Chapter 7

浮動小数点演算回路を極める

科学技術計算ではよく浮動小数点が使われます．浮動小数点は表現できる数値の範囲が広く，その範囲において一定の精度で表現できるので科学技術計算に向いているためです．浮動小数点の規格としてはIEEE 754が策定されています．ここでは，このIEEE 754にのっとった加算，減算，積算，除算などの演算と，浮動小数点演算を実現する回路について説明します．

7-1 単精度浮動小数点数の構造を理解しよう

単精度浮動小数点数は32ビット幅であり，符号1ビット，指数8ビット，仮数23ビットで数字を表します．それぞれのビットの意味は次のようになります（**図7-1**）．

1）符号（sign）

符号ビットが‘1’であれば負数を，‘0’であれば正数を表します．

2）指数（exponent）

指数は8ビットの正数であり，1～254の値が使われます（整数ではないことに注意）．そして，この値から127（オフセット）を引いた数が2^eのeの値を表します．つまり，－126～＋127の指数を表します．例えば，指数の値が1であれば2^{-126}を，127であれば2^0を，254であれば2^{+127}を表します．

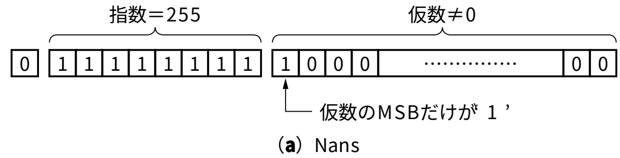
指数の値が0と255の場合は特別な意味を持っています．これらの指数と仮数の組み合わせによって意味が異なります．**表7-1**に示すように，指数の値が255で仮数が0でない場合をNaN（Not a Number；非数）と言います．これは，0×∞，＋∞－∞，除数が0の場合に発生します．NaNには，NansとNanqの2種類があります（**図7-2**）．Nansは演算の結果発生したNaNのことで，仮数のMSB（most significant bit）だけが‘1’の場合を言います．Nanqは仮数のMSBから2ビットが‘1’の場合であり，プログラムで意図的に入れられたNaNです．

図7-1
単精度浮動小数点数の構造
単精度浮動小数点は32ビット幅であり，符号1ビット，指数8ビット，仮数23ビットで数字を表す．

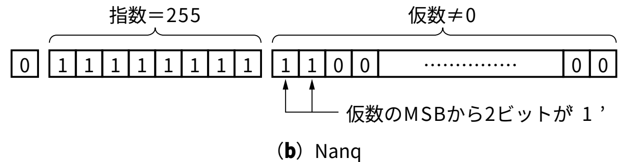
見本

表7-1
指数が0または255の場合の意味

指　数	仮　数	意　味
0	0	±0
0	0ではない	±0.xxxx…x×2^{-126}（不正規化数．グランデュアル・アンダフローと言う）
255	0	±∞
255	0ではない	NaN（「ナン」と読む）．符号は通常＋

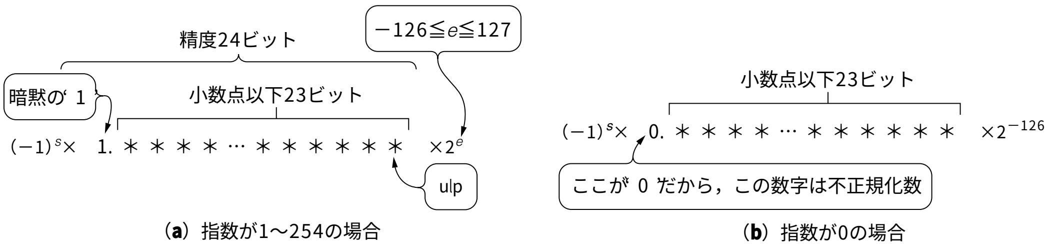


（**a**）Nans



（**b**）Nanq

図7-2
NansとNanq
NaNが発生するということは，演算になんらかのエラーが生じていること意味する．



（**a**）指数が1〜254の場合　（**b**）指数が0の場合

図7-3　仮数の精度
浮動小数点数の指数の値が1〜254の場合，暗黙の‘1’と小数点以下23ビットで，その精度は24ビットになる．

3）仮数（mantissa；マンティッサ）

仮数は23ビットの数です．浮動小数点数の指数の値が1〜254の場合，小数点より上の数はかならず‘1’になります．この‘1’を省いたものが仮数となります（**図7-3（a）**）．つまり，仮数の23ビットは小数点以下の数だけを表すことになり，精度は24ビット（暗黙の1＋23ビット）になります．

