2.6　シフト演算

2.3 節の

$$1110.011(2) \times 10(2),\ 16.3(8) \times 100(8),\ E.6(16) \times 1000(16) \quad (例題\ 5),$$
$$1110.011(2) \div 1000(2),\ 16.3(8) \div 100(8),\ E.6(16) \div 10(16) \quad (例題\ 6)$$

のように、基数のベキ乗数で掛けたり割ったりする演算 (数値を固定し、小数点の位置を動かす操作) を、情報科学の分野では**シフト演算** (shift instruction) と呼んでいます。シフト演算には、**小数点を固定し**、数値を左に動かす操作を施す**左シフト演算**と数値を右に動かす操作を施す**右シフト演算**があります。これらの操作は、数値を固定し、小数点を移動させる操作と相対的に同じですが、左右が逆になることに注意してください。なお、実際のシフト演算では、シフト演算する桁数 k (2 進数の場合は k ビット) を付加し、**k 回左シフト演算** (**k ビット左シフト演算**) や **k 回右シフト演算** (**k ビット右シフト演算**) という言い方を用います[12]。

具体例としては、1234.5678 に対して、下記のように 3 回**左**シフト演算を行うことで演算結果 1234567.8 を得ます。この演算は、数値を固定し、小数点の位置を**右**に 3 桁移動させる操作と同じで、1234.5678×10^3 を計算しています。

1 2 3 4 . 5 6 7 8

1 2 3 4 5 6 7 . 8　← ← ←　3 回左シフト演算

同様に、1234.5678 に対して、下記のように 6 回**右**シフト演算を行うことで演算結果 0.0012345678 を得ます (シフト演算によって空欄になった桁には、適切に 0 を補います)。この演算は、数値を固定し、小数点の位置を**左**に 6 桁移動させる操作と同じで、$1234.5678 \div 10^6$ を計算しています。

1 2 3 4 . 5 6 7 8

6 回右シフト演算　→ → → 0 . 0 0 1 2 3 4 5 6 7 8

例題 1　123(10) を 2 回右シフト演算しなさい。

解答例　1.23(10) (= 123(10) ÷ 100(10))

例題 2　123(8) を 4 回左シフト演算しなさい。

解答例　1230000(8) (= 123(8) × 10000(8))

例題 3　123(16) を 6 回右シフト演算しなさい。

解答例　0.000123(16) (= 123(16) ÷ 1000000(16))

[12] シフト演算する回数には、0 や負の数 $-k$ ($k > 0$) を指定することができます。0 回左 (右) シフト演算は何も操作を施さないのと同じで、$-k$ 回左 (右) シフト演算は k 回右 (左) シフト演算と同じです。

コンピュータは、掛け算や割り算の代わりに、足し算 (引き算には補数を用いる) とシフト演算を巧みに使って積や商を計算します。

掛け算 12×233 を足し算とシフト演算のみを使って計算してみましょう。12×233 は、

$$12 \times 233 = \underbrace{12 + 12 + \cdots + 12}_{233\text{ 個}} = 2796$$

のように 12 をひたすら 233 回足すことでことで求められますが、効率の良い計算方法とは言えません。そこで、

$$\begin{aligned}
&12 \times 233 \\
&\quad = 12 \times 2 \times 10^2 + 12 \times 3 \times 10^1 + 12 \times 3 \times 10^0 \\
&\quad = (12 + 12) \times 10^2 + (12 + 12 + 12) \times 10^1 + (12 + 12 + 12) \times 10^0 \\
&\quad = ((12 + 12)\underline{\times 10} + (12 + 12 + 12))\ \underline{\times 10} + (12 + 12 + 12)
\end{aligned}$$

のように変形し、上式の下線部をシフト演算に置き換えて計算します。すなわち、次の①から⑤の手順によって積 2796 を効率的に求めることが出来ます。

[準備] 233 の桁数を求める (3 桁より 1 回左シフト演算を $(3 - 1 =)$ **2 回**行う)

① 0 に 12 を 2 回足す $(0 + 12 + 12 = 24)$

② 24 に対して 1 回左シフト演算 (**1 回目**) を行う $(24 \times 10 = 240)$

③ 240 に 12 を 3 回足す $(240 + 12 + 12 + 12 = 276)$

④ 276 に対して 1 回左シフト演算 (**2 回目**) を行う $(276 \times 10 = 2760)$

⑤ 2760 に 12 を 3 回足す $(2760 + 12 + 12 + 12 = 2796)$

上記で紹介した掛け算の方法は、下記のような私たちがふだん何気なく行っている掛け算の方法とあまり変わりがないことがわかります。

```
      1 2
×   2 3 3
-----------
      3 6
    3 6 ←    ・・・      ④
  2 4 ← ←    ・・・   ②, ④
-----------
  2 7 9 6
```

