はじめに

- 今回の講義テーマは浮動小数点数の表現形式と、浮動小数点数演算器の構成です。
- まず、浮動小数点数の表現形式について説明します。

- 教科書の対応範囲は、以下の通りです。
  - 3.2.2項（p.87～92）

固定小数点数だけではなぜダメなのか？

- この科目の最初の方の講義で、固定小数点数について学びました。
  - 通常は整数を表現しているものとして扱いますが、必ずしも整数だけではなく、実数も表現できます。
  - しかし、その表現範囲は、表現するビット数で限られています。
    - » もちろん、これは当たり前のことですが。。。
    - 例えば、64ビットの符号なし整数で表現できる最大値は $2^{64}-1 \fallingdotseq 2^{64} = 2^4 \times (2^{10})^6 \fallingdotseq 16 \times (10^3)^6 = 16 \times 10^{18}$ 程度です。
      - これではアボガドロ定数（6.022×10^{23}）も表現できませんから、化学の分野ではコンピュータはひどく使いづらいものになるでしょう。
      - ところで、その一方で、アボガドロ定数の1の位の値は何か、などは全く気にしませんね。

- 科学技術分野では、$10^{\square}$や$10^{-\square}$がつく、非常に大きな数や非常に小さな数を扱いますが、幸いなことに、有効数字という考え方があります。
- そこで、有効数字の考え方を利用しながら、限られたビット数で非常に大きな数や非常に小さな数を表現するために考え出された（設けられた）のが、浮動小数点数です。
  - とは言っても、大げさに言うほどの難しいことはしません。
  - 普通に使用している表現方法に合わせて、基数（radix）を10ではなく2（のべき）にするぐらいで、$m \times 2^e$ の形で表現します。
    - ここで、*m*を仮数（mantissa）部、*e*を指数（exponent）部、と言います。
    - なお、昔は、基数が16（$=2^4$）の表現が使われていたこともありますが、現在は、基数は2で統一されています。

- ところが、この $m \times 2^e$ の形の表現には、やや困ったことが発生します。
  - 例えば、
    - $(3.14)_{10}$ $= 314 \times 10^{-2}$ $= 31.4 \times 10^{-1}$
      $= 3.14 \times 10^{0}$ $= 0.314 \times 10^{1}$ $= \ldots$
    - $(11.001)_{2}$ $= 11001 \times 2^{-3}$ $= 1100.1 \times 2^{-2}$
      $= 110.01 \times 2^{-1}$ $= 11.001 \times 2^{0}$
      $= 1.1001 \times 2^{1}$ $= 0.11001 \times 2^{2}$ $= \ldots$
  - つまり、表現が一意に決まりません。
  - そこで、例えば、以下のような形に統一することが考えられます。

$$0.xy_2y_3y_4\ldots \times r^{e} \quad (x \neq 0)$$

    - つまり、仮数の整数部が0、小数第一位が0以外、となるように指数部（e）を調整します。

浮動小数点数のデータ形式

- フィールド構成
  - 結局、以下の3つをセットにして、1つの数を表現することにします。
    - 符号ビット
    - 仮数部
    - 指数部
  - 基数はあらかじめ決めたもの（2や、過去には16など）を使用しますので、わざわざデータ中に含めません。
  - 仮数部と指数部を用いて、数の絶対値を表現します。
    - 固定小数点数の場合のように、負の数を2の補数で表現する、というようなことは行いません。

- 正規化（normalization）
  - 先に、表現形式を以下の形に統一すると述べました。

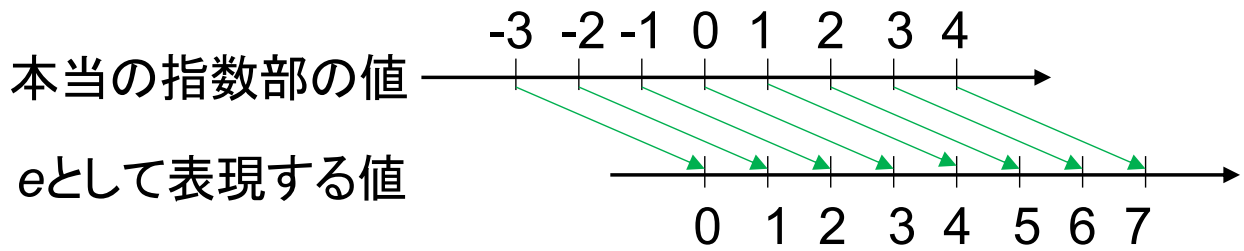
$$0.xy_2y_3y_4\ldots \times r^{e} \quad (x \neq 0)$$

  - しかし、基数が2の場合は $x=1$ しかありません。
    - ならば、必ず1だと決まっている部分（ビット）をわざわざ表現に含めるのは無駄です。それなら、たった1ビットですが、有効桁を増やしたい。
  - ということで、仮数部は以下のように表現することに改めます。

$$1.y_1y_2y_3y_4\ldots$$

    - 整数部の1は省略して、小数部（fraction）のみをデータ表現に含めます。（これをけち表現と呼んでいます。）
      - なお、整数部を必ず1にすると決めたことによって、0が表現できなくなったことに注意してください。
      - このため、英語では mantissa ではなく fraction と言いますが、日本語では「仮数部」のまま呼称しています。
    - 演算結果は、必ずしもこの形式になりません。そのような数をこの形式にすることを「正規化する（normalize）」と言います。

