Dodatek do Wykładu 01: Kodowanie liczb w komputerze

[materiał ze strony: http://sigma.wsb-nlu.edu.pl/~szyszkin/]

Wszelkie dane zapamiętywane przetwarzane przez komputery muszą być odpowiednio zakodowane. Z powodów technicznych najwygodniej jest realizować układy, w których są rozróżnialne dwa stany. Stany te będziemy dla wygody oznaczali symbolami 0 i 1. Tak więc w pamięci komputera znajdują się same ciągi zer i jedynek. Na ogół są one grupowane w pewne większe porcje: osiem bitów to *bajt* (zwany czasami oktetem), szesnaście bitów to pojedyncze słowo. Dokładna terminologia zależy przede wszystkim od architektury danego sytemu komputerowego (głównie chodzi o procesor i pamięć), ale bajt jest standardem. Tak więc w komputerze ośmiobitowym w pamięci mamy komórki, w których znajdują się porcje po osiem bitów (np. jedne z pierwszych procesorów firmy Intel (1974 r.) – oznaczane jako 8008, 8080 – były ośmiobitowe[1]). Tak więc fragment pamięci możemy sobie wyobrazić następująco:

1	0	0	0	1	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	1	0	1	0
0	1	0	1	0	0	1	1

Z drugiej strony, aby efektywnie przetwarzać dane – np. używając języków programowania do pisania odpowiednich programów – potrzebujemy bardziej abstrakcyjnych pojęć takich, jak liczby całkowite, liczby wymierne (z „częścią po przecinku dziesiętnym”), znaki tekstowe. Wszystkie te obiekty muszą być jednak zapamiętane w postaci ciągów zer i jedynek.

Aby to zrobić efektywnie należy umówić się co do sposobu kodowania tych obiektów. Jak reprezentować liczby 20, -31, 124.12, symbole ‘A’, ‘a’ czy ‘x’?

Mimo, że od strony czysto matematycznej liczby wymierne są rozszerzeniem zbioru liczb całkowitych, to w technice komputerowej stosuje się zupełnie inne sposoby kodowania dla liczb całkowitych i liczb wymiernych. Stąd w większości języków programowania rozróżnia się generalnie typy całkowitoliczbowe (np. **`int`**, **`short`** w języku C/C++) od typów nie całkowitoliczbowych (np. **`float`**, **`double`** w języku C/C++). Ze względu na sposób reprezentacji używa się tutaj często określenia typy *stałoprzecinkowe* i typy *zmiennoprzecinkowe*.

Reprezentacje liczb całkowitych

Kodowanie NKB i U1

Podstawą jest tutaj tzw. naturalny kod binarny (NKB). Jest to reprezentacja analogiczna do używanej na co dzień, gdy posługujemy się liczbami całkowitymi, czyli zapis pozycyjnym o podstawie 10. Na przykład w układzie dziesiętnym zapis 24564 znaczy tyle co

[1] Podstawowe rejestry tych dwóch procesorów były 8-bitowe, ale były one w stanie używać adresów 16-bitowych. Tak więc ich przestrzeń adresowa wynosiła 2^{16}=64 K. Współpracowały one z 8-bitową szyną danych, dlatego najbardziej naturalną pamięcią była taka, w której organizacja była 8-bitowa.

$$24564 = 2 \cdot 10^4 + 4 \cdot 10^3 + 5 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0.$$

Do takiego kodowania potrzebujemy dziesięciu symboli (cyfr) $0,1,2,3,4,5,6,7,8,9$ co nie jest praktyczne przy realizacji sprzętowej, tak więc używamy podstawy 2, co wymaga tylko dwóch symboli $0, 1$. Oznacza to, że np. ciąg bitów 10100110 **interpretujemy** jako 102, gdyż:

$$(01100110)_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 = 102.$$

Problem pojawi się gdy będziemy chcieli reprezentować liczby całkowite ze znakiem. Jak zapisać tylko przy pomocy 0 i 1 liczbę –102? Jednym z rozwiązań jest wprowadzenie tzw. bitu znaku: pierwszy bit (licząc od lewej strony), nie oznacza wagi, ale informuje o znaku: 1 oznacza – (minus), a 0 oznacza + (plus). Takie kodowanie nazywamy kodem uzupełnień do jeden (U1). Tak więc: ciąg 10011001 oznacza:

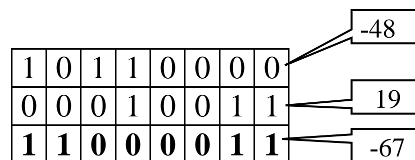
- w kodzie U1: -25
- w kodzie NKB: 153

Zauważmy, że w kodze U1 mamy dwie reprezentacje zera, czyli dwa różna ciągi należy interpretować jako jedno matematyczne 0:

- 10000000 = -0 = 0
- 00000000 = +0 = 0

Kod U1 mimo, że bardzo prosty i oczywisty nie jest jednak używany w praktyce. Wynika to przede wszystkim z tego, że bardzo niewygodnie realizuje się w nim operacje dodawania czy mnożenia. Nie wystarczy zwykła suma bitowa z przeniesieniem.