2 進数の場合も同様に考えると、例えば、$1011(2) \times 1010(2)$ を

$$
\begin{aligned}
&1011(2) \times 1010(2) \\
&\quad = 1011(2) \times 1 \times 2^3 + 1011(2) \times 0 \times 2^2 + 1011(2) \times 1 \times 2^1 + 1011(2) \times 0 \times 2^0 \\
&\quad = \big(1011(2) \times 1 \times 2^2 + 1011(2) \times 0 \times 2^1 + 1011(2)\big) \underline{\times 2} + 0(2) \\
&\quad = \big((1011(2) \underline{\times 2} + 0(2)\big) \underline{\times 2} + 1011(2)\big) \underline{\times 2} + 0(2)
\end{aligned}
$$

と変形することにより、以下の手順で計算すると積 $1101110(2)$ が求めることが出来ます[13]。

[準備] 1010 の桁数を求める (4 桁より 1 ビット左シフト演算を $(4-1=)$ **3 回**行う)

① $0(2)$ に $1011(2)$ を 1 回足す ($0(2) + 1011(2) = 1011(2)$)

② $1011(2)$ に対して 1 ビット左シフト演算 (**1 回目**) を行う ($1011(2) \times 2 = 10110(2)$)

③ $10110(2)$ に $1011(2)$ を 0 回足す ($10110(2) + 0(2) = 10110(2)$)

④ $10110(2)$ に対して 1 ビット左シフト演算 (**2 回目**) を行う ($101100 \times 2 = 101100$)

⑤ $101100(2)$ に $1011(2)$ を 1 回足す ($101100(2) + 1011(2) = 110111(2)$)

⑥ $110111(2)$ に対して 1 ビット左シフト演算 (**3 回目**) を行う ($110111 \times 2 = 1101110$)

⑦ $1101110(2)$ に $1011(2)$ を 0 回足す ($1101110(2) + 0(2) = 1101110(2)$)

割り算　$72207 \div 355$ を足し算とシフト演算のみを使って計算してみましょう。$72207 \div 355$ は、言い換えると、下記のように 72207 から -355 (実際は補数表現する) を何回足せるかという問題に帰着されます。

$$
(0 \leq) \quad 72207 + \underbrace{(-355) + (-355) + \cdots + (-355)}_{203\text{ 個}} = 142 \quad (< 355)
$$

しかしながら、この計算方法も効率の良いとは言えません。そこで、掛け算に習って、シフト演算が利用できるように、

$$
\begin{aligned}
&72207 + (-355) \times 203 \\
&\quad = 72207 + \big((-355) \times 10^2\big) \times 2 + \big((-355) \times 10^1\big) \times 0 + \big((-355) \times 10^0\big) \times 3 \\
&\quad = 72207 + \big(((-355) + (-355)) \times 10^2\big) + \big(0 \times 10^1\big) + ((-355) + (-355) + (-355)) \\
&\quad = 72207 + (((-355) + (-355)) \underline{\times 10} + 0) \underline{\times 10} + ((-355) + (-355) + (-355))
\end{aligned}
$$

と変形し、上式の下線部をシフト演算に置き換えて計算します。すなわち、次の①から⑥と足した回数をカウントする①' から⑥' の手順によって商 203 を効率的に求めることが出来ます ($355 \times x$ が、72207 を超えないような最大の x を求めています)。

[13] 実際にコンピュータ内部で行われる掛け算は、もっと巧みに足し算とシフト演算を使って機械的に計算されます。ただし、基本的な考え方は本テキストで述べた通りです。

[準備] 72207 と同じ桁数になるまで 355 に繰り返し 1 回左シフト演算を施し、
その回数を求める (**2 回**)

① (-355) に対して**2 回**左シフト演算を行う $((-355) \times 10^2 = (-35500))$
①' 1 に対して**2 回**左シフト演算を行う $(1 \times 10^2 = 100)$

② 72207 に (-35500) を 2 回足す $(72207 + (-35500) + (-35500) = 1207)$
②' 0 に 100 を②と同じく 2 回足す $(0 + 100 + 100 = 200)$

③ (-355) に対して**1 回**左シフト演算を行う $((-355) \times 10^1 = (-3550))$
③' 1 に対して**1 回**左シフト演算を行う $(1 \times 10^1 = 10)$

④ 1207 に (-3550) を 0 回足す $(1207 + 0 = 1207)$
④' 200 に 10 を④と同じく 0 回足す $(200 + 0 = 200)$