- 指数部のバイアス表現
  - 指数部eは2の補数で表現してもよかったのですが、現在使用されている規格（後述）では、“下駄（げた）”（バイアス; bias）を履かせて表現することにしています。
  - つまり、座標軸上を平行移動することにより、非負数のみでeを表現します。
    - eがnビットのとき、げたを$2^{n-1}-1$とします。
      - eを3ビットとすると、げたは$2^{3-1}-1=3$（2進数で011）。



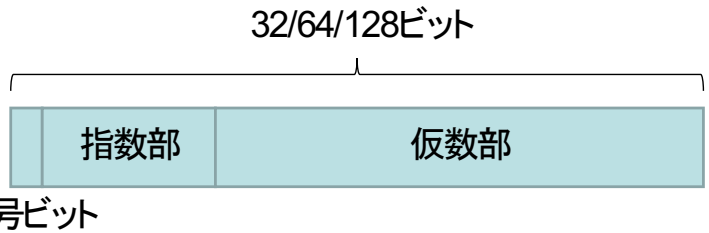
ANSI/IEEE標準754-1985

- 規格団体であるANSIと電気関係の学会IEEEが、浮動小数点演算について1985年に策定した標準規格です。
  - 現在は、その内容を含む形で改訂されたANSI/IEEE標準754-2008が最新の規格です。
    - » 長いので、普段の会話では「IEEE（の）標準」だけで済まします。
  - 現在のほとんどのコンピュータは、この規格に従っています。
    - それでも、なお、実際には、異なるコンピュータ間では、同じ入力に対して必ずしも同じ演算結果とはならない場合があり、注意が必要です。
      - 演算器の内部で、何ビットまで多く処理しているかによって変わってきます。

「アイトリプル イー」と読みます。

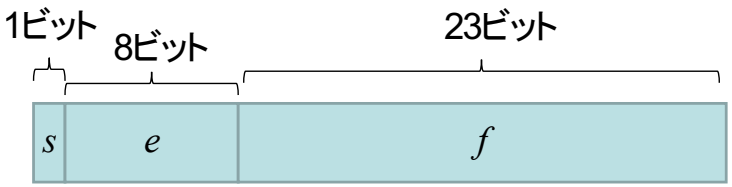
- 精度
  - 単精度（single precision）
    - 32ビット（符号：1ビット、指数部：8ビット、仮数部：23ビット）で表現します。
    - C言語のfloat型は単精度の浮動小数点数です。
  - 倍精度（double precision）
    - 64ビット（符号：1ビット、指数部：11ビット、仮数部：52ビット）で表現します。
    - C言語のdouble型は倍精度の浮動小数点数です。
  - 4倍精度（quad precision）
    - 128ビット（符号：1ビット、指数部：15ビット、仮数部：112ビット）で表現します。
    - ハードウェアで128ビットの浮動小数点数演算器を備えているコンピュータはあまりありません。
    - C言語のlong double型は4倍精度の浮動小数点数です。

フォーマット：



- 例） 単精度の場合の値の定義
  - 0＜ e ＜255 のとき
    - 正規化数です。 $(-1)^s \times 2^{e-127} \times 1.f$ を表します。
  - e=0、f ≠0のとき
    - 整数部が1にならない非正規化数として、 $(-1)^s \times 2^{e-126} \times 0.f$ を表現します。
  - e=0、f =0のとき
    - 0（+0、–0）を表します。
  - s=0、e=255、f =0のとき
    - +∞（プラスの無限大）を表します。
  - s=1、e=255、f =0のとき
    - –∞（マイナスの無限大）を表します。
  - e=255で上記以外のとき
    - 数値を表現しません。非数として、これをNaN（Not a Number）と呼びます。

不自然な表記ですが、ここでは、整数部が0で、小数部がfの数を表しているものと解釈してください。上の$1.f$についても同様です。



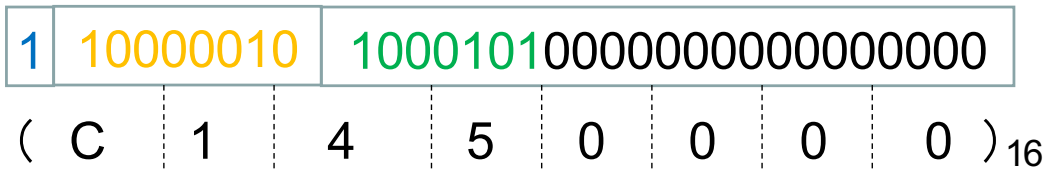
– 補足

- 0は正規化数として扱えない（整数部が1でない）ので、特殊な数としての扱いです。
- 例えば、正の正規化数を0で割ると、結果は+∞になります。
  - 割り込みを起こすようにすることもできますが、浮動小数点数で0で割った場合は、大抵のシステムでは、デフォルト処理として∞にすることにしているようです。
- 例えば、+∞を+∞で割ると、結果はNaNになります。
- 実は、プログラム上では、右のようなプログラムはちゃんと機能します。変数aが浮動小数点数で±∞やNaNのとき、最後のelse節を実行することになります。

例えば、C言語で∞やNaNのfloat型変数をprintf()で出力すると、infやnanと出力されます。これは、printf()が変数の値をチェックして、infなどの文字列に置き換えているのです。

```
if( a > 0.0 ) {
    ……
} else
if( a < 0.0 ) {
    ……
} else
if( a == 0.0 ) {
    ……
} else {
    ……
}
```