<u>Przykład 1</u>



1	0	1	1	0	0	0	0	-48
0	0	0	1	0	0	1	1	19
1	**1**	**0**	**0**	**0**	**0**	**1**	**1**	-67

Jak widać zwykłe dodawanie bitowe z przeniesieniem prowadzi do niepoprawnych wyników. Oznacza to, że do obsługi liczb całkowitych ze znakiem kodowanych w systemie U1 należałoby wykorzystać bardziej skomplikowane algorytmy (zrealizowane sprzętowo lub programowo).

Kodowanie U2

Zakładamy, że używamy zawsze porcji (ciągów) bitów o długości n (w praktyce komputerowej jest to 8, 16, 32, 64 czy 128).

Ciąg zer i jedynek o długości n interpretujemy w kodzie U2 jako następującą wartość:

$$(x_{n-1}x_{n-2}...x_1x_0)_{U2} = -x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + ... + x_1 \cdot 2^1 + x_0 \cdot 2^0.$$

Tak więc mamy

$$(x_{n-1}x_{n-2}...x_1x_0)_{U2} = \begin{cases} (x_{n-2}\dots x_1x_0)_2 & \text{gdy } x_{n-1}=0, \\ -2^{n-1}+(x_{n-2}\dots x_1x_0)_2 & \text{gdy } x_{n-1}=1. \end{cases} \qquad (0.1)$$

Przykład 2

Dysponujemy maszyną ośmiobitową. Naturalne jest więc używanie kodu U2 z $n=8$. Jakim liczbom całkowitym odpowiadają następujące sekwencje znajdujące się w pamięci?

1	1	0	1	1	0	0	1
0	0	0	0	1	1	1	0
0	1	1	0	0	0	1	0
1	0	1	1	1	0	0	1
1	1	0	1	1	0	1	0

Dla pierwszego ciągu mamy

$$(11011001)_{U2} = -2^7+1\cdot2^6+0\cdot2^5+1\cdot2^4+1\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0 =$$
$$= -128+64+16+8+1=-39.$$

a dla drugiego ciągu

$$(00001110)_{U2} = -0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+1\cdot2^3+1\cdot2^2+1\cdot2^1+0\cdot2^0 =$$
$$=8+4+2=14.$$

Jaka jest największa liczba dodatnia, którą można zapisać używając kodu U2 dla n=8 lub n=16?

Aby uzyskać liczbę dodatnią pierwszy (od lewej) bit musi być 0. Jeżeli pozostałe będą równe 1, to uzyskamy

$$(01111111)_{U2} = 2^6+2^5+2^4+2^3+2^2+2^1+2^0 = 127.$$
$$(0111111111111111)_{U2} =$$
$$= 2^{14}+2^{13}+2^{12}+2^{11}+2^{10}+2^8+2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0 = 32767$$

Ogólnie dla reprezentacji n bitowej największa liczba w kodzie U2 to

$$(011111\dots111)_{U2} = 2^{n-1}-1.$$

Jaka jest najbardziej ujemna liczba, którą można zapisać używając kodu U2 dla n=8 lub n=16?

Liczby ujemne w kodzie U2 mają 1 na pierwszej (od lewej) pozycji. Aby uzyskać liczbę ujemna o największej wartości bezwzględnej, widzimy ze wzoru (0.1), że pozostałe bity muszą być równe 0. Tak więc

$$(10000000)_{U2} = -2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+0\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0 = -128.$$

Analogicznie dla kodu 16-bitowego U2 mamy

$$(1000000000000000)_{U2} = -2^{15}+0\cdot2^{16}+\dots+0\cdot2^0 = -32768.$$

Ogólnie dla kodów o długości n w reprezentacji U2 „najbardziej" ujemna liczba to

$$(10000\ldots000)_{U2} = -2^{n-1}.$$

Ile różnych liczb całkowitych można zapisać używając kodu U1 i U2 dla n=8 lub n=16?