⑤ (-355) に対して**0 回**左シフト演算を行う $((-355) \times 10^0 = (-355))$
⑤' 1 に対して**0 回**左シフト演算を行う $(1 \times 10^0 = 1)$

⑥ 1207 に (-355) を 3 回足す $(1207 + (-355) + (-355) + (-355) = 142)$
⑥' 200 に 1 を⑥と同じく 3 回足す $(200 + 1 + 1 + 1 = 203)$

割り算についても掛け算と同様に、上記で紹介した方法は、下記のような私たちがふだん何気なく行っている割り算の方法とあまり変わりがないことがわかります。

```
          2 0 3
      ------------
3 5 5 ) 7 2 2 0 7
        7 1 0 ←←   ・・・ ①
      ------------
          1 2 0
              0 ←  ・・・ ③
      ------------
          1 2 0 7
          1 0 6 5  ・・・ ⑤
      ------------
            1 4 2
```

2 進数の場合も同様に考えると、例えば、1101110(2) ÷ 1010(2) を

$$
\begin{aligned}
&1101110(2) + (-1010(2)) \times 1011(2) \\
&= 1101110(2) + (-1010(2)) \times 1 \times 2^3 + (-1010(2)) \times 0 \times 2^2 \\
&\qquad + (-1010(2)) \times 1 \times 2^1 + (-1010(2)) \times 1 \times 2^0 \\
&= 1101110(2) + (((-1010(2)) \underline{\times 2} + 0(2)) \underline{\times 2} + (-1010(2))) \underline{\times 2} + (-1010(2))
\end{aligned}
$$

と変形することにより、以下の手順で計算すると商 1011(2) が求めることが出来ます[14]。

[14]実際にコンピュータ内部で行われる割り算は、もっと巧みに足し算とシフト演算を使って機械的に計算されます。ただし、基本的な考え方は本テキストで述べた通りです。

[準備] 1101110(2) と同じ桁数になるまで 1010(2) に繰り返し 1 ビット左シフト演算を施し、その回数を求める (**3 回** (ビット))

① $(-1010(2))$ に対して**3 ビット**左シフト演算を行う $((-1010(2)) \times 2^3 = (-1010000(2)))$

①' $1(2)$ に対して**3 ビット**左シフト演算を行う $(1(2) \times 2^3 = 1000(2))$

② 1101110(2) に $(-1010000(2))$ を 1 回足す $(1101110(2) + (-1010000(2)) = 11110(2))$

②' $0(2)$ に $1000(2)$ を②と同じく 1 回足す $(0(2) + 1000(2) = 1000(2))$

③ $(-1010(2))$ に対して**2 ビット**左シフト演算を行う $((-1010(2)) \times 2^2 = (-101000(2)))$

③' 1 に対して**2 ビット**左シフト演算を行う $(1(2) \times 2^2 = 100(2))$

④ 11110(2) に $(-101000(2))$ を 0 回足す $(11110(2) + 0(2) = 11110(2))$

④' $1000(2)$ に $100(2)$ を④と同じく 0 回足す $(1000(2) + 0(2) = 1000(2))$

⑤ $(-1010(2))$ に対して**1 ビット**左シフト演算を行う $((-1010(2)) \times 2^1 = (-10100(2)))$

⑤' 1 に対して**1 ビット**左シフト演算を行う $(1(2) \times 2^1 = 10(2))$

⑥ 11110(2) に $(-10100(2))$ を 1 回足す $(11110(2) + (-10100(2)) = 1010(2))$

⑥' $1000(2)$ に $10(2)$ を⑥と同じく 1 回足す $(1000(2) + 10(2) = 1010(2))$

⑦ $(-1010(2))$ に対して**0 ビット**左シフト演算を行う $((-1010(2)) \times 2^0 = (-1010(2)))$

⑦' 1 に対して**0 ビット**左シフト演算を行う $(1(2) \times 2^0 = 1(2))$

⑧ 1010(2) に $(-1010(2))$ を 1 回足す $(1010(2) + (-1010(2)) = 0(2))$

⑧' 1010(2) に $1(2)$ を⑧と同じく 1 回足す $(1010(2) + 1(2) = 1011(2))$

これまで (2.5 節と合わせて)、足し算・引き算・掛け算・割り算の全ての四則演算を足し算とシフト演算のみで演算する方法を学んできましたが、さらに第 3 章の論理演算を学習し、第 4 章で実際のコンピュータ内部でどのように計算が行われている (論理回路で実現されている) のかを学びます。

