O liczbach w komputerze

Literatura: Java™ Number Cruncher: The Java Programmer's Guide to Numerical Computing by Ronald Mak

1. Liczby całkowite

Załóżmy, ż pracujemy w systemie dwójkowym na słowie 4 bitowym (nybble, nibble; pół bajta). W takim słowie możemy zapisać liczby od -7 do 7 jeśli umówimy się, ze zero w pierwszym bicie oznacza liczbę dodatnią, a jedynka ujemną (system znak-moduł). Mamy więc liczby dodatnie

```
 0  0000
 1  0001
 2  0010
 3  0011
 4  0100
 5  0101
 6  0110
 7  0111
```

oraz liczby ujemne

```
-0  1000
-1  1001
-2  1010
-3  1011
-4  1100
-5  1101
-6  1110
-7  1111
```

Wadą tego systemu jest to, że mamy dwie reprezentacje zera (0000 i -10000) oraz, że nie możemy korzystać ze zwykłej arytmetyki dwójkowej. Np.,

```
  6    0110      -3  1011
+-3    1011      -2  1010
 ----------      --------
  1!   0001      5!  0101
```

Widzimy, że wyniki dodawania są złe! Potrzebna jest więc inna reprezentacja liczb całkowitych. Stosuje się tzw. reprezentację uzupełniania do dwójki. Znak liczby określa się jak poprzednio, lecz liczby ujemne otrzymuje się uzupełniając każdy bit do 2, a następnie dodając jedynkę do całości. Np.

```
startujemy z +6:     0110
uzupełniamy bity:    1001
dodajemy 1:             1
otrzymujemy:         1010
```

Reprezentacja liczb ujemnych wygląda więc, następująco:

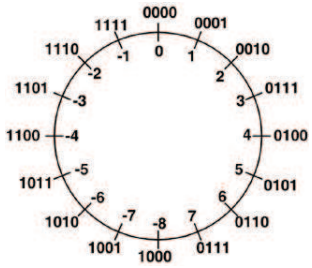
```
-1  1111
-2  1110
-3  1101
-4  1100
-5  1011
-6  1010
-7  1001
-8  1000
```

W reprezentacji uzupełniania do 2 jest tylko jedno zero (0000) i dodatkowo mamy jedną liczbę -8 więcej.

Poprzednio wykonane dadawanie jest teraz prawidłowe:

```
  6    0110      -3  1011
+-3    1101     +-2  1110
 ----------      --------
  3    0011      -5  1001
```

PRZYKŁAD. Porównać dodawanie na osi liczbowej z dodawaniem na tarczy koła gdzie liczby z naszego systemu tworzą ciąg 1, ..., 7, -8, -7, ..., 0 (0001, 0010, ... ,0111, 1111, 1001,..., 1110, 1111). Podobnie jest w systemach z większą liczbą bitów przeznaczonych na słowo liczbowe.



Rysunek 1: Liczby na tarczy zegara ... w kodzie uzupełnieniowym do 2.

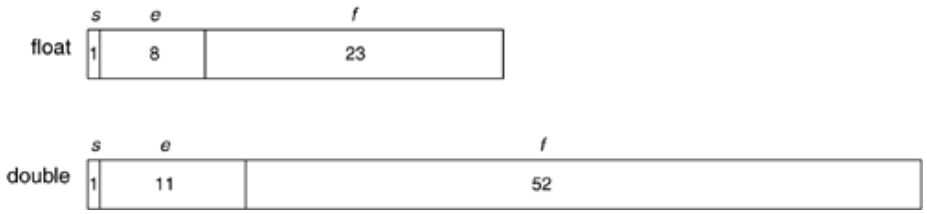
Tablica 1: Liczby całkowite w IEEE 754 (np. Java).