Z powyższych przykładów widać, że podzbiory liczb całkowitych, kodowane metodą U2 są następujące. Dla $n = 8$ to

$$\{-128,\ldots,-1,0,1,\ldots,127\},$$

a dla n=16 to

$$\{-32768,\ldots,-1,0,1,\ldots 32767\}.$$

Ogólnie dla reprezentacji n bitowej mamy zakres

$$\{-2^{n-1},\ldots,2^{n-1}-1\}.$$

Liczba zakodowanych wartości wynosi oczywiście 2^n.

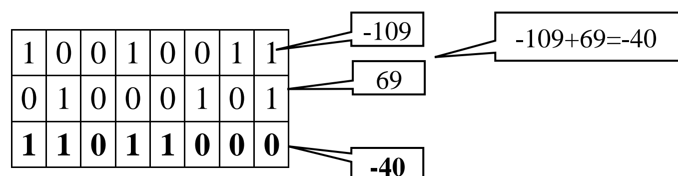
Zalety kodu U2:

- jednoznaczna reprezentacja liczby 0
- łatwa realizacja dodawania i odejmowania

Dodawanie dwóch liczb zapisanych w kodzie U2 można wykonać wykonując zwykłe dodawanie binarne z przeniesieniem. Tak długo jak nie ma przepełnienia wyniki wychodzą poprawne.

Przykład

Poniżej są pokazane liczby w pamięci 8-bitowej, które interpretujemy jak liczby całkowite zapisane w kodzie U2. Wykonajmy dodawanie (dwa pierwsze rzędy) metodą bitową z przenoszeniem, aby przekonać się, że wyniki są poprawne.



Przykład

Wykonać dodawanie następujących par liczb po zakodowaniu ich w kodzie U2:

a) x = -25, y = 80

b) x = -68, y = 92

c) x = -37, y = 37

Odejmowanie liczb jest oczywiście szczególnym przypadkiem dodawania. Warto jednak omówić osobno to działanie, aby zobaczyć jak prosto w kodzie U2 można uzyskać liczbę przeciwną do danej. Jeżeli mamy zakodowaną liczbę $x \in \{-2^{n-1},\ldots,2^{n-1}-1\} \subset \mathbb{Z}$

$$(x)_{U2} = (x_{n-1}x_{n-2},\ldots,x_0)_{U2}, \tag{0.2}$$

to liczbę przeciwną, $-x$, uzyskujemy przez negację bitową reprezentacji wyjściowej i dodanie bitowe jedynki. Tak więc reprezentację w kodzie U2 liczby ujemnej możemy uzyskać następującym wzorem

$$(-x)_{U2} = \overline{(|x|)_2} + 1,$$

gdzie kreska oznacza negację bitową, czyli $\overline{1} = 0$ oraz $\overline{0} = 1$.

Przykład

Zakładamy, że liczby są reprezentowane ośmiobitowo $(n = 8)$. Chcemy podać reprezentację U2 liczby -42.

Obliczamy reprezentację liczby $x = |-42| = 42$:

0	0	1	0	1	0	1	0

Dokonujemy zaprzeczenia bitowego powyższego kodu, zatem $\overline{(|x|)}$:

1	1	0	1	0	1	0	1

Dodajemy bitowo jedynkę, czyli obliczamy $\overline{(|x|)_2} + 1$:

1	1	0	1	0	1	1	0

Sprawdzamy:

$$(11010110)_{U2} = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -128 + 86 = -42.$$

Jeszcze jeden przykład, liczba -108.

$|-108| = 108$:

0	1	1	0	1	1	0	0

$\overline{(|-108|)}$:

1	0	0	1	0	0	1	1

$\overline{(|-108|)} + 1$:

1	0	0	1	0	1	0	0

Sprawdzamy:

$$(10010100)_{U2} = -2^7 + 2^4 + 2^2 = -128 + 16 + 4 = -108.$$

Zauważmy, że w kodzie U2 powielenie najstarszego (pierwszego od lewej) bitu nie zmienia wartości liczby. Na przykład powielanego bitu 1 mamy

$$(10110)_{U2} = -2^4 + 2^2 + 2^1 = -16 + 6 = -10,$$

$$(110110)_{U2} = -2^5 + 2^4 + 2^2 + 2^1 = -32 + 16 + 4 + 2 = -10$$

$$(1110110)_{U2} = -2^6 + 2^5 + 2^4 + 2^2 + 2^1 = -64 + 32 + 16 + 4 + 2 = -10$$

W przypadku, gdy powielamy bit 0 jest to oczywiste (w rozwinięciu w ogóle nic się nie zmienia).