指数の値が0のときは，2^{-126}固定で，仮数は小数点以下の数を表します（**図7-3（b）**）．

また，仮数の最下位ビットは「ulp（unit in the last place）」と言います．ulpは，浮動小数点数の演算精度を表すために用いられます．

指数が正数である理由

指数は整数（integer）ではなく，オフセットを持った正数で表されることは前述しました．数値演算では，負数を表すときに2の補数を使うよりも1の補数を使ったほうが動作速度は速くなります．単精度浮動小数点では127のオフセットを使うことにより，1の補数を使ったときと同等の動作速度を実現

見本

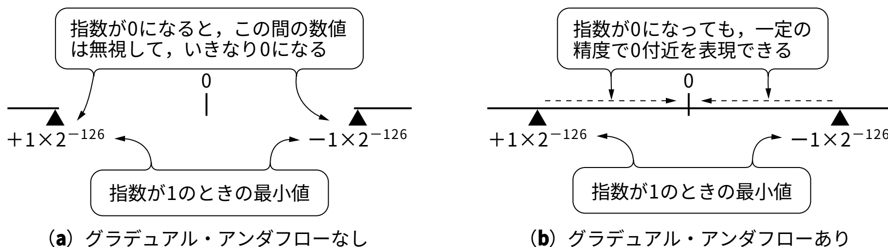


図7-4　グラデュアル・アンダフロー

グラデュアル・アンダフローを使うと，小さな数の最小区間の大きさを一定に保てる反面，ビット落ちによって有効数字が少なくなり，計算値として信用できなくなる．

することができます．これは，積算を行うときに指数どうしを加算しますが，整数どうしの加算の場合は符号拡張が必要になるのに対して，正数どうしの加算では符号拡張を行う必要がないからです．

また，オフセットが128でなく127であると，ほんの少し便利なことがあります．指数の減算と加算において，以下に示すようにLSB（least significant bit）の周りの回路を同じにすることができます．

a－b＋00000001（減算）……………………………………………………………………(7-1)
a＋b＋10000001（加算）……………………………………………………………………(7-2)

このように，IEEE 754では速度優先のしくみが随所に見られます．そして，譲れないところは完ぺきに守るという部分もあります．

一般の整数演算のときでも，整数を「正数＋オフセット」に変換してから，演算の最後でもう一度整数に戻すという演算方法があります．これは定数加算方法と言いますが，加算，減算，積算の高速化に有効です．

グラデュアル・アンダフローのじょうずな使いかた

指数＝0，仮数≠0の場合，IEEE 754では不正規化数（denormalized number）という数を扱うことになります．このとき，指数は－126に固定され，仮数の小数点より上の部分は0になります（**図7-3（b）**）．数値が小さくなるごとに最上位の‘1’は右へ移動していき，最終的に0となります．これをグラデュアル・アンダフローと言います．

グラデュアル・アンダフローには良い面と悪い面があります．まず良い面は，グラデュアル・アンダフローを使うと，小さな数について最小区間の大きさを一定に保てることです．指数が1のときの精度はulpのところで2^{-149}になりますが，グラデュアル・アンダフローを使わない場合，指数が0になるといきなりその値は0となり，0の近辺で大きな精度のギャップが発生してしまいます．これに対して，グラデュアル・アンダフローを使えば，一定精度の2^{-149}で0近辺を記述することができます（**図7-4**）．

一方，悪い面を挙げれば，グラデュアル・アンダフローを起こすと，必然的にビット落ちが発生することです．グラデュアル・アンダフローが起こった数は，有効数字が少なくなっている可能性があり，計算値としては信用できないかもしれません．科学技術計算は精度が命です．精度を保証できない計算というものは，致命的であるといえます．