問題 1　11111111(2) に対して 4 ビット右シフト演算を行うといくつになるか、10 進数で答えなさい。

問題 2　11111111(2) に対して 4 ビット左シフト演算を行うといくつになるか、10 進数で答えなさい。

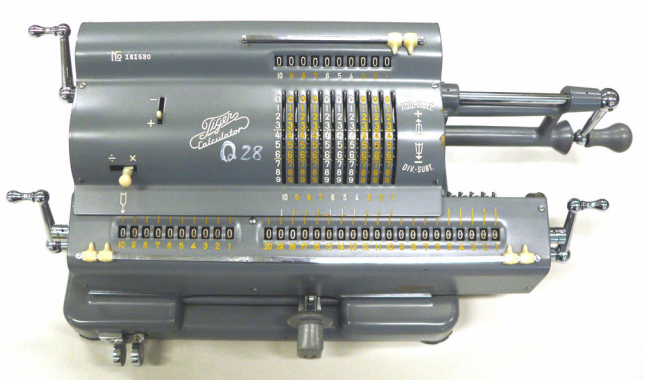
問題 3　60752(8) ÷ 123(8) を例にならって足し算とシフト演算のみを用いて計算しなさい。

問題 4　A45F(16) × CDE(16) を例にならって足し算とシフト演算のみを用いて計算しなさい。

● コーヒーブレイク — タイガー手廻し計算器 —

タイガー手廻し計算器は、1919 年 (大正 8 年) に国内計算器として開発がスタートし、1923 年には「虎印計算器」として商品化されました。その後、「タイガー計算器」と名称を変え、正確かつ大量の計算が必要な建築・財務・研究機関等で使用されましたが、昭和 40 年代初頭に半導体を使用した電卓が登場すると、その姿を消していきました。

タイガー手廻し計算器とコンピュータの数値計算の仕組み (補数とシフト演算) は全く同じで、コンピュータの基礎を学ぶには大変役に立ちます。筆者ホームページ「http://kouyama.math.u-toyama.ac.jp/main/computer/personal/tiger/index.htm」に、ウェブブラウザ上で動く、タイガー手廻し計算器のシミュレータがあるので、是非おためし下さい。



＊さらに詳しい情報については、タイガー手廻し計算器を製造・販売してきたタイガー計算器株式会社 (現在は「株式会社タイガー」) のホームページ「http://www.tiger-inc.co.jp」をご覧ください。

2.7　数値データ

コンピュータ内部で直接扱うことのできる数値データの表示方法は、構造の違いにより 2 つに分類され、それぞれ**固定小数点表示**と**浮動小数点表示**と呼ばれています。なお、各表示方法によって表示された数を、それぞれ**固定小数点数**と**浮動小数点数**と呼びます。

固定小数点表示　この表示方法は、主に整数を表示する場合に用いられ、1 ワードを 8, 16, 32, 64 などのビット数で表します。例えば、k ビットで表すことのできる 0 と 1 の組み合わせの数は 2^k 通りですから、自然に 2 進数に対応させれば 0 から $2^k - 1$ までの整数を表すことができます。また、補数表現された 2 進数に対応させれば -2^{k-1} から $2^{k-1} - 1$ までの整数を表すことができます。どちらも、図 2.4 のように小数点が第 0 ビットの右側に固定されています。

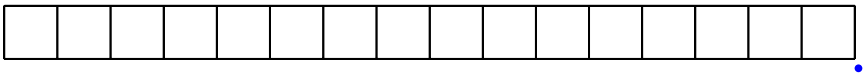


図 2.4: 固定小数点表示の小数点の位置

以前は、小数点の位置を任意に固定して小数を表していましたが、現在では、固定小数点表示と浮動小数点表示で行う演算の処理時間に差がなくなったことと、浮動小数点表示でも十分な精度が得られるようになったため、小数は浮動小数点表示で扱われるようになりました。逆にいえば、上記の例は、小数点が第 0 ビットの右側に固定されている特殊な場合といえます。

浮動小数点表示　この表示方法は、実数を表示する場合に用いられ、固定小数点表示に比べ非常に大きな数や非常に小さな数を表示するのに適しています。科学の分野では、

$$-1.0 \times 10^{16}, \quad +2.5 \times 10^{-8}, \quad +3.33 \times 10^{10}$$

といった表示方法を用いますが、これが浮動小数点表示です。この表示方法に習って、浮動小数点表示を

$$S\, M \times B^E$$

のように表すと、S は**符号** (sign)・M は**仮数** (mantissa)・E は**指数** (exponent)・B は**底** (base) と呼ばれ、図 2.5 のような構造でコンピュータ内に記憶されます (通常、底には 2 が使用され、決まった数なので省略されます)。なお、現在使用されている浮動小数点表示には精度によって、1 ワードを 32 ビットで表す**単精度浮動小数点表示**と、1 ワードを 64 ビットで表す**倍精度浮動小数点表示**があります。