- 例） 10進数の–12.3125が単精度の浮動小数点数としてどのように表現されるかを見てみます。
  - $(12.3125)_{10}=1100.0101=1.1000101\times 2^{3}$ これで仮数部は求まりました。
  - 指数部は127の下駄を履かせて、3+127=130を2進数で表します。$(130)_{10}=10000010$。
  - 負の数ですから、符号ビットは1です。
  - よって以下の表現になります。



コンピュータのメモリには、このように（16進表記でC1450000）記憶されます。これを32ビットの浮動小数点数と解釈した場合には、−12.3125を表しています。しかし、これが32ビットの符号なし整数と解釈すると、その値は3242524672であり、また、32ビットの符号付き整数と解釈すれば–1052442624を表します。メモリ上のデータを見ても、それが何を表しているのかは一意に決まりません。表現と、それが表している値との関係を、しっかりと理解しておいてください。

- では、続けて、浮動小数点数演算器の構成について説明します。

- 教科書の対応範囲は、以下の通りです。
  - 6.2節（p.202～209）

浮動小数点数の四則演算

- 浮動小数点数演算器は、浮動小数点数を入力し、演算結果の浮動小数点数を出力します。
  - しかし、浮動小数点数は、符号ビット、指数部、仮数部の3つをセットにして32ビットまたは64に納めているだけですので、32/64ビットそのままデータを、直接、加算したり乗算したりすることはできません。
  - 浮動小数点数演算器の内部では、浮動小数点数を符号ビット、指数部、仮数部のそれぞれに分割して処理します。
  - 指数部、仮数部のそれぞれの計算は、固定小数点数の演算として処理します。したがって、そこでは、負の数は2の補数で表現することを前提に、演算回路を設計します。

- 以下では、浮動小数点数xの指数部をe_x、仮数部をm_xと表すことにします。
  - m_xには正規化数の場合の整数部1も含むものとします。したがって、$x = m_x \times 2^{e_x - b}$ です。ここで、bは下駄（バイアス）です。
  - 煩雑になるので、ここでは、正の数の場合のみを説明します。実際には負の数の場合も考えなければなりませんが、皆さんなら容易に想像できるはずです。
  - 高校の数学のレベルの話ですが、仮数部と指数部のそれぞれをどのように処理しなければならないかに注意してください。

- 加減算
  - まず、小数点の桁合わせを行わなければなりません。
  - 浮動小数点数x、yに対して、その指数部が$e_x \geq e_y$とすると、2^{e_x-b}で桁合わせを行います。
  - $x \pm y = m_x \times 2^{e_x-b} \pm m_y \times 2^{e_y-b}$

    $= (m_x \pm m_y \times 2^{e_y-e_x}) \times 2^{e_x-b}$
  - $m_y \times 2^{e_y-e_x}$は、m_yを$e_x - e_y$ビットだけ右にシフトすることで求まります。
  - 上式の括弧内の加減算は、固定小数点数として計算します。

- 乗算
  - 桁合わせの必要がありませんので楽ですが、指数部のバイアスの扱いに注意してください。
  - $x \times y = m_x \times 2^{e_x - b} \times m_y \times 2^{e_y - b}$
    $= (m_x \times m_y) \times 2^{(e_x + e_y - b) - b}$

- 除算
  - $x \div y = (m_x \times 2^{e_x - b}) \div (m_y \times 2^{e_y - b})$
    $= (m_x \div m_y) \times 2^{(e_x - e_y + b) - b}$

浮動小数点数演算器

- 現在は、以下の種類の演算器を独立に設けるのが普通です。
  - 浮動小数点加減算器
    - 浮動小数点加算命令
    - 浮動小数点減算命令
    - 浮動小数点数の精度変換（単精度←→倍精度）命令
    - 浮動小数点数と固定小数点数との間の変換命令

    などを実行します。
  - 浮動小数点乗算器
    - 浮動小数点加算命令を実行します。
  - 浮動小数点除算器
    - 浮動小数点除算命令を実行します。

- 演算器の構成
  - 演算器内でのおおよその処理の流れは以下の通りです。

1. まず、演算を行います。
   - 演算器の種類によって、その演算内容に合わせて構成は異なります。
2. 次に、演算結果を正規化します。
3. 最後に、演算結果を丸めます。
   - 演算の結果、仮数部が長くなってしまう場合があります。それを、定められた仮数部の桁数（23/52ビット）に収まるように処理します。