グラデュアル・アンダフローを使うと，計算をしぶとく続行することができます．しかし，その結果として精度が信頼できなくなると，不ぐあいが起こります．ですから，最初のプログラム実行で‘0’

見本

表7-2 IEEE 754の丸めの法則

ulp	Gビット	Gビットより下位	丸　め
x	0	xxxxxxxx	0.1ulp未満→切り捨て
0	1	00000000	0.1ulp→切り捨て（ラウンド・イーブン）
1	1	00000000	0.1ulp→切り上げ（ラウンド・イーブン）
x	1	xxxxxxxx	0.1ulp超え→切り上げ

表7-3 GビットとRビットを使って丸めを表現

ulp	Gビット	Rビット	丸　め
0	0	0	―
0	0	1	―
0	1	0	―
0	1	1	＋1：切り上げ
1	0	0	―
1	0	1	―
1	1	0	＋1：切り上げ
1	1	1	＋1：切り上げ

が発生する演算を見つけ，けた合わせをするためにグラデュアル・アンダフローを使い，デバッグが終わったらグラデュアル・アンダフローはディセーブルにする，というような使いかたがよいのかもしれません．

丸め（rounding）の法則

有効数字のビット数が24ビットの範囲からあふれた場合は，「丸め」という作業が行われます．丸めとは，10進法の四捨五入に相当するものですが，IEEE 754ではラウンド・イーブン（round even）という独特の手法が用いられます．

表7-2のように，ulpの下位が0.1ulpより小さい場合は切り捨て，0.1ulpより大きい場合は切り上げます（0.1ulpということは，ulpから1ビット下位に‘1’が存在しているといえる）．0.1ulpと等しければ，ulpが‘0’の場合は切り捨て，‘1’の場合は切り上げます．このようにして丸めを行うと，その結果は偶数になります．そのため，これを「ラウンド・イーブン（丸めて偶数）」と言うわけです．

ulpの一つ下位のビットをG（guard）ビットと言います．また，Gビットより下位の値をすべてORしたものをR（round）ビットと言います（Rビットによって，Gビットより下位に何か値があるかどうかがわかる）．Rビットは，ulpから数えて二つ下位のビットにすべて丸めているのと同じことになります．つまり，Rビットは0.01ulp（ulpから2ビット下位），Gビットは0.1ulp（ulpから1ビット下位）と考えて，数学的にまったく問題がありません．

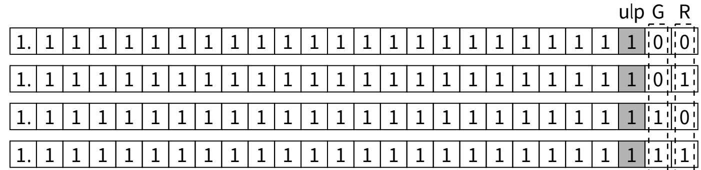
GビットとRビットの二つを使って，丸めについてまとめると**表7-3**のようになります．**表7-3**を論理圧縮すると式(7-3)になります．

G and (ulp or R) ……………………………………………………………………………(7-3)

つまり，Gビットが‘1’，かつ，ulpまたはRビットが‘1’であれば切り上げになります．

丸めを行うと，仮数に＋1を加算するということが起こりえます．このとき，仮数のけた数が増えてしまうかもしれません．**図7-5**に示すように，丸めによってけた上がりが起きたときは，かならず仮数がすべて‘0’になります．浮動小数点数では仮数は正数しか扱わないので，丸めとしての演算は＋1だけです．正負の数を扱うことによって＋1と－1の丸めが必要になるかもしれないという事態を，ここではうまく回避しています．

見本



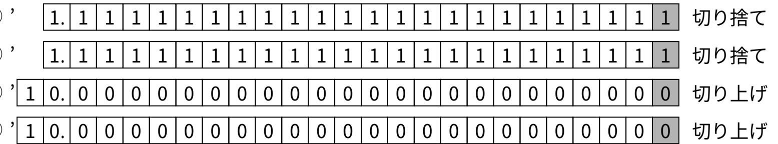


図7-5
丸めによるけた上がり
①～④のそれぞれの数値について丸めを行う．式(7-3)を利用して丸めを行うと，①と②では切り捨てが起こり，③と④では切り上げが起こる．③と④については，仮数+1となり，けた上がりが発生する．