Podsumujmy podstawowe własności kodu uzupełnień do dwóch.

- Najstarszy (pierwszy od lewej) bit informuje o znaku liczby. Bit 0 mają liczby nieujemne, bit 1 mają liczby ujemne. Np. $(1011)_{U2} = -2^3 + 2^1 + 2^0 = -5$.

- Zmiana znaku liczby zakodowanej w U2 dokonuje się przez negację poszczególnych bitów kodu i dodanie bitu 1 do najmłodszej (pierwszej od prawej) pozycji. Np. $(1011)_{U2} = -5.$ Zatem $\overline{(1011)_{U2}} + 1 = (0100)_{U2} + 1 = (0101)_{U2} = 2^2 + 2^0 = 5.$
- Powielanie najstarszego (pierwszego od lewej) bitu nie zmienia wartości zakodowanej liczby.
- Zakres liczb w kodzie U2 o długości n wynosi $[-2^{n-1},\ 2^{n-1}-1].$ Np. dla $n=8$ daje to zakres $[-128,\ 127],$ a dla $n=16$ zakres $[-32768,\ 32767].$

Reprezentacje liczb rzeczywistych

Matematyczny zbiór liczb rzeczywistych $\mathbb{R}$ jest nieskończony,[2] więc nie można reprezentować wszystkich liczb rzeczywistych przy pomocy skończonej liczby kombinacji, dostępnej dla kodowania w komputerach. Podobnie zresztą było dla liczb całkowitych $\mathbb{Z}$, ale w tamtym przypadku można przynajmniej kodować w skończony sposób „odcinki" zawarte w $\mathbb{Z}$. Natomiast w przypadku liczb rzeczywistych, nawet nie możemy zakodować wszystkich liczb rzeczywistych z przedziału np. $[0,\ 1].$ Tak naprawdę w komputerach kodujemy tylko liczby wymierne i to nie wszystkie, bo znów np. na odcinku $[0,\ 1]$ jest nadal nieskończenie wiele liczb wymiernych (przykładowo $\{1/2,\ 1/3,\ 1/4,\ldots\} \subset [0,\ 1] \cap \mathbb{Q}$). Tak więc znane z języków programowania *typy rzeczywiste* (np. w Pascalu **`Real`**, w języku C/C++ typ **`float`** czy **`double`**) są to w gruncie rzeczy skończone podzbiory zbioru liczb wymiernych $\mathbb{Q}$. Przypomnijmy, że liczby wymierne posiadają skończone rozwinięcie dziesiętne lub nieskończone, ale okresowe. Jeżeli więc chcemy prowadzić obliczenie na komputerze, które w teorii są obliczeniami na liczbach rzeczywistych, to aby te obliczenia komputerowe w sensowny sposób aproksymowały naszą abstrakcję, powinniśmy typy rzeczywiste tak konstruować, aby pokrywały one wystarczająco „gęsto" odpowiedni odcinek osi liczbowej. Temu właśnie służą reprezentacje zmiennoprzecinkowe (inaczej: zmiennopozycyjne, ang. *floating point numbers*).

Zanim omówimy nieco dokładniej pewien standard kodowania liczb rzeczywistych, zaczniemy od kilu przykładów wyjaśniających podstawowe idee. Zasadniczy pomysł przy reprezentacji liczb „z częścią ułamkową" opiera się na następującym podejściu. Jeżeli $x \in \mathbb{R}$, to możemy tę liczbę zapisać następująco:

$$x = \pm m \cdot 10^c, \tag{0.3}$$

przy czym zakładamy, że:

$$\frac{1}{10} \le m < 1. \tag{0.4}$$

[2] Okazuje się, że pojęcie *nieskończoności* w matematyce ma wiele obliczy. Na przykład zbiór liczb naturalnych $\mathbb{N}$ oraz zbiór liczb rzeczywistych $\mathbb{R}$ oba są nieskończone, ale są to różne rodzaje nieskończoności. W szczególności „liczba" elementów zbioru $\mathbb{R}$ jest „większa" od liczby elementów zbioru $\mathbb{N}$. Mówiąc opisowo, elementów zbioru $\mathbb{R}$ jest więcej niż w elementów zbioru $\mathbb{N}$. Oczywiście każdy niepusty odcinek $J \subset \mathbb{R}$ jest również nieskończony i można pokazać, że typ jego nieskończoności jest taki sam jak całego zbioru $\mathbb{R}$. Czyli – aby znów odwołać się do języka opisowego – liczba elementów odcinka $(a,\ b)$ jest taka sama, jak liczba elementów zbioru $\mathbb{R}$. Wydaje się to dziwne, gdyż odcinek jest istotnym podzbiorem zbioru $\mathbb{R}$, jednak jest z nim „równoliczny". Można tym pojęciom nadać precyzyjny sens i rozwinąć tzw. *teorię mocy*.