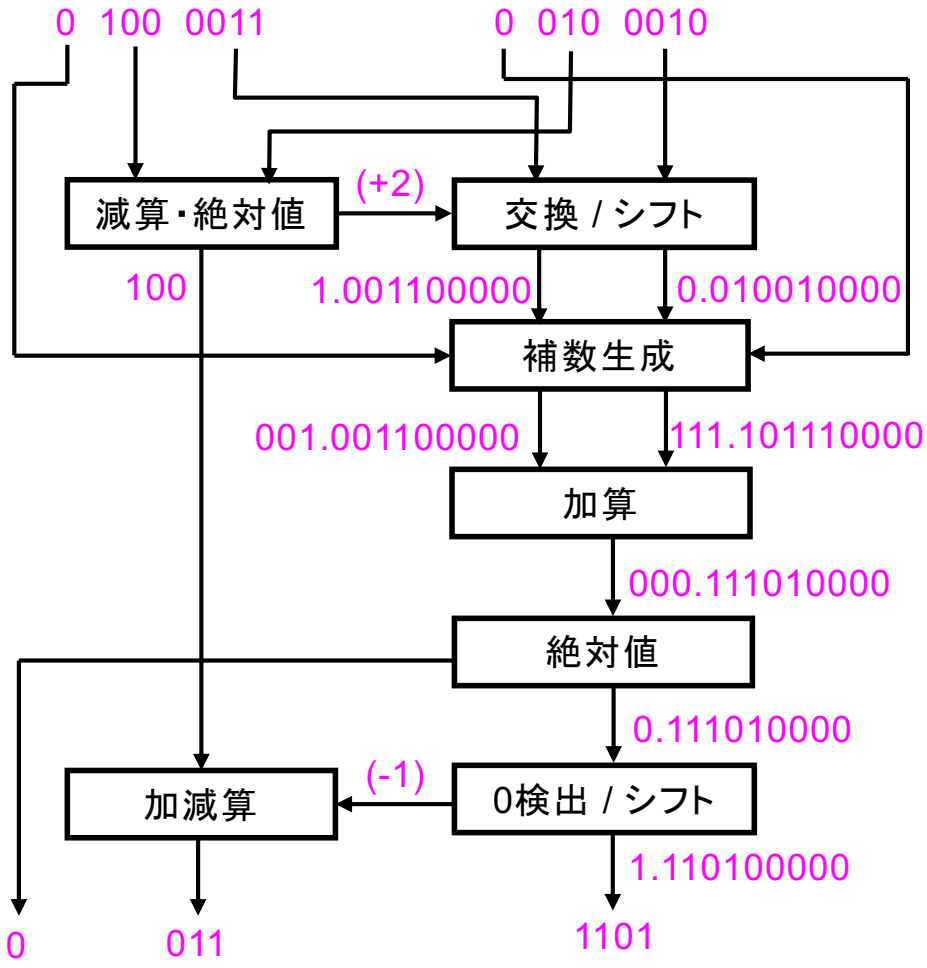
- 以降では、浮動小数点加減算器と浮動小数点乗算器の大まかなブロック図を示します。

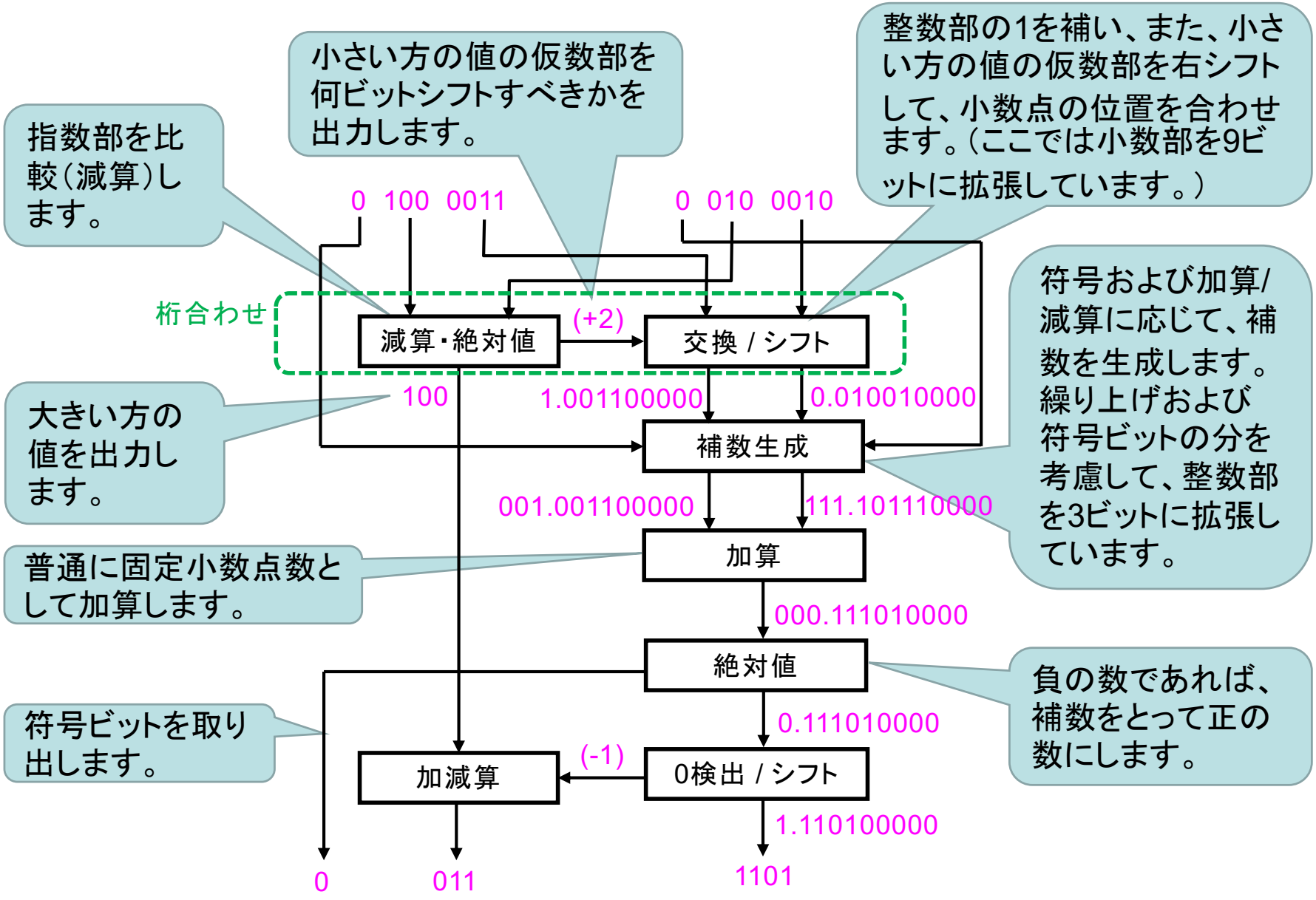
  - なお、演算処理の例を示しますが、簡単のため、8ビットの浮動小数点数を定義（考え方はIEEE標準と同じ）して用いることにします。
    - 符号ビットは1ビットです。
    - 指数部は3ビットです。下駄を011（=3）とします。
    - 仮数部は4ビットです。