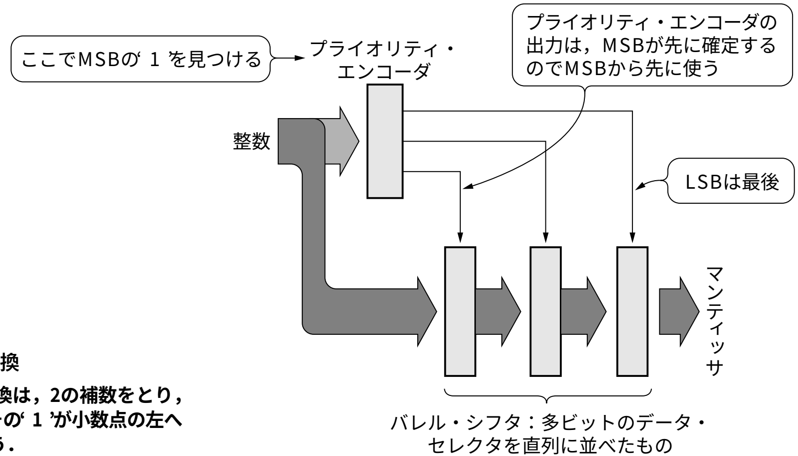


図7-6
整数から浮動小数点への変換
整数から浮動小数点数への変換は，2の補数をとり，最上位の‘1’の位を見つけ，その‘1’が小数点の左へ来るように調整し，丸めを行う．

7-2 浮動小数点演算のための部品集め

整数から浮動小数点数への変換は，次の四つのステップからなります(**図7-6**)．

1) 整数を正数に変換する(絶対値をとる作業)．
2) 最上位の‘1’の位を見つける(浮動小数点数に変換したときの指数の値を求める作業)．ここではプライオリティ・エンコーダが使われる．このとき見つかった指数が仮の指数になる．
3) 最上位の‘1’が小数点の左に来るようにバレル・シフタでそろえる．
4) 丸めを行う．このときけた上がりが起こる場合は指数を+1する．

丸めのときにけた上がりが起きた場合，あらためて先頭の‘1’に合わせて仮数を取り直す必要はありません．なぜなら，丸めの結果としてけた上がりが起きる場合は丸めの前の仮数はすべて‘1’で，Gビットも‘1’であり，丸めの後では仮数はすべて‘0’になっているからです．すべて‘0’の仮数を左に

見本

1ビットずらしても，やはり仮数はすべて‘ 0 ’なので結果は同じであり，この処理を省くことができます．つまり，正規化のためにバレル・シフタを動かすのは，丸めの前の1回だけでよいことになります．整数の有効ビット数が24ビット以下の場合，浮動小数点に変換しても精度は変わりません．このように，浮動小数点数に変換することを正規化と言います．