Liczbę m nazywamy *mantysą*, a liczbę c *cechą* liczby x (przy podstawie dziesiętnej). Okazuje się, że każdą liczbę można tak zapisać, i to w sposób jednoznaczny! Jeżeli jest spełniony warunek (0.4), to mówimy, że postać (0.3) jest znormalizowana.

Przykład

Zapiszmy w postaci znormalizowanej liczby: 12,23; 243,10; 0,00221, 44. Mamy oczywiście

$$12,23 = +0,1223 \cdot 10^2$$
$$-243,10 = -0,2431 \cdot 10^3$$
$$0,00221 = 0,221 \cdot 10^{-2}$$
$$44 = 0,44 \cdot 10^2.$$

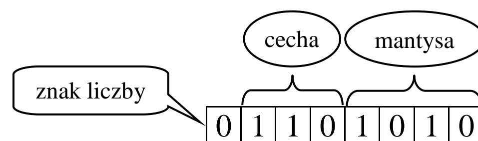
Podobnie jak dla liczb całkowitych, także w przypadku liczb rzeczywistych przechowywanych w komputerze, lepiej jest używać podstawy dwa zamiast dziesięć. W tym przypadku odpowiednimi wagami przypisanymi do poszczególnych pozycji cyfr (0 i 1) są teraz potęgi dwójki. Na przykład

$$101,1101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 5 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 5\frac{13}{16} = 5,8125.$$

Znormalizowana postać przy podstawie 2 (odpowiednik wzorów (0.3), (0.4)) zdefiniowana jest następująco

$$x = \pm m \cdot 2^c, \quad \frac{1}{2} \le m < 1.$$

Podstawą reprezentacji zmiennopozycyjnej jest wyrażenie (0.3) (wraz z warunkiem (0.4)) oraz reguły dotyczące kodowania znaku, cechy i mantysy. Na przykład możemy się umówić, że przy 8-bitowej pamięci przeznaczymy 1 bit na znak, 3 bity na cechę, 4 bity na mantysę przy czym pierwszy bit od lewej strony określa znak liczby, a znak cechy uzyskujemy przez kodowanie jej w kodzie U2.[3] Przykładowo wartość reprezentowana przez następującą sekwencję bitów, przy założeniu, że używamy wspomnianej konwencji



Mamy więc (pierwszy bit to znak) tak: $m = (1010)$, $c = (110)$, tak więc:

$$m = 1 \cdot \frac{1}{2^1} + 0 \cdot \frac{1}{2^2} + 1 \cdot \frac{1}{2^3} + 0 \cdot \frac{1}{2^4} = +\frac{5}{8}$$
$$c = -2^{3-1} + 1 \cdot 2^1 + 1 \cdot 2^0 = -4 + 2 = -2.$$

Daje to liczbę

[3] W realnych systemach kodowania (np. w opisanym dalej standardzie IEEE 754) nie koduje się znaku cechy dodatkowym bitem znaku. Znak cechy kodowany jest przez specjalną konwencję, polegającą na odejmowaniu tzw. obciążenia (ang. *bias*). Szczegóły są dalej.

$$x = +\frac{5}{8} \cdot 2^{-2} = \frac{13}{32} = 0,203125.$$

Przykład

Jaka jest największa i najmniejsza liczba dodatnia, które można reprezentować w podanym powyżej, przykładowym kodowaniu zmiennopozycyjnym 8-bitowym.

Największa: $(00111111)_{zmp} = +(2^2 + 2^1)(2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}) = \frac{3}{8}(8+4+2+1) = \frac{45}{8} = 5,625.$

Najmniejsza dodatnia: $(01000001)_{zmp} = +(2^{-3-1} + 0 + 0)(0 + 0 + 0 + 2^{-4}) = 2^{-6} = 0,015625.$

Łatwo jest podać przykład liczby, której nie można reprezentować. Wyliczyć tę liczbę.

Standard zmiennoprzecinkowy IEEE 754

Norma IEEE 754 jest standardem opisującym precyzyjnie sposób kodowania i wykonywania operacji na pewnym podzbiorze liczb wymiernych. Dzięki niemu możemy wykonywać obliczenia na liczbach, które posiadają część ułamkową.