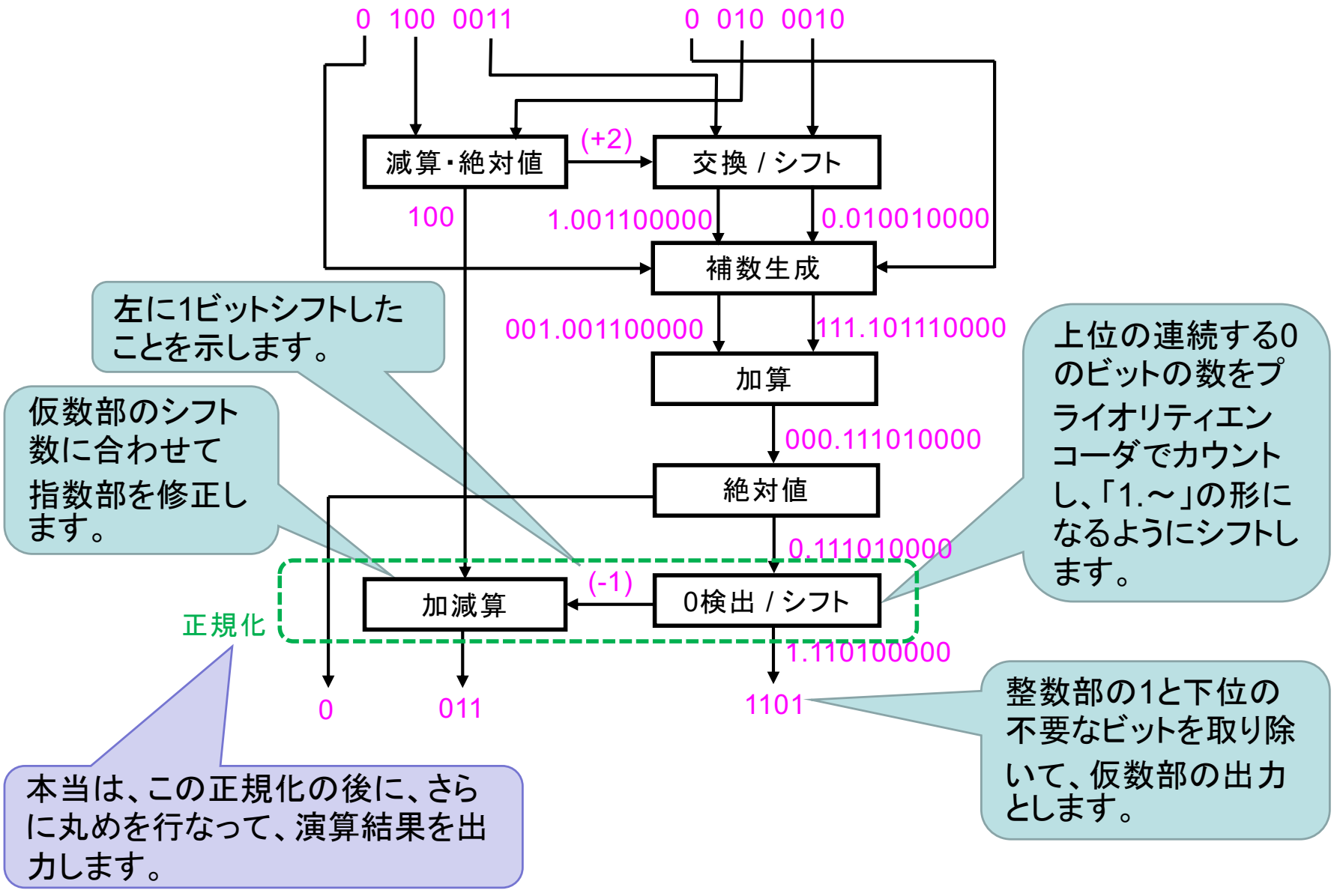
    例）$(0\ 010\ 0100)_2$ は 0.625 を表す。

    $+(1.0100)_2 \times 2^{2-3} = 1.25 \times 2^{-1} = 0.625$

- 浮動小数点加減算器の構成
  - ピンク色で示しているのは、2.375−0.5625を計算する場合の各部のデータ値です。
  - 参考
    - 2.375の表現：0 100 0011
    - 0.5625の表現：0 010 0010
    - 1.8125の表現：0 011 1101




