動小数点専用の比較演算が必要になります．そもそもIEEE 754はCPU演算を前提として考えられていて，いかに単純な演算で効率化を図るかが考慮されています．

### ● 仮数部や指数部が‘0’のときには注意が必要

ところで，指数部は整数で8ビットなので256状態をとりますが，先の範囲では254状態しか定義されていません．その理由は，－127と128（ゲタの127を加えれば0と255）は予約語（特定の状態の割り当てに使用）となっていて，通常の数として使用してはいけないことになっているからです．指数部が－127と128の場合，仮数部の状態によって扱いが異なります（**表6-1**）．

**表6-1**から，暗黙の‘1’を付けてはならない場合が存在することがわかります．例えば，ゲタばきの指数部が‘0’で仮数部が‘0’のときは0を表現しているので，暗黙の‘1’はあってはなりません．この場合は24ビット目は‘0’になります．また，ゲタばきの指数部が‘0’で仮数部が‘0’でないときは，‘0’より大きく，正規化された最小値（絶対値）よりも小さい状態を表しています．これを不正規化数（de-



見本

表 6-1　指数部と仮数部の状態割り当て

| 単精度 | | 倍精度 | | 意　味 |
|---|---|---|---|---|
| 指数部（＋ゲタ） | 仮数部 | 指数部（＋ゲタ） | 仮数部 | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | ≠ 0 | 0 | ≠ 0 | ±不正規化数 |
| 1 ～ 254 | 任意 | 1 ～ 2046 | 任意 | ±浮動小数点数（正規化数） |
| 255 | 0 | 2047 | 0 | ±無限大 |
| 255 | ≠ 0 | 2047 | ≠ 0 | NaN（Not a Number：非数） |

normalized number）といいます．単精度の場合では，

- 正規化数の最小値　：$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}$
- 不正規化数の最小値：$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_2 \times 2^{-126}$

となります．したがって，この場合も暗黙の ‘1’ を付けてはならないことになります．以上のことから，指数部が ‘0’ のときには暗黙の ‘1’ は不要になります．

正規化数の最小値の仮数部を右シフトして順次値を小さくしていき，‘0’ に至るまでの状態を「漸進アンダフロー（gradual underflow）」といいます．この状態にある数が不正規化数です．不正規化数は，ハードウェアを扱う設計者にとって悩みの種です．例えば，除算においては正規化されていることを前提に多くの恩恵を受けたことを思い出してください．不正規化数はその前提を崩してしまうわけで，非常にありがたくない状態です．また，仮数部の上位の何けたかに ‘0’ が詰まっているということは，すでに有効けたが不足している状態なので，精度の点から好ましくなく，例外として扱うべき状態にあるわけです．NaN（Not a Number；非数）や無限大のようなあきらめのつく状態ではなく，不正規化数は「数らしき状態」にあるので，実に扱いづらいものであるといえます．

ハードウェアでは，このような不正規化数の扱いは放棄してしまい，演算結果が不正規化数となった場合には例外を発生させて，ソフトウェアでなんらかの処理を行うというような扱いが多いようです．基本的に不正規化数はハードウェアでは扱わないことにするほうが，回路が複雑にならず現実的であるといえます．

無限大は，0 以外の数を ‘0’ で割った場合などに使用される予約語になります．‘0’ による除算を行った際に例外を発生させるのではなく，入力条件により予約語を割り当ててしまうのです．

NaN は基本的に無効な演算の結果を表す予約語です．‘0’ を ‘0’ で除算した場合や無限大から無限大を引いた場合などがそれにあたります．

### ● 浮動小数点における四つの丸めモード

固定小数点では必要なけたを確保することで，かならず精度を保つことができます．しかし，浮動小数点は仮数部による有限なけたで数を表現して演算精度を保たなければなりません．そのため，固定小数点にはない若干うるさい作法が必要となります（これが，浮動小数点が嫌われる原因の一つでもあるのだが…）．その最たるものが丸めです．

浮動小数点演算の結果の多くは入力よりもけたが増えるので，入力の形式に戻すにはなんらかの丸めが必要になります．IEEE 754 では次に示す四つの丸めモードが規定されています．