S ： **符号部**と呼ばれ、符号を 1 ビットで表します (正 → 0, 負 → 1)。
E ： **指数部**と呼ばれ、指数を 2 進数で**イクセス表現**します。
M ： **仮数部**と呼ばれ、仮数を 2 進数で表現します (補数表現しない)。

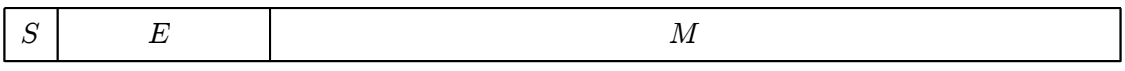


図 2.5: 浮動小数点表示の構造

浮動小数点表示について、さらに詳しく見ていきましょう。

現在、主に使用されている浮動小数点表示は、1985 年に米国電気電子技術者協会 (the Institute of Electrical and Electronic Engineers inc.; 通称アイトリプルイー) が提唱し規格化した、「**IEEE754**」[15]で、高精度の数値計算[16]に適しています。**IEEE754 形式**の単精度浮動小数点表示および倍精度浮動小数点表示の指数部と仮数部のビット数は表 2.12 のようになっています。また、指数部の**イクセス表現**に必要な**バイアス**は、**単精度で 127**、**倍精度で 1023** となっています (詳細は下記の注意を参照すること)。本テキストも、断りのない限り IEEE754 形式に従います。

	単精度	倍精度
符号部	1 ビット	1 ビット
指数部	8 ビット	11 ビット
仮数部	23 ビット (**実質 24 ビット**)	52 ビット (**実質 53 ビット**)
合計	32 ビット	64 ビット

表 2.12: IEEE754

【注意】IEEE754 形式では、指数部の**全てのビットが 0 の場合**と**全てのビットが 1 の場合**は特別な状態を表しています。そのため単精度 IEEE754 形式のバイアスは 127 ($= 2^8/2-1$) となります (倍精度の場合は 1023 ($= 2^{11}/2-1$))。また、指数部で表現可能な数は -126 から 127 までとなります (倍精度の場合は -1022 から 1023 まで)。なぜなら、$(-126)+127=1$ ($=$ `00000001(2)`) かつ $127+127=254$ ($=$ `11111110(2)`) となり、全てのビットが `0` または `1` という状態にならないからです。以下に単精度 IEEE754 形式の特殊な状態を挙げておきます (ただし、「`-`」は `1` または `0` のいずれかを表します)。

`0`	`00000000`	`00000000000000000000000`	+0 (浮動小数点表示としての 0)
`1`	`00000000`	`00000000000000000000000`	−0 (浮動小数点表示としての 0)
`-`	`00000000`	`-----------------------`	正規化されていない数
`-`	`11111111`	`00000000000000000000000`	無限大
`-`	`11111111`	`-----------------------`	数を表さない (無限大を除く)

* 詳細については、URL「**http://docs.sun.com/source/806-4836/ANSIC.html**」の極値表現などを参照してください。

[15]その他にも、IBM System360 以来、メインフレーム (汎用コンピュータ) で採用された「**IBM 形式**」があり、底を 16・指数部を 8 ビット・単精度および倍精度の仮数部をそれぞれ 23 と 56 ビットで表示します。

[16]数値計算では、近似値による計算がほとんどで、たとえ高精度であっても必ず誤差が生じます。

浮動小数点表示を行う上で、もう 1 つ注意しなければならないことがあります。コンピュータは有限桁の 2 進数しか扱うことが出来ないため、数値計算などで出来るだけ高精度の結果が得られるように、浮動小数点数の仮数部を一意かつ有効桁数が多くなるように表示する必要があります。例えば、$\frac{1}{3}$ を浮動小数点数で表示すると $33.33\cdots\times10^{-2}$, $3.333\cdots\times10^{-1}$, $0.3333\cdots\times10^{0}$ のように異なる形で書き表すことができます。この中で、$3.333\cdots\times10^{-1}$ の形をした表示を**正規形**と呼びます。すなわち、基数を r としたとき、基数記数法による表現で

$$\pm1\times(a_0\times r^0+a_{-1}\times r^{-1}+a_{-2}\times r^{-2}+\cdots)\times r^k \qquad (\text{ただし、} a_0\neq0)$$