指数部を比較(減算)します。
小さい方の値の仮数部を何ビットシフトすべきかを出力します。
整数部の1を補い、また、小さい方の値の仮数部を右シフトして、小数点の位置を合わせます。(ここでは小数部を9ビットに拡張しています。)
0 100 0011
0 010 0010
桁合わせ
減算・絶対値
(+2)
交換 / シフト
大きい方の値を出力します。
100
1.001100000
0.010010000
補数生成
符号および加算/減算に応じて、補数を生成します。繰り上げおよび符号ビットの分を考慮して、整数部を3ビットに拡張しています。
001.001100000
111.101110000
普通に固定小数点数として加算します。
加算
000.111010000
絶対値
負の数であれば、補数をとって正の数にします。
符号ビットを取り出します。
0.111010000
(-1)
加減算
0検出 / シフト
1.110100000
0
011
1101


0 100 0011
0 010 0010
減算・絶対値
(+2)
交換 / シフト
100
1.001100000
0.010010000
補数生成
001.001100000
111.101110000
加算
000.111010000
絶対値
0.111010000
(-1)
加減算
0検出 / シフト
正規化
1.110100000
0
011
1101
左に1ビットシフトしたことを示します。
仮数部のシフト数に合わせて指数部を修正します。
上位の連続する0のビットの数をプライオリティエンコーダでカウントし、「1.～」の形になるようにシフトします。
本当は、この正規化の後に、さらに丸めを行なって、演算結果を出力します。
整数部の1と下位の不要なビットを取り除いて、仮数部の出力とします。

- 丸め（端数処理、Rounding）
  - 循環小数n場合や、演算した結果、仮数部が長くなってしまった場合でも、定められた仮数部の桁数に収めなければなりません。
  - 我々は、日常的に、四捨五入や切り捨て・切り上げなどで数値を丸めますが、IEEE標準で定義された丸めはちょっと厄介です。
  - IEEE標準では、以下の5種の丸めが定義されています。
    - round to 0
    - round to +∞
    - round to −∞
    - round to nearest (even if tie)
    - round to nearest (away if tie)

- round to 0
  - 0に近い方に丸めます。つまり、絶対値を切り捨てます。
  - 例
    - $+1.1011011 \times 2^e \rightarrow +1.1011 \times 2^e$
    - $-1.1011011 \times 2^e \rightarrow -1.1011 \times 2^e$
- round to $+\infty$
  - 値が大きくなるように丸めます。
  - 例
    - $+1.1011011 \times 2^e \rightarrow +1.1100 \times 2^e$
    - $-1.1011011 \times 2^e \rightarrow -1.1011 \times 2^e$
- round to $-\infty$
  - 値が小さくなるように丸めます。
  - 例
    - $+1.1011011 \times 2^e \rightarrow +1.1011 \times 2^e$
    - $-1.1011011 \times 2^e \rightarrow -1.1100 \times 2^e$

– round to nearest (even if tie)
  - 通常、デフォルトで使用される丸めです。
  - 四捨五入のように、最も近い値に丸めますが、ちょうど半分のときの扱いが四捨五入とは異なります。ちょうど半分のときは偶数になる（even if tie； つまり、最下位桁が0になる）ように丸めます。
  - 例
    – $+1.1011011 \times 2^e \rightarrow +1.1011 \times 2^e$
    – $+1.1011101 \times 2^e \rightarrow +1.1100 \times 2^e$
    – $+1.1011100 \times 2^e \rightarrow +1.1100 \times 2^e$
      » 切り捨てると最下位桁が1になるので、切り上げます。
    – $+1.1010100 \times 2^e \rightarrow +1.1010 \times 2^e$
      » 切り捨てると最下位桁が0になるので、切り捨てます。
    – 負の数の場合は、上記の「+」を「−」に置き換えて考えてください。