Typ	Długość (bity)	Wartość min	Wartość max
byte	8	-128	127
short	16	-32,768	32,767
char	16	0	65,535
int	32	-2,147,483,648	2,147,483,647
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

2. Standard IEEE-754

W pamięci komputera liczby zmiennoprzecinkowe reprezentowane są przez ich wartości przybliżone - pewnych liczb nie da się zapisać w systemie zmiennoprzecinkowym. Typowym przykładem są ułamki dziesiętne typu 1/10, 3/10. Takie ułamki posiadają nieskończone rozwinięcia w systemie dwójkowym, zatem wymagają mantys o nieskończonej ilości bitów. Ilość bitów mantysy określa precyzję zapisu. Im jest większa, tym dokładniej można przedstawiać liczby rzeczywiste - błędy zaokrągleń są mniejsze. Ilość bitów cechy określa dopuszczalny zakres reprezentowanych liczb.

Rysunek 2 przedstawia schematycznie formaty liczb zmiennoprzecinkowych standardu IEEE 754.



Rysunek 2: Format liczb zmiennoprzecinkowych standardu IEEE 754. Oprócz bitu znaku s podana jest liczba bitów cechy (e) i mantysy (f) liczby.

Standard IEEE 754 ogłoszono w roku 1985. Standard określa formaty, operacje, konwersje i wyjątki. Formaty opisują sposoby kodowania liczb w pamięci komputera. Są zdefiniowane dwa format liczb zmiennoprzecinkowych: zwykły (*float; real, ...*) i podwójnej precyzji (*double; real*8, double precision*). Format float zapisuje się na 32, a double na 64 bitach. Formaty różnią się liczbą bitów w części mantysy (f) i cechy liczby (e) (liczba = mantysa $\times$ b^{cecha}; b – podstawa systemu liczbowego; $b = 2$ dla systemu binarnego).

2.1. Liczby znormalizowane

Liczbę zmiennoprzecinkową można reprezentować jako liczbę znormalizowaną (jeśli e nie jest równe zarezerwowanej wartości 0 i 255 dla float lub 0 i 2047 dla double i dodatkowo mamy umowną jedynkę i kropką dziesiętną przed f):

$$L = (-1)^s \times 2^E \times (1.f)$$

gdzie s jest bitem znaku liczby (0 – dodatnia, 1 – ujemna). Prawdziwy wykładnik E liczby (cecha) jest zakodowany i jest bez znaku. W przypadku gdy wykładnik ma reprezentować potęge ujemną ustala się dodatnie przesunięcie (ang. *bias*). Właściwą wartość wykładnika E uzyskuje się odejmując od zakodowanej wartości wykładnika e wartość przesunięcia (*bias*). W formacie liczb w pojedynczej precyzji wykładnik zajmuje 8 bitów co odpowiada liczbom dodatnim z zakresu 0–255 lecz 0 i 255 są zarezerwowane. Przesunięcie wynosi 127, a więc wykładnik zmienia się w zakresie od -127 do 127. W formacie precyzji podwójnej na wykładnik rezerwuje się 11 bitów co odpowiada zakresowi 0–2047, a ponieważ liczby 0 i 2047 są zarezerwowane to przesunięcie jest równe 1023 i wykładnik E zmienia się w granicach od -1023 do 1023.

PRZYKŁAD.

$$\begin{aligned} s &= 0 \\ e &= 125\,, \quad \text{zakodowany wykładnik} \\ f &= \text{binarnie}10000000000000000000000 \end{aligned}$$

wówczas

$$E = 125 - 127 = -2$$

co po dodaniu umownej jedynki na początku i umownej kropki dziesiętnej odpowiada zapisowi 1.10000000000000000000000. Po przesunięciu kropki o dwa miejsca w lewo otrzymujemy 0.011, a więc

$$2^{-2} + 2^{-3} = \frac{1}{4} + \frac{1}{3} = 0.375\,.$$

Maksymalna liczba zmiennoprzecinkowa float posiada

$$\begin{aligned} s &= 0 \\ e &= 254 \\ f &= \text{binarnie}11111111111111111111111 \end{aligned}$$

Stąd

$$E = 254 - 127 = 127\,,$$

znacznik liczby jest więc równy 1.11111111111111111111111, a jej wartość wynosi:

$$(2^{127} \times \sum_{i=0}^{23} 2^{-i} = 2^{127} \times 2 = 3.4 \times 10^{38}\,.$$

Kładąc $s = 1$ otrzymamy liczbę najmniejszą w tym systemie, $-3.4\text{x}10^{38}$.