の形をしていることです (位取り記数法による表現は $\pm a_0.a_{-1}a_{-2}\cdots\times r^k$)。また、一般的な浮動小数点数から正規形に変換することを**正規化**といいます。以下に正規形の例を挙げておきます。

$$1.23\times10^{-10},\quad -5.11\times10^{17},\quad 1.01011_{(2)}\times2^{-3},\quad -1.1111_{(2)}\times2^{1010_{(2)}}$$

以上を踏まえて、$0.000001010111_{(2)}$ を単精度 IEEE754 形式で表示してみましょう。まず、

$$\begin{aligned}1.010111_{(2)}\times2^{-6} &= +1.010111_{(2)}\times2^{(-6+127)-127}\\ &= +1.010111_{(2)}\times2^{01111001_{(2)}-127}\end{aligned}$$ ← 2 進数 8 桁でイクセス表現

のように正規形に変形します。ここで、仮数の先頭は必ず $1.\cdots$[17] となるので、**IEEE754 形式ではさらに有効数字が長くなるように「1.」を省略して表示します** (単精度 IEEE754 形式の仮数部は**実質 24 ビット**になります)。従って、単精度 IEEE754 形式のコンピュータ内での表示は下図のようになります。

0	01111001	01011100000000000000000
		∧
		1.

例題 1　$-110110101_{(2)}$ を単精度 IEEE754 形式で表示しなさい。

解答例　$-110110101_{(2)}$ を正規形に直すと

$$\begin{aligned}-110110101_{(2)} &= -1.10110101_{(2)}\times2^{8}\\ &= -1.10110101_{(2)}\times2^{(8+127)-127}\\ &= -1.10110101_{(2)}\times2^{10000111_{(2)}-127}\end{aligned}$$

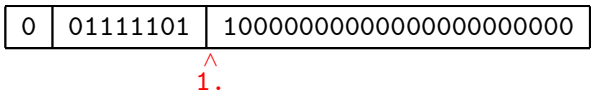
となる。従って、仮数の符号が負 (符号部を 1 にする) であることに注意して、単精度 IEEE754 形式で表示すると

1	10000111	10110101000000000000000
		∧
		1.

になる。

[17] 0 は除きます。

例題 2 以下の単精度 IEEE754 形式で表示された数を正規形の 10 進数で答えなさい。



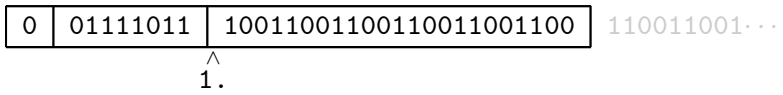
解答例

$$
\begin{aligned}
&1.10000000000000000000000(2) \times 2^{01111101(2)-127} \\
&= (1(2) + 0.1(2)) \times 2^{125-127} \\
&= (1 + 0.5) \times 2^{-2} \\
&= 1.5 \times 0.25 \\
&= 0.375 \\
&= 3.75 \times 10^{-1}
\end{aligned}
$$

もう一つ例として、10 進数 0.1 を単精度 IEEE754 形式で表示して見ましょう。まず、2.2 節の例題 2 で求めたように 10 進数 0.1 を 2 進数に直すと循環小数 $0.0001100110011\cdots(2)$ となります。次に、これを正規形に変形すると

$$1.100110011\cdots(2) \times 2^{-4} = +1.100110011\cdots(2) \times 2^{01111011(2)-127}$$

となります。従って、単精度 IEEE754 形式のコンピュータ内での表示は下図のようになります。ただし、ココでは**収まらなかった桁については切り捨てる**ものとすれば、元の値とは異なった値になります[18]。



逆に、単精度 IEEE754 形式で表示された数を 10 進数に直してみましょう。上図から、

$$
\begin{aligned}
&1.10011001100110011001100(2) \times 2^{01111011(2)-127} \\
&= 1.10011001100110011001100(2) \times 2^{123-127} \\
&= 11001100110011001100(2) \times 2^{-23} \times 2^{-4} \\
&= 13421772 \times 2^{-27} \\
&= 13421772 \div 134217723 \\
&= 0.09999999403953552246093750
\end{aligned}
$$

となります。このように、コンピュータに計算させるために 10 進数から浮動小数点数に変換したり、逆に浮動小数点数から 10 進数に変換すると少なからず**誤差**[19]を生じます。

[18] 切り上げするにしろ、四捨五入をするにせよ、無限の桁を有限の桁で表すことはできません。
[19] これらに起因する誤差やその他の誤差については、後ほどまとめて詳しく解説します。