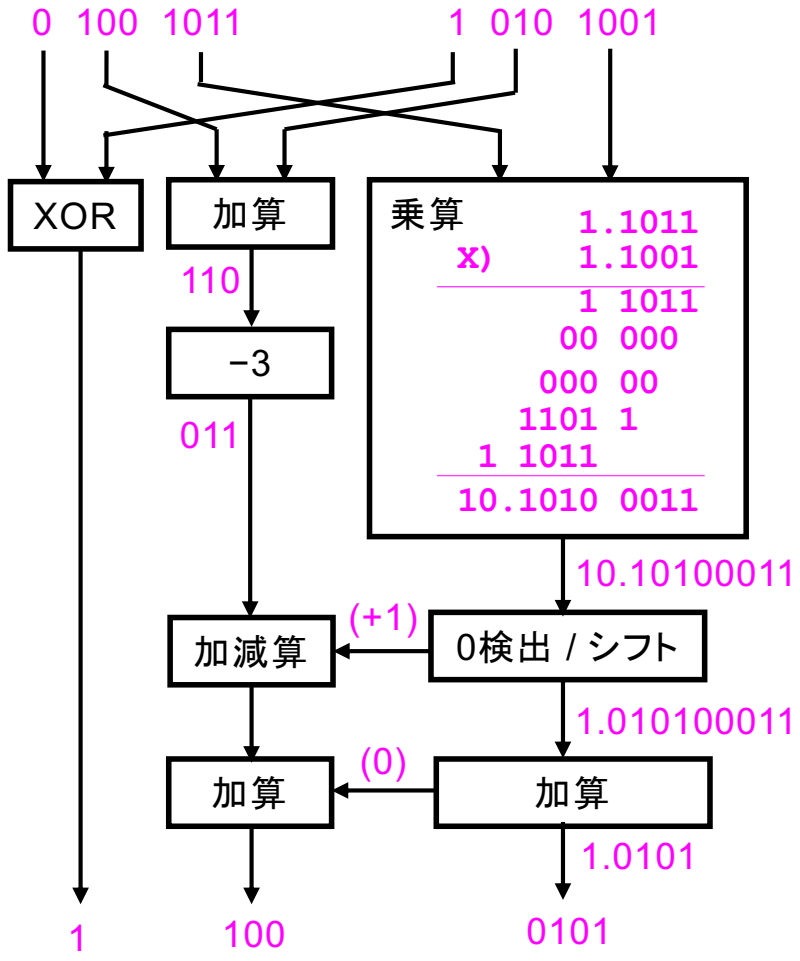
- しかし、この「even if tie」のルールが厄介な問題を引き起こします。
  - 例えば、先ほどの浮動小数点加減算器の内部では、小数部を9ビットに拡張していました。したがって、
    $1.1010 \times 2^{6-3} + 1.0001 \times 2^{1-3}$
    $= (1.1010\,00000 + 0.0000\,10001) \times 2^{6-3}$
    $= 1.1010\,10001 \times 2^{6-3}$
    $\rightarrow 1.1011 \times 2^{6-3}$
  - しかし、ハードウェアコストを削減するために、小数部を7ビットに拡張していたとすると、
    $1.1010 \times 2^{6-3} + 1.0001 \times 2^{1-3}$
    $= (1.1010\,000 + 0.0000\,100) \times 2^{6-3}$
    $= 1.1010\,100 \times 2^{6-3}$
    $\rightarrow 1.1010 \times 2^{6-3}$

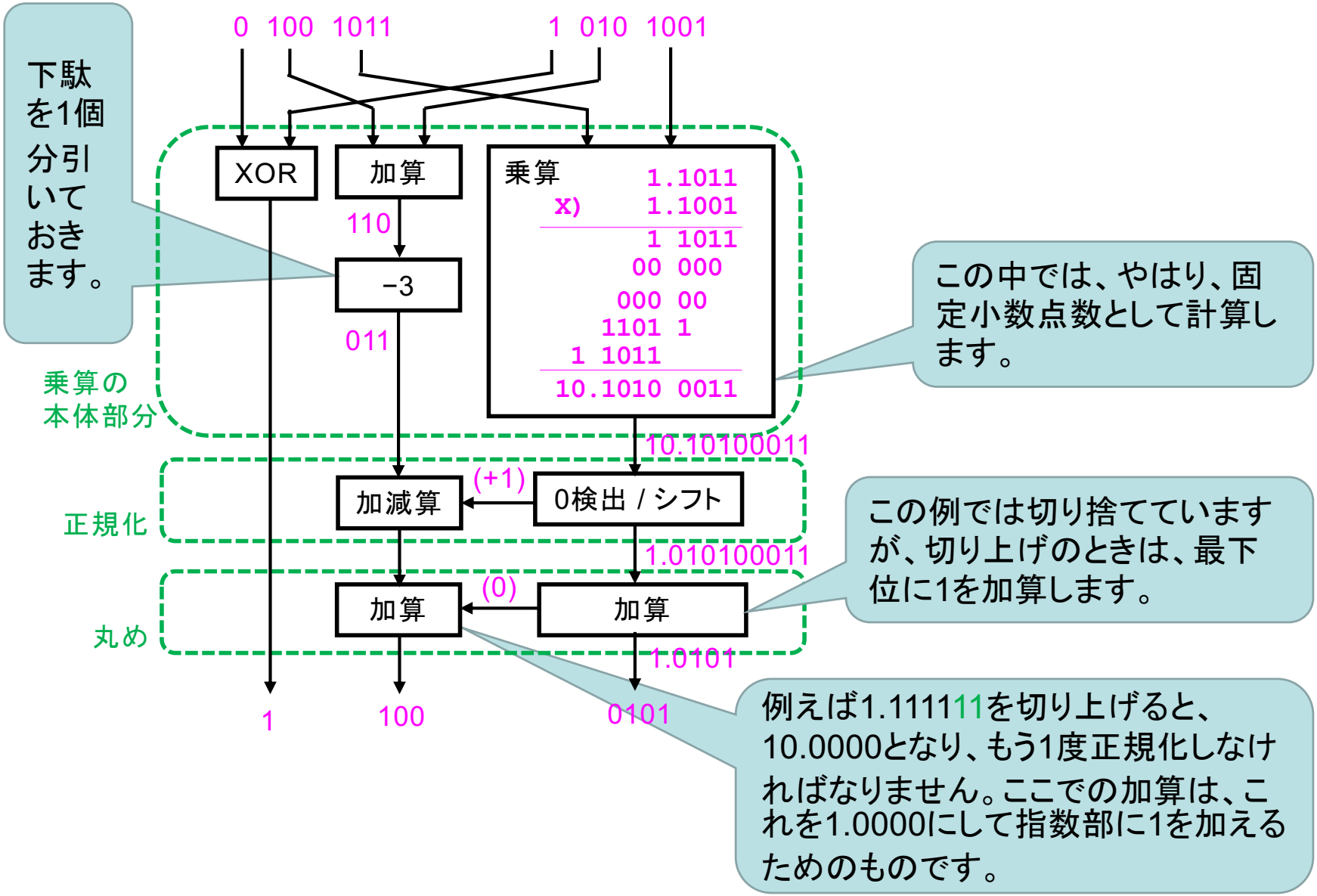
    これより下の位の値が失われてしまいます。
  - つまり、演算器の内部構成によって、結果が異なってしまうのです。このため、コンピュータによって演算結果が異なる、ということが実際に起こっています。