Reprezentacja liczb w standardzie IEEE 754

W standardzie IEEE 754 określono trzy formaty stałoprzecinkowe dwójkowe (binarne), jeden stałoprzecinkowy format dziesiętny BCD, oraz cztery formaty zmiennoprzecinkowe. We wszystkich formatach zmiennoprzecinkowych wykładnik jest reprezentowany w kodzie z obciążeniem (ang. *bias B*). Jeżeli liczba bitów wykładnika (cechy) wynosi *e*, to obciążenie (zwane też przemieszczeniem) wynosi

$$B = 2^{e-1} - 1.$$

Na przykład dla $e = 8$ mamy $B = 2^{8-1} - 1 = 2^7 - 1 = 127.$

Dalej opiszemy dokładniej format zmiennopozycyjny zwykły o pojedynczej precyzji. Ciąg bitów dzielimy na trzy części: znak, cecha i mantysa jak poniżej

znak	cecha	mantysa

W formacie tym:

- całkowita liczba wszystkich bitów wynosi 32,
- liczba bitów cechy wynosi $e = 8$,
- liczba bitów mantysy wynosi $f = 23$.

Wartość liczby znormalizowanej jest obliczana w regularnych przypadkach wg wzoru

$$v = (-1)^z \cdot 2^{cecha-B} \cdot 1,mantysa. \qquad (0.5)$$

Użyte wyżej określenie „regularne przypadku” sugeruje, że nie zawsze wartość przypisywane do kodu w standardzie IEEE 754 jest obliczana wzorem (0.5). W niektórych przypadkach przypisywana wartość jest określona bezpośredni. Na przykład ciągom (1000…00) lub (0000…00) przypisujemy

wartość zero, „dodatnie”, +0 oraz „ujemne”, -0 (oczywiście matematycznie są to te same wartości, tylko inaczej reprezentowane).

Ponieważ w formacie pojedynczej precyzji obciążenie $B=127,$ więc możemy ten wzór zapisać jako

$$v=(-1)^{z}\cdot 2^{cecha-127}\cdot 1,mantysa. \tag{0.6}$$

Zauważmy, że kodowanie znaku liczby jest proste: 0=„+”, 1=„-”. Jeżeli chodzi o cechę, to znak jest w tym przypadku kodowany niejawnie, poprzez umowę, że od wartości cechy odejmujemy obciążenie, które dla formatu pojedynczej precyzji wynosi $B=127.$ Tak więc zakres wartości cechy wynosi

$$\{-127,-126,\ldots,128,129\}.$$

Przykład

Dany jest kod liczby w formacie pojedynczej precyzji

0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Oblicz wartość liczbową tego kodu.

Rozwiązanie

$$cecha=01111100=2^{2}+2^{3}+2^{4}+2^{5}+2^{6}=124$$

$$mantysa=01100...0=2^{-2}+2^{-3}$$

$$1,mantysa=1+2^{-2}+2^{-3}=\frac{11}{8}$$

Zatem $v=(-1)^{0}\cdot 2^{124-127}\cdot\frac{11}{8}=2^{-3}\cdot\frac{11}{8}=\frac{11}{64}=0,171875.$

Odp. Wartością liczbową podanego kodu jest liczba, która w zapisie dziesiętnym wynosi 0,171875.

Pewna komplikacja z poprawnym odczytywaniem wartości przypisanych do kodów jest to, że norma definiuje dwie kategorie liczb – *liczby znormalizowane* i *liczby zdenormalizowane* – i dwa sposoby obliczania ich wartości.

Liczby znormalizowane mają cechę niezerową, a dokładniej od 1 do 2^{e}-2, czyli dla standardu pojedynczej precyzji od 1 do 2^{8}-2=126. Wartość liczby znormalizowanej obliczana jest wg wzoru (0.6). Natomiast ***liczby zdenormalizowane*** mają cechę równą zero (ale mantysę niezerową) i wartość obliczana jest wg poniższego wzoru

$$v=(-1)^{z}\cdot 2^{-126}\cdot 0,m. \tag{0.7}$$

Na przykład poniższa liczba jest zdenormalizowana

0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

której wartość wynosi

$$v=(-1)^{0}\cdot 2^{-126}\cdot 0,01100\ldots 0=2^{-126}\left(\frac{1}{2^{2}}+\frac{1}{2^{3}}\right)\approx 1,75510^{-38}\cdot 0,375\approx 4,40810^{-39}. \tag{0.8}$$