リスト7-1　プライオリティ・エンコーダの記述例

```
function Priority3( bits : std_logic_vector(2 downto 0))
  return std_logic_vector is
  variable Code : std_logic_vector(1 downto 0);
  begin
    Code(1) := bits(2) or bits(1);
    Code(0) := bits(2) or ( not bits(1) and bits(0));
  return Code;
end Priority3;


function Priority7( bits : std_logic_vector(6 downto 0))
  return std_logic_vector is
  variable Code : std_logic_vector(2 downto 0);
  begin
    Code(2) := bits(6) or bits(5) or bits(4) or bits(3);
    Code(1) := bits(6) or bits(5)
          or( not bits(4) and not bits(3) and( bits(2) or bits(1)));
    Code(0) := bits(6)
          or ( not bits(5) and( bits(4)
          or ( not bits(3) and( bits(2)
          or ( not bits(1) and( bits(0)))))));
  return Code;
end Priority7;

function Priority15( bits : std_logic_vector(14 downto 0))
  return std_logic_vector is
  variable Code : std_logic_vector(3 downto 0);
  begin
    Code(3) := bits(14) or bits(13) or bits(12) or bits(11)
          or bits(10) or bits(9) or bits(8) or bits(7);
    Code(2) := bits(14) or bits(13) or bits(12) or bits(11)
          or( not bits(10) and not bits(9) and not bits(8) and not bits(7)
           and( bits(6) or bits(5) or bits(4) or bits(3)));
    Code(1) := bits(14) or bits(13)
          or( not bits(12) and not bits(11) and( bits(10) or bits(9)
          or( not bits(8) and not bits(7) and( bits(6) or bits(5)
          or( not bits(4) and not bits(3) and( bits(2) or bits(1)))))));
    Code(0) := bits(14)
          or ( not bits(13) and( bits(12)
          or ( not bits(11) and( bits(10)
          or ( not bits(9) and( bits(8)
          or ( not bits(7) and( bits(6)
          or ( not bits(5) and( bits(4)
          or ( not bits(3) and( bits(2)
          or ( not bits(1) and  bits(0))))))))))))));
  return Code;
end Priority15;
```

3入力

7入力

15入力

見本

第1の部品 ―― プライオリティ・エンコーダ

正数のMSBの‘1’を見つける(FFO：find first one)ために，プライオリティ・エンコーダが使われます．プライオリティ・エンコーダはMSB側のほうが簡単な回路なので，信号の確定が早くなります．**リスト7-1**に，3，7，15入力プライオリティ・エンコーダのVHDLの記述例を示します．

複雑な回路を組むときは，まずその回路が小さな単位で組めるかどうかという検討をつねに行うべきです．小さな部品を組み合わせて大きな構造を作ったほうが効率良くなったとき，大きな構造物は意味を失います．しかし，これは小さな構造がつねに良いという意味ではありません．

プライオリテイ・エンコーダなら4ビットは専用の回路のほうがいいし，8ビットでも意味があります．しかし，16ビットになると，専用の回路を組むより4ビットのプライオリティ・エンコーダを五つ並べたほうが回路規模，動作速度ともに有利になります．小さな部品を明示的に書くことによって，コンパイルの時間が短くなるという現実的なメリットもあります．このように基本セルを二つ直列に並べたほうが，専用の回路より回路規模，動作速度の点から有利になるときは，それが基本ユニットになります．

一般に，回路規模の最適点と動作速度の最適点は異なります．例えば，加算器なら回路規模が最小となるのはフル・アダーを並べたリプル・アダーと呼ばれる構成ですが，その一方で動作速度はいちばん遅くなります．動作速度を上げていくと，回路規模が加速度的に増えていきます．そのため，双方にとっての最適点を探す必要があります．

実際には，まず回路規模が最小となる構成で機能を記述し，動作速度を指定して論理合成を行ってみます．このとき，動作速度が出ない箇所があれば，そこを速い回路に置き換えていく，あるいは複数の回路を並列に動かすなど，回路の構成を変えていきます．どうしても思いどおりに動かないときは，動作周波数を下げることになるかもしれません．

第2の部品 ―― バレル・シフタ

バレル・シフタには二つのシフトの方向(ライト，レフト)があります．これらは，浮動小数点演算では違った用途に使われます．

まず，レフト・バレル・シフタは正規化に使われます．このレフト・バレル・シフタはFFOと組み合わせて使われ，最上位の‘1’を左端にそろえる働きをします(**図7-7**)．

バレル・シフタの本体は複数のデータ・セレクタを直列に並べたものですが，データ・セレクタの並べかたには定石があります．レフト・バレル・シフタではシフト量の大きい側から，ライト・バレル・シフタではシフト量の小さい側から並べます．なぜなら，レフト・バレル・シフタのシフト量を示す信号(**図7-7**の`Shift`)はプライオリティ・エンコーダの出力なので，MSB側から決定される傾向があるからです．`Shift`が決定される順にバレル・シフタで使えるようにすれば，回路の遅延が少なくなります．