見本

- 切り上げ（＋∞方向への丸め）
- 切り下げ（－∞方向への丸め）
- 切り捨て（0方向への丸め）
- 最近点への丸め

最初の三つのモードはそれほど違和感がないと思います．

最後の最近点への丸めも0捨1入にきわめて近い考えかたなのですが，若干の違いがあります．ulpより下位のけたが1/2ulpより大きければ切り上げ，小さければ切り捨てるところまでは同じです．ただし，1/2ulpに等しいときの操作がulpの値によって異なります．つまり，ulp = 1の場合は切り上げ，ulp = 0の場合は切り捨てになります（0捨1入では，ulpより下位のけたが1/2ulp以上であれば切り上げとなる）．この操作により，ulpより下位のけたが1/2ulpに等しいときには，丸めた結果がかならず偶数になります．これを「最近偶数への丸め（round even）」と呼び，最近点への丸めモードの特徴的な操作となります．名まえに惑わされて，「丸めの後はかならず偶数になる」と勘違いしないようにしてください．このような操作を行う理由は，0捨1入で丸めを行うと，ulpより下位のけたが1/2ulpに等しい場合にulpの中点を指しているにもかかわらず切り上げとなり，積算などを繰り返したときに＋（プラス）方向へ偏るのを防ぐためです．ulpの‘0’と‘1’の生起確率が等しいという前提で，切り捨てと切り上げを等確率で得ようというわけです．

### ● 演算精度を保つためにGビット，Rビット，Sビットを用意

最近点への丸めモードを実施するにあたり，ulpより下位の大きさを判断する以下のようなビットが厳密に定義されています（**図6-6**）．

- **ガード・ビット（guard bit．以下Gビットとする）**――Gビットは1/2ulpの重みをもつビット．先に述べたulpより下位のけたの1/2ulpに対する大小を決める境界の役目をもっている
- **ラウンド・ビット（round bit．以下Rビットとする）**――Rビットは1/4ulpの重みをもつビット．ここでは準Gビット的な役割と理解すればよい
- **スティッキ・ビット（sticky bit．以下Sビットとする）**――Sビットは，それ以下のビットのORをとった値．ulpより下位のけたが1/2ulpに等しいか否かを判定するのに用いられる

これらのビットの役割を，浮動小数点の減算を例に説明することにしましょう．

$1.0001\ 0101 \times 2^{0} - 1.0101\ 1011 \times 2^{-4}$を考えてみましょう．その手順を**図6-7**に示します．すでに述べたように，浮動小数点においては加減算の場合はけたを合わせてから演算を実行します．けたを合わ