Jaka jest najmniejsza dodatnia liczba, którą możemy reprezentować w opisywanym standardzie pojedynczej precyzji? Inaczej mówiąc, jak blisko możemy podejść do zera (od prawej strony)? Jeżeli spojrzymy na wyrażenie (0.6) i (0.7), to łatwo się przekonać, że powinna to być liczba zdenormalizowana. (Mogłoby się wydawać, że powinniśmy wybrać takie bity, aby $mantysa = 0$ oraz $cecha = 0$. Ale taki kod składający się z samych zer (ewentualnie z jedynką na początku), nie interpretujemy ani wg wyrażenia (0.6) ani wg (0.7), tylko przypisujemy mu wartość 0.) Dlatego musimy wziąć $c = (00000000)$ oraz $m = (000\ldots01)$, czyli

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

któremu odpowiada wartość (wg (0.7), bo jest to liczba zdenormalizowana)

$$v_{min,den} = 2^{-126} \cdot (0 + 2^{-23}) = 2^{-149} = \left(10^{\log 2}\right)^{-149} \approx 10^{-0,30103 \cdot 149} \approx$$
$$\approx 10^{-44,8534} \approx 1,401 \cdot 10^{-45}.$$

Z drugiej strony możemy zapytać jaka jest największa liczba zdefiniowana w kodzie pojedynczej precyzji? Ponownie patrząc na wzór (0.6) spodziewamy się, że należy brać same jedynki dla cechy i mantysy. Ale ponieważ w normie IEEE 754 zarezerwowano specjalne znaczenie dla kodu, którym cecha składa się z samych jedynek, to tak naprawdę największa liczba ma $c = (11111110)$. Daje to kod

0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

któremu odpowiada wartość

$$v = (-1)^0 \cdot 2^{254-127} \cdot (1 + 2^{-1} + 2^{-2} + \ldots + 2^{-23}) = 2^{127} \frac{1-(1/2)^{-24}}{1-(1/2)} \approx 2^{128} =$$
$$= \left(10^{\log 2}\right)^{128} \approx 10^{0,30103 \cdot 128} \approx 10^{38,532} \approx 3,403 \cdot 10^{38}.$$

Jaka jest odległość pomiędzy największą liczbą a drugą w kolejności? Wiemy już z powyższego rachunku, że

$$v_{max} = 2^{127} \frac{1-(1/2)^{-24}}{1-(1/2)} = 2^{128} - 2^{104}.$$

Najbliższa od lewej strony ma oczywiście kod

0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

któremu odpowiada wartość

$$v = (-1)^0 \cdot 2^{254-127} \cdot (1 + 2^{-1} + 2^{-2} + \ldots + 2^{-22}) = 2^{127} \frac{1-(1/2)^{-23}}{1-(1/2)} = 2^{128} - 2^{105}.$$

Zatem

$$v_{max} - v = 2^{128} - 2^{104} - (2^{128} - 2^{105}) = 2^{105} - 2^{104} = 2^{104} \approx 2,03 \cdot 10^{31}.$$

Jaka jest odległość pomiędzy dwoma liczbami pojedynczej precyzji najbliższymi zeru od prawej strony? Jeżeli weźmiemy pod uwagę liczby zdenormalizowane, to nieco wyżej wyliczyliśmy, że $0 < v_{min,den} = 2^{-126} \cdot 2^{-23} \approx 1,4 \cdot 10^{-45}$. Kolejne liczba (zdenormalizowana) będzie miała kod

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

któremu odpowiada wartość (wg wzoru (0.7))

$$v = (-1)^0 \cdot 2^{-126} \cdot (0 + 2^{-22}) = 2^{-128} \approx 2,938 \cdot 10^{-39}.$$

Zatem różnica pomiędzy dwoma dodatnimi najbliższymi zeru liczbami (zdenormalizowanymi) wynosi

$$v - v_{min,den} = 2^{-128} - 2^{-129} = 2^{-129} = 1,469 \cdot 10^{-39}.$$

Dla dwóch najmniejszych dodatnich liczb znormalizowanych różnica ta jest rzędu 10^{-38}. Jak widać z powyższych rozważań liczby reprezentowalne zmiennopozycyjne nie są rozmieszczone równomiernie na osi liczbowej. Dla pojedynczej precyzji: w pobliżu zera są one rozmieszczone z gęstością rzędu 10^{-38}, ale w pobliżu końca zakresu z gęstością rzędu 10^{30}.