ライト・バレル・シフタは加減算で使われます．加減算を行う二つの数値の指数の差を求め，その差の分だけ小さいほうを右にシフトします(**図7-8**)．シフト量は引き算の結果なので，下位から順に確定します．よってシフト量の少ない側から使うのです．

レフト・バレル・シフタに入力される信号(**図7-7**の`Data`)の先頭の‘1’の位置は仮のものです．この位置は，正規化の結果で変わるかもしれません．例えば，加算と積算の後の正規化では，先頭の‘1’の位置は2ヵ所，すなわち，もとの‘1’の位置か，その左隣にあります．この2ヵ所に合わせて丸めの

見本

```
entity BarrelShifterLeft23 is
port(
  Data       : in std_logic_vector (27 downto 0);
  Shift      : in std_logic_vector (4 downto 0);
    q        : out std_logic_vector (22 downto 0)
  );
end BarrelShifterLeft23 ;


architecture RTL of BarrelShifterLeft23 is

signal Parameter1 : std_logic_vector(26 downto 0);
signal Parameter2 : std_logic_vector(25 downto 0);
signal AddAll : std_logic_vector(24 downto 0);
signal Round : std_logic_vector(2 downto 0);;
signal RoundOp : std_logoc

begin

Round(0) <= '1' when Data(27 downto 25) ="001"
             and Data(1) ='1' and (Data(2) ='1' or Data(0) ='1')
      else  '0';

Round(1) <= '1' when Data(27 downto 26) ="01" and Data(2) ='1'
             and (Data(3) ='1' or Data(1 downto 0) s/="00")
      else  '0';

Round(2) <= '1' when Data(27) ='1' and  Data(3) ='1'
             and (Data(4) ='1' or Data(2 downto 0) /="000")
      else  '0';

RoundOp  <= '1' when Data(27 downto 25) /= "000"  else '0';

AddAll <= Data(26 downto 2)
       + ( "0000000000000000000000" & Round );

Parameter1 <= Data(26 downto 0) when Shift(4 downto 3)="00"
      else "00000000" & Data(26 downto 8) when Shift(4 downto 3)="01"
      else "0000000000000000" & Data(26 downto 16) when Shift(4 downto 3)="10"
      else "000000000000000000000000" & Data(26 downto 24);

Parameter2 <= AddAll(26 downto 1) when RoundOp ='1'
      else Parameter1(26 downto 1) when Shift(2) ='0'
      else Parameter1(22 downto 0) & "000";

q    <=  Parameter2(25 downto 3) when Shift(1 downto 0) ="00"
      else Parameter2(24 downto 2) when Shift(1 downto 0) ="01"
      else Parameter2(23 downto 1) when Shift(1 downto 0) ="10"
      else Parameter2(22 downto 0);

end RTL;
```

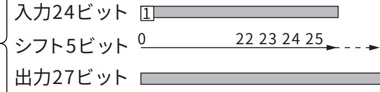
入力28ビット
シフト5ビット
出力24ビット 1

Data 27 0
0 8 16 24
0 4
0 1 2 3
q

図7-7　レフト・バレル・シフタ
レフト・バレル・シフタではシフト量の大きい側から並べる．

見本

```
entity BarrelShifterRight24 is
port(
  Data      : in STD_LOGIC_VECTOR (23 downto 0);
  Shift     : in STD_LOGIC_VECTOR (4 downto 0);
    q       : out STD_LOGIC_VECTOR (26 downto 0)
  );
end BarrelShifterRight24;


architecture RTL of BarrelShifterRight24 is


signal Parameter1 : std_logic_vector(26 downto 0);
signal Parameter2 : std_logic_vector(26 downto 0);
signal Parameter3 : std_logic_vector(26 downto 0);


begin

Parameter1 <= Data & "000" when Shift(1 downto 0) ="00"
         else '0' & Data & "00" when Shift(1 downto 0) ="01"
         else "00" & Data & '0' when Shift(1 downto 0) ="10"
         else "000" & Data ;

Round0 <= '1' when Parameter1(3 downto 0) /="0000"  else '0';

Parameter2 <= Parameter1 when Shift(2) ='0'
         else "0000" & Parameter1(26 downto 4) & Round0 ;

Round1 <='0' when Parameter2(8 downto 0) ="00000000"  else '1';
Round2 <='0' when Parameter2(16 downto 0)="0000000000000000"  else '1';
Round3 <='0' when Parameter2(24 downto 0)="000000000000000000000000" else '1';

q          <= Parameter2 when Shift(4 downto 3) ="00"
         else "00000000" Paqrameter2((26 downto 9)  & Round1
                   when Shift(4 downto 3) ="01"
         else "0000000000000000" Paqrameter2((26 downto 17) & Round2
                   when Shift(4 downto 3) ="10"
         else "000000000000000000000000" Paqrameter2((26 downto 17) & Round2
                   when Shift(4 downto 3) ="11" ;

end RTL;
```