– round to nearest (away if tie)
  - ちょうど半分のときは、その絶対値が大きくなる方向に切り上げます。
  - 四捨五入と同じですね。
  - 「even if tie」の問題を避けるために、ANSI/IEEE標準754-2008で追加されました（ANSI/IEEE標準754-1985では定義されていませんでした）が、現在もあまり使われません。
    – 過去に作成して今でも使用しているソフトの計算結果が変わってしまうかもしれませんから、簡単には乗り換えられないのです。

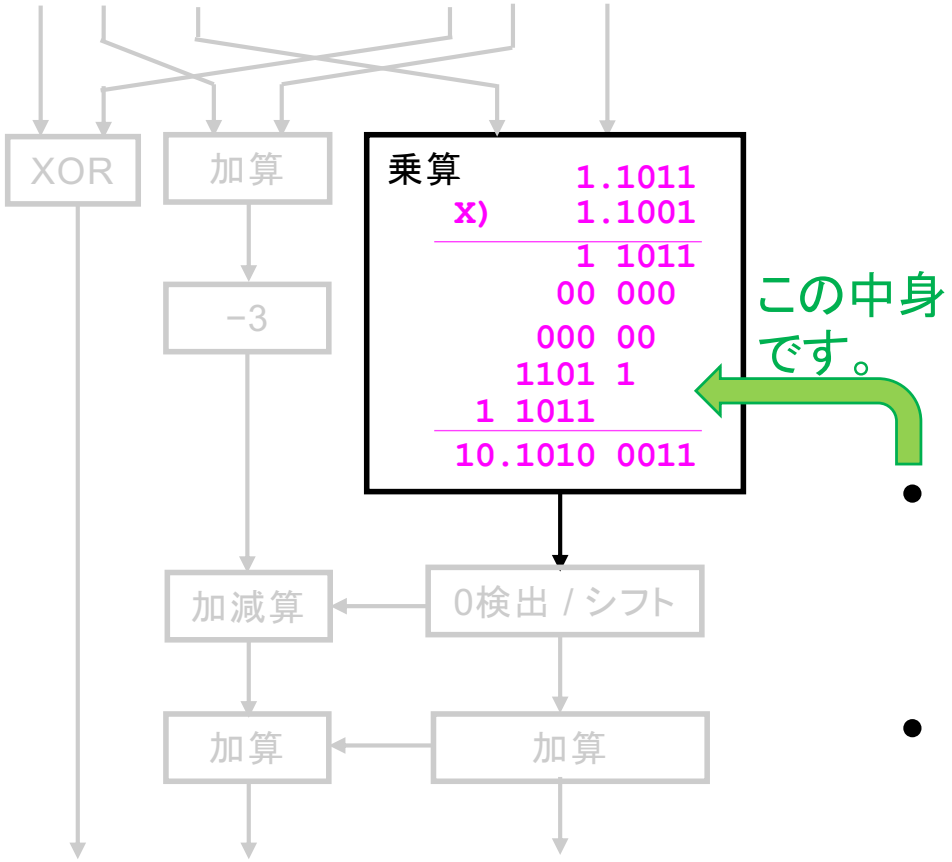
コンピュータの設計の怖さを示す事例の1つです。最初に賢くない選択をしてしまうと、その後もそれを永く引きずることになります。ですから、逆に、コンピュータを設計するときには、永い将来に渡って有効に機能するように、よく考えておかなければならないのです。

- 浮動小数点乗算器の構成
  - ピンク色で示しているのは、3.375×(−0.78125)を計算する場合の各部のデータ値です。
  - 参考
    - 3.375の表現:
      0 100 1011
    - −0.78125の表現:
      1 010 1001
    - − 2.625の表現:
      1 100 0101




下駄を1個分引いておきます。
0 100 1011
1 010 1001
XOR
加算
乗算
1.1011
x) 1.1001
1 1011
00 000
000 00
1101 1
1 1011
10.1010 0011
110
−3
011
乗算の本体部分
10.10100011
(+1)
加減算
0検出 / シフト
正規化
1.010100011
(0)
加算
加算
丸め
1.0101
1
100
0101
この中では、やはり、固定小数点数として計算します。
この例では切り捨てていますが、切り上げのときは、最下位に1を加算します。
例えば1.111111を切り上げると、10.0000となり、もう1度正規化しなければなりません。ここでの加算は、これを1.0000にして指数部に1を加えるためのものです。

おわりに



- 浮動小数点数の表現、浮動小数点数演算器の中身について理解できましたか？
  - 結局、計算の本質的な部分は固定小数点数の計算と同じです。
- そこで、次回は、乗算のアルゴリズムについて講義します。
- 因みに、次々回は、除算のアルゴリズムについて講義します。