2.2. Liczby zdenormalizowane

Jeżeli $e = 0$ i $f \neq 0$ wówczas mamy tzw. liczby *zdenormalizowane*. W znaczniku liczby mamy kropkę dziesiętną na lewo od f oraz umowny bit zerowy na lewo od kropki dziesiętnej. Wartość liczby otrzymamy przesuwając kropkę dla float o 126 miejsc w lewo, a dla double o 1022 bity na lewo. Bit znaku s określa czy liczba jest ujemna czy dodatnia.

$$\text{float} \quad L = (-1)^s \times 2^{-126} \times (0.f)\,, \qquad \text{double} \quad L = (-1)^s \times 2^{-1022} \times (0.f)\,.$$

Przykład. Float. Niech

$$\begin{aligned} s &= 0 \\ e &= 0 \\ f &= \text{binarnie}00101000000000000000000 \end{aligned}$$

Znacznik binarny, po wstawieniu umownego zera i kropki, jest więc 0.00101000000000000000000. Po przesunięciu kropki o 126 miejsc w lewo otrzymamy

$$(2^{-129} + 2^{-131} \approx 1.84 \times 10^{-39}.$$

Tablica 2: Format liczb standardu IEEE 754.

	Przesunięcie	Zakres E	Rozmiar znacznika	Min	Max
float	127	-126 – 127	24 bity	$\pm 1.4 \times 10^{-45}$	$\pm 3.4028235 \times 10^{+38}$
double	1023	-1022 – 1023	53 bity	$\pm 4.9 \times 10^{-324}$	$\pm 1.7976931348623157 \times 10^{+308}$

Zadanie 1. Wartość minimalna dodatnia float 2

Znaleźć najmniejszą liczbę dodatnią typu float. [*Odp.*][*Spis*]

Zadanie 2. Wartość minimalna ujemna float 2

Znaleźć największą liczbę ujemną typu float. [*Odp.*][*Spis*]

Zadanie 3. …double … 2

To samo co wyżej dla liczb typu double [*Spis*]

2.3. Inne liczby

W standardzie IEEE 754 określono kilka przypadków wartości stałych.

2.3.1. Zero

Jeśli e i f są zerowe wówczas, w zależności od s wartość liczby jest albo -0 albo $+0$:

$$L = (-1)^s \times 0 .$$

2.3.2. `Infinity` – nieskończoność

Jeśli e jest równe 255 (float) lub 2047 (double) – wartości zarezerwowane i $f = 0$ to wartość liczby jest `Infinity` lub `-Infinity`, zależnie od s.

2.3.3. NaN

Jeśli e jest równe 255 (float) lub 2047 (double) i $f \neq 0$ to mamy `NaN` – Not a Number – liczba nieokreślona. Przykład: Dzielenie zera przez zero daje `NaN`.

3. EPSILON

Liczba EPSILON maszynowe jest największą liczbą dodatnią, która dodana do jedynki daje jedynkę: $1 + \text{EPSILON} = 1$ (w arytmetyce komputera).

Poniższy program (FORTRAN) oblicza EPSILON dla typu float/real oraz double-/doubleprecision.

```
program Epsilon

implicit none
real floatEpsilon, fTemp;
doubleprecision doubleEpsilon, dTemp;
!
! Compute the machine epsilon for the float and double types,
! the largest positive floating-point value that, when added to 1,
! results in a value equal to 1 due to roundoff.
!
! Loop to compute the float epsilon value.
fTemp = 0.5;
do while (1 + fTemp > 1)
   fTemp = fTemp/2
end do
floatEpsilon = fTemp

! Loop to compute the double epsilon value.
dTemp = 0.5;
do while (1 + dTemp > 1)
   dTemp = dTemp/2
end do
doubleEpsilon = dTemp

! output the float epsilon value.
write(*,*) "floatEpsilon  =", floatEpsilon

! output the double epsilon value.
write(*,*) "doubleEpsilon =",doubleEpsilon

end program epsilon

-----------------
```

```
WYNIKI:
floatEpsilon  =  5.96046448E-08
doubleEpsilon =  1.11022302462515654E-016
```

Spis zadań