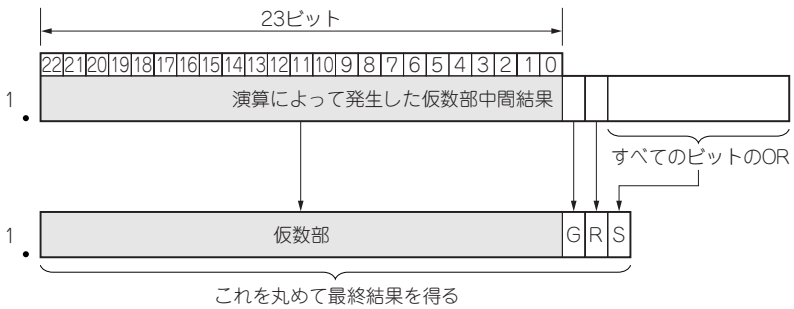



**図6-6**
**G，R，Sビットのけた位置**
Gビット，Rビット，Sビットは図に示すけた位置，および意味を持っている

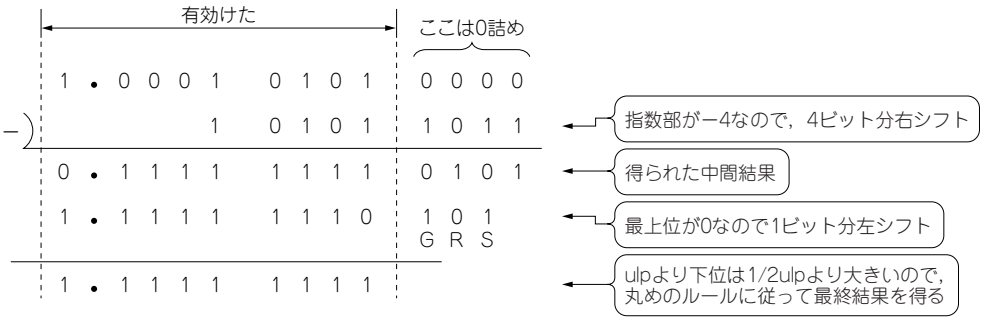


見本

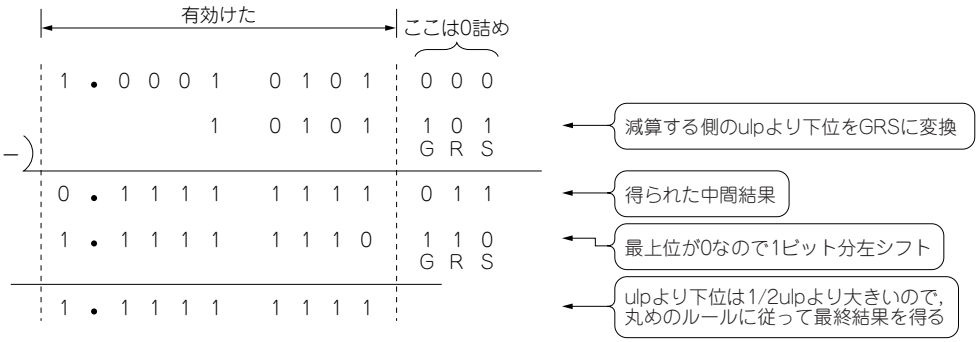
せるため，減算する側の仮数部1.0101 1011を4ビット右シフトします．これを前シフト（pre-shift）と呼びます．前シフト結果をそのまま演算した例が**図6-7**（**a**）になります．一方，**図6-7**（**b**）は前シフトした結果を**図6-6**の形（ulpより下位をG，R，Sで表現）に変換してから演算した例になります．

前シフトした結果を**図6-6**の形式に変換する理由は，前シフトの結果をそのまま用いると，無用に大きなけた（最大2倍のけた）の演算を必要とするからです（**図6-7**では4ビットしか前シフトしていないので，ありがたみが薄いのだが…）．ここで重要なのは，G，R，Sの各ビットは精度を保つために用意されたので，その演算結果は「無限大のけたをもつ演算器で得た結果を丸めたのと同一の結果が得られなければならない」ということです．ここでは，**図6-7**（**a**）がその例と考えてください．

こうした観点で演算の過程を見ていきましょう．**図6-7**（**a**），（**b**）いずれの場合も減算後は最上位が‘0’になるので，正規化が必要になります．したがって，中間結果を1ビット左シフトして最上位に‘1’が現れるようにします．ところが，Gビットになるはずだったけたが左にシフトしたことによって有効けたの中に入ってしまい，Rビットとなる予定だったけたはGビットに格上げされています．かりにRビットがなかった場合，Sビットが代わりに格上げされます．**図6-7**（**b**）の場合，そのビットは前シフトの後の特定のビット以下をORしたものなので，1/4ulpあるいは1/8ulpといった算術的重みをもったものではありません．あくまでGビットと組み合わせて「1/2ulpちょうどではない」という判定をするためにしか使えないものです．したがって，格上げされてGビットの代役を務めるには適当ではありません．




（**a**）右シフト結果をそのまま使った減算




（**b**）右シフト結果のulpより下位をGRSに変換してから減算

**図6-7　G，R，Sビットの役割**



見本

**表6-2　最近点への丸めモード**（G，R，Sビットによる操作）
G & ( ulp | R | S )をulpに加算することで丸めが完了する

| ulp | Gビット | Rビット | Sビット | 丸め操作 | 備　考 |
|---|---|---|---|---|---|
| 0 | 0 | − | − | none | − |
| 0 | 1 | 0 | 0 | none | 最近偶数への丸め |
| 0 | 1 | 0 | 1 | +1ulp | − |
| 0 | 1 | 1 | − | +1ulp | − |
| 1 | 0 | − | − | none | − |
| 1 | 1 | 0 | 0 | +1ulp | 最近偶数への丸め |
| 1 | 1 | 0 | 1 | +1ulp | − |
| 1 | 1 | 1 | − | +1ulp | − |

そこで，このような場合にGビットの代役を務められる，1/4ulpの重みをもったRビットが必要になります．

先に，GビットやSビットが明確な役割をもっていたのに対し，Rビットは「準Gビット的な役割」とだけ述べましたが，実はこのような重要な役割を果たしています．

なお，先の例において減算する側の指数部が減算される側と同じであった場合は，$-0.01000110\times2^{0}$となって有効けたが失われたかのように見えます．しかし，このような演算の場合は，固定小数点で行っても結果に変わりはありません．また，けたを合わせる必要がないので前シフトの必要がありません．そのため，中間結果にも本来のけたより下位のけたは現れません．当然，中間結果に対する丸めそのものが不要なので，先の例と同じに考えてはいけません．