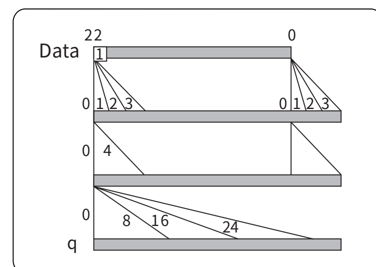


図7-8 ライト・バレル・シフタ

ライト・バレル・シフタではシフト量の小さい側から並べる．

見本

位置が変わります．減算の後では先頭の‘1’の位置は複雑な挙動を示します（詳細は後述）．

丸めを飛ばして高速演算

浮動小数点の演算では一つの演算がすむたびに正規化するのが規則です．正規化するための下準備としてMSBの位置がだいたい決まっていて，丸めが終わっていない数を，ここでは「未正規化数」と呼ぶことにします．この未正規化数を丸めて，MSBの位置を合わせれば正規化数になります．

正規化を行うには，一つの加算器と1ビットのバレル・シフタを含んだ簡単なしくみでよいのですが，そのわりに処理時間はしっかり1クロックを要するため，ある意味やっかい者です．未正規化数のまま次の演算を開始できればシステム性能を上げることができます．正規化する前の数を27ビットの未正規化数として受け入れるような加算器，積算器を作ることもできます．

このように正規化を省略して加算や積算を続けていくと，MSBの位置は一つ演算が進むごとに‘1’だけ不確定になります．3回程度なら連続演算にも耐えられます．

7-3　浮動小数点数の積算を行う

では部品がそろったところで，具体的な演算としてまず積算を行ってみましょう．

指数の計算ではオフセットを引くことを忘れずに

積算は，まず指数の加算を行ってから127を減算します．これは，例えばもとの指数をa，b，およびaとbの和をcとしたとき，

$$(a+127)+(b+127)=(c+254)-127 \quad \cdots\cdots (7\text{-}4)$$

を求めることになります．最後にオフセットを引かなければならないことに注意してください．計算した結果は，次のようになります（ただし，ここではグラデュアル・アンダフローは採用していない）．

- －1以下：0
- 0：仮数の計算結果で指数が＋1されれば正規化数，そうでない場合は0
- 1～253：計算結果は正規化数
- 254：仮数の計算結果で指数が＋1されれば∞，そうでない場合は正規化数
- 255以上：∞

仮数48ビットのうち下位22ビットは丸め

仮数は先頭に‘1’を付けて積算します．積算した結果は48ビットになります．この48ビットのうち最初の26ビットはそのまま取り出して，残りの下位の22ビットはすべてORします．ORした結果はRビットとして右端に付けます（**図7-9**）．

さて，ここで積算結果の27ビットの左端の2ビットについては，“11…”，“10…”，“01…”の3通りになります．これはもともと1≦（仮数）≦（2－ulp）の変域なので，計算した結果は1≦（仮数）≦（4－4ulp）になるからです．ここでMSBが‘1’の場合と，MSBのすぐ下のビットから26ビット目のビットがすべて‘1’の場合は，指数を＋1します（**図7-10**）．後者は丸めの結果として指数を＋1する場合であり，このときのulpはかならず‘1’なので，ラウンド・イーブンの規則が優先されるため，この指数計

見本