単精度 IEEE754 形式で表示可能な数の範囲と仮数の有効桁数は次のようにして求められます (倍精度 IEEE754 形式については問題 4)。単精度 IEEE754 形式の指数の範囲に注意すると、表示可能な数の範囲は

$$-2^{127} \sim -2^{-126} \quad と \quad 0 \quad と \quad +2^{-126} \sim +2^{127}$$

となります。10 進数で近似すると

$$-10^{38} \sim -10^{-38} \quad と \quad 0 \quad と \quad +10^{-38} \sim +10^{38}$$

となります。2 進法から 10 進法への変換は、

$$2^{127} = 10^{x}$$

と置き、両辺に常用対数を取ることで

$$x = \log_{10} 2^{127} = 127 \times 0.301\cdots = 38.227\cdots \fallingdotseq 38 \quad (\log_{10} 2 = 0.301\cdots)$$

が求められます (同様に、$\log_{10} 2^{-126} = -126 \times 0.301\cdots = -37.929\cdots \fallingdotseq -38$)。また、単精度 IEEE754 形式の仮数の有効桁数は、仮数が 24 $(= 23 + 1)$ ビットで表示されることから、10 進数で約 7 桁に相当します。2 進法から 10 進法への変換は、

$$2^{24} = 10^{y}$$

と置き、両辺に常用対数を取ることで

$$y = \log_{10} 2^{24} = 24 \times 0.301\cdots = 7.224\cdots$$

が求められます。なお、図 2.6 のように、表示可能な絶対値の一番大きな数 $|\pm 2^{127}|$ より絶対値の大きな数を数値データとしてコンピュータに記憶させようとすると**オーバーフロー** (over flow) という状態になり、表示可能な絶対値の一番小さな数 $|\pm 2^{-126}|$ より絶対値の小さな数を記憶させようとすると**アンダーフロー** (under flow) という状態になります。

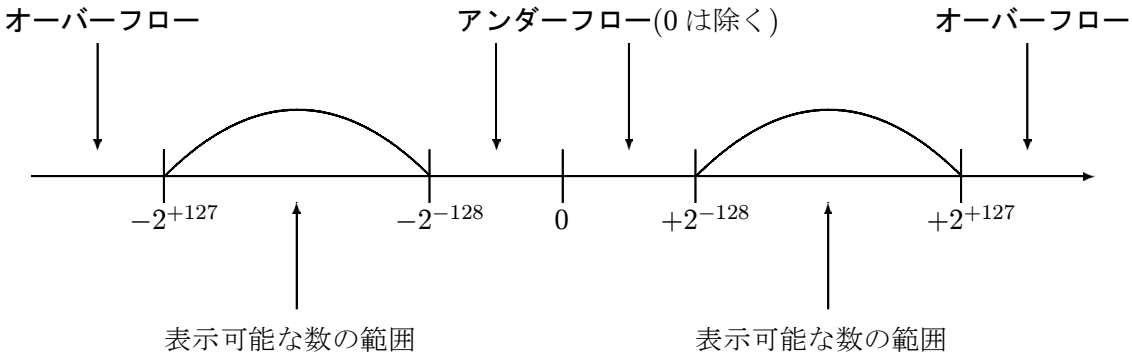
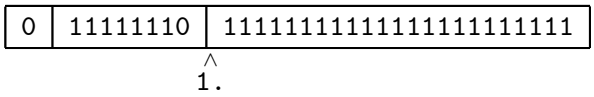


図 2.6: オーバーフローとアンダーフロー

最後に、単精度 IEEE754 形式で表示可能な数の限界値について述べておきます。

厳密な単精度 IEEE754 形式で表示可能な絶対値の一番大きな (正の) 数 厳密な単精度 IEEE754 形式で表示可能な絶対値の一番大きな (正の) 数は

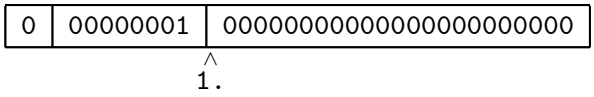


で、正規形の 10 進数に直すと

$$\begin{aligned}
&\texttt{1.11111111111111111111111(2)} \times 2^{\texttt{11111110(2)}-127} \\
&\quad = \texttt{1.11111111111111111111111(2)} \times 2^{127} \\
&\quad = \texttt{11111111111111111111111(2)} \times 2^{-23} \times 2^{127} \\
&\quad = (2^{24}-1) \times 2^{104} \\
&\quad = 340282346638528859811704183484516925440 \\
&\quad = 3.402823466385288598117041834845169254 40 \times 10^{38}
\end{aligned}$$