最近点への丸めモードは，浮動小数点演算では精度を保持するための重要な手段です．G，R，Sのそれぞれのビットによる丸めを**表6-2**にまとめておきます．

## 6-2　固定小数点と浮動小数点の変換

実際の浮動小数点演算に進む前に，固定小数点と浮動小数点の変換について学んでおきましょう．というのも，数値演算結果を正規化する際に，この変換で用いられる要素がたびたび登場するので，先に知っておいたほうが便利だからです．

### ● 固定小数点から浮動小数点への変換回路を作る

まずは，固定小数点から浮動小数点への変換回路です．32ビットの固定小数点形式を単精度の浮動小数点形式に変換してみましょう．この変換を行うには，次の手順が必要になります．

**1）固定小数点の数値を絶対値化する．このとき，符号が確定**
**2）左側のけた（最上位側）から見て，最初に‘1’が立っているビットを探す**
**3）そのビットがMSB（most significant bit）となるようにシフト（正規化）する．このときシフト量によって仮の指数部が決まり，シフト結果によって仮の仮数部が決まる**
**4）仮の仮数部の不要となるけたを丸めて仮数部が確定する．丸めの際，けた上がりが生じたら，仮の指数部を＋1し，そうでなければそのままとして指数部が確定する**

では，**リスト6-1**のRTL記述をベースに説明することにしましょう．**リスト6-1**では，入力`fix`の小



見本

数点位置をパラメータ Frac で定義しています．ここでは Frac = 16 としてあります．小数点の位置が fix の 32 ビットの中にない場合（fix がとても大きいあるいは小さい場合）は，32 以上あるいは負の値をとってもかまいません．パラメータ Ofst は，入力 fix の小数点位置が LSB の下だったと仮定し，その小数点を MSB の下に移動した場合のオフセットを定義しています．このオフセットを最初に指数部に加えてやれば（あとで加算してもよいが），fix の小数点位置は MSB の下にあると考えてシフトを開始できます．さらに，ゲタばきの 127 もついでに加えてしまいます．

まず，**1**）の絶対値化を実行します．入力 fix の MSB が符号なので，これを条件として 2 の補数をとります．

次に，**2**）の「最初の '1' を探す」ことになりますが，一般にはプライオリティ・エンコーダが使われるとされています．ここでは，よりわかりやすい回路を目ざし，この探す操作と **3**）のシフト操作を，2 のべき乗単位で同時に行うことにしましょう．そのようすを p.182 の**図 6-8** に示します．最初に上位 16 ビットに着目し，その中に '1' が立っているビットがあるかどうかを判定します．これは上位 16 ビットの OR 演算で成立します．上位 16 ビットに '1' があればそのまま，なければ（すべてのビットが '0' なら）左方向に 16 ビット分シフトします．そのシフト量を仮の指数部の 4 ビット目（expo_temp[4]）とします．

**リスト 6-1　固定小数点から浮動小数点への変換回路**

```
module fx2fla (
    fl_sign, // 符号
    fl_expo, // 指数部オフセット
    fl_sgnf, // 仮数部

    fix );   // 固定小数点位置

    input     [31:00] fix      ;

    output            fl_sign  ;
    output    [07:00] fl_expo  ;
    output    [22:00] fl_sgnf  ;

    parameter Frac    = 16     ;
    parameter Ofst    = 31+127 ;

// function（演算処理部）
    wire          sign = fix[31] ;
    wire   [04:00] expo_temp      ;
    wire   [31:00] sgnf_temp5    = ( fix[31] )? ~fix + 1'b1 :fix ;

    assign         expo_temp[04] = ~|sgnf_temp5[31:16] ;
    wire   [31:00] sgnf_temp4    = ( expo_temp[04] )? {sgnf_temp5[15:00],16'h0000 } :sgnf_temp5 ;

    assign         expo_temp[03] = ~|sgnf_temp4[31:24] ;
    wire   [31:00] sgnf_temp3    = ( expo_temp[03] )? {sgnf_temp4[23:00], 8'h00 }   :sgnf_temp4 ;

    assign         expo_temp[02] = ~|sgnf_temp3[31:28] ;
    wire   [31:00] sgnf_temp2    = ( expo_temp[02] )? {sgnf_temp3[27:00], 4'h0 }    :sgnf_temp3 ;

    assign         expo_temp[01] = ~|sgnf_temp2[31:30] ;
    wire   [31:00] sgnf_temp1    = ( expo_temp[01] )? {sgnf_temp2[29:00], 2'h0 }    :sgnf_temp2 ;

    assign         expo_temp[00] = ~|sgnf_temp1[31] ;
    wire   [31:00] sgnf_temp     = ( expo_temp[00] )? {sgnf_temp1[30:00], 1'h0 }    :sgnf_temp1 ;

/*
```



見本