となります。

厳密な単精度 IEEE754 形式で表示可能な絶対値の一番小さな (正の) 数 厳密な単精度 IEEE754 形式で表示可能な絶対値の一番小さな (正の) 数は



で、正規形の 10 進数に直すと

$$\begin{aligned}
&\texttt{1.00000000000000000000000(2)} \times 2^{\texttt{00000001(2)}-127} \\
&\quad = 1 \times 2^{1-127} \\
&\quad = 2^{-126} \\
&\quad = \texttt{1.17549435082228750796873653722224567781866555677208} \\
&\qquad \texttt{75215087517062784172594547271728515625} \times 10^{-38}
\end{aligned}$$

となります。

問題 1 10 進数 12345 と 10 進数 −0.12345 を単精度 IEEE754 形式で表示しなさい。

問題 2 以下の単精度 IEEE754 形式で表示された数を正規形の 10 進数で答えなさい。

1	10000101	00110000000000000000000

問題 3 以下の単精度 IEEE754 形式で表示された数を正規形の 10 進数で答えなさい。

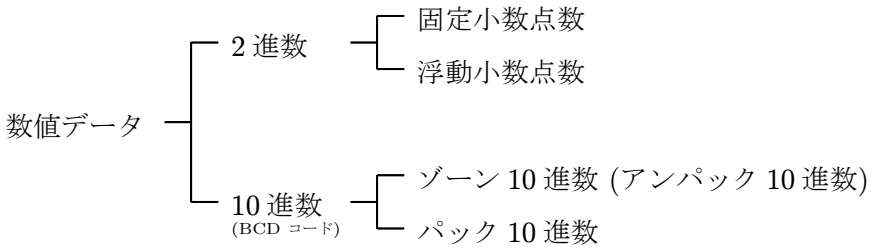
0	01111100	01100000000000000000000

問題 4 単精度 IEEE754 形式に習って、倍精度 IEEE754 形式で表示可能な数の範囲と仮数の有効桁数を求めなさい。

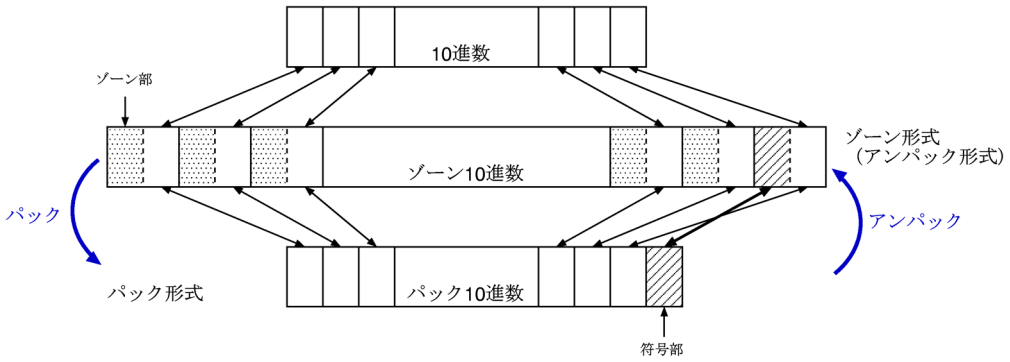
問題 5　単精度 IEEE754 形式に習って、倍精度 IEEE754 形式で表示可能な絶対値の一番大きな (正の) 数と絶対値の一番小さな (正の) 数を正規形の 10 進数で答えなさい。

● 参考資料 — BCD コード —

コンピュータ内部での数値表示には、固定小数点表示や浮動小数点表示以外にも、10 進数を表示するのに適した **BCD コード** (binary coded decimal notation code) があります。主なもとしては、**ゾーン 10 進数** (**アンパック 10 進数**) と**パック 10 進数**があります。これらの表示方法は、大量の桁の 10 進数を正確に扱うことができ、財務など多額の金額を正確に扱う必要のあるビジネス分野などで利用されています。



下図のように、**ゾーン形式** (**アンパック形式**) によるゾーン 10 進数は、10 進数の 1 桁分を 1 バイトで表しますが、上位 4 ビットはゾーン部として定数が入り、下位 4 ビットには 0000(2) = 0 から 1001(2) = 9 までのいずれかの値が入ります (ただし、10 進数の 1 桁目に相当する部分のゾーン部は符号を表す)。逆に、**パック形式**によるパック 10 進数は、ゾーン 10 進数からゾーン部を取りのぞき、無駄な部分を省いた形で表します (ただし、後ろに 4 ビット付加して符号を表す)。なお、ゾーン 10 進数からパック 10 進数への変換をパックと呼び、逆の変換をアンパックと呼びます (下図参照)。



関連キーワード： **3 増し符号** (excess 3 code)