Z przykładów tych widać jasno, że używając standardowych typów danych (float, double itd.) nie możemy na ogół wykonywać obliczeń z dowolna dokładnością. Należy zatem pamiętać, że algorytmy działające na abstrakcyjnych liczbach z □ (czy □) będą na ogół dawać inne wyniki niż, gdy działają na dziedzinie licz reprezentowalnych. Przykładowo ten sam algorytm

<table>
<tr><th>Dziedzina: $x \in$ □</th><th>Dziedzina: $x \in$ float</th></tr>
<tr><td><pre>x=1.0;
while (x>0) {
    cout << x << "\n";
    x = x/2.0;
}</pre></td><td><pre>x=1.0;
while (x>0) {
    cout << x << "\n";
    x = x/2.0;
}</pre></td></tr>
</table>

da odmienne rezultaty, gdy będzie „pracował” w dziedzinie □ oraz w dziedzinie liczb reprezentowalnych **`float`**. W tym pierwszym przypadku pętla jest nieskończona, ale w tym drugim oczywiście nie: w pewnym momencie „zejdziemy” poniżej najmniejszej wartości dodatniej (pamiętamy, dla formatu pojedynczej precyzji najmniejsza liczba dodatnia (zdenormalizowana) to ok. $1,401 \cdot 10^{-45}$, a zatem x przyjmie wartość zero w skończonej liczbie obiegów powyższej pętli **`while`**.)

Poniżej jest tabelka opisująca podstawowe obiekty standardu IEEE 754 (liczby zwykłe (znormalizowane), liczby nie znormalizowane (zdenormalizowane), zera maszynowe, nieskończoności maszynowe oraz kody, które nie odpowiadają żadnym liczbom, tzw. NaNs (od ang. *Not a Number* – nie liczba).

Typ	**Cecha**	**Mantysa**
Zera (±0)	0	0
Liczba Znormalizowana	1,...,254	Dowolna
Liczba Zdenormalizowana	0	≠0
±∞	255	0
NaNs (Not Numbers)	255	≠0

Własności arytmetyki zmiennopozycyjnej

Dotychczas opisywaliśmy tylko sposób reprezentowania liczb (całkowitych lub wymiernych) w komputerze. Oczywiście liczby te są przydatne tylko pod warunkiem, ze mamy zdefiniowane na nich operacje arytmetyczne (+, ·, −, /). Jak już wiemy zbiór liczb reprezentowalnych $F \subset \mathbb{R}$ jest skończony i nierównomiernie rozmieszczony na osi liczbowej. Wszelkie obliczenie, które wykonujemy w programach komputerowych, gdy używamy standardowych typów liczbowych są tak naprawdę wykonywane w zbiorze F a nie w zbirze $\mathbb{Q}$ czy $\mathbb{R}$. Tak więc już na etapie wprowadzania danych mamy prawie zawsze do czynienia z przybliżeniami. Jest to źródło tzw. *błędów obcięcia* (czasami zwanych *błędami reprezentacji* czy *błędami zaokrąglenia*). Jeżeli na przykład w programie mamy fragment

```
...
float x;
cout << "Podaj wartość x: ";
cin >> x;
...
```

i użytkownik wpisze wartość 0.3, to oczywiście nastąpi przypisanie x = 0.3, ale w pamięci zarezerwowanej dla zmiennej x pojawi się reprezentacja zmiennopozycyjna, której wartość nie będzie dokładnie równa matematycznej liczbie 0.3, gdyż ta liczba nie jest reprezentowalna w formacie IEEE 754 – nastąpi właśnie błąd obcięcia. Co więcej, jeżeli teraz będziemy wykonywali operacje arytmetyczne na takich liczbach, np. $x+y$, to na ogół wartość nie będzie równa matematycznej sumie, nawet gdy obie liczby były reprezentowane dokładnie. Inaczej, jeżeli $x, y \in F$, to na ogół mamy $x+y \notin F$, co oznacza, że komputer nie obliczy matematycznej sumy $x+y$ tylko pewne przybliżenie. Dlatego w rozważaniach nad arytmetyką zmiennopozycyjną często używa się innych symboli, np. dla sumy $x \oplus y$ czy $x \otimes y$. Wygodnie też jest wprowadzić funkcję, która opisuje zaokrąglenie, $fl: \mathbb{R} \to F$, określoną tak: dla $x \in \mathbb{R}$, to $fl(x)$ oznacza najbliższą jej liczbę ze zbioru F. Oczywiście dla $x \in F$ mamy $fl(x) = x$. Ściśle rzecz biorąc określenie $fl(x) \in F$ podane przed chwila dotyczy przypadku, gdy liczba nie jest zbyt duża i nie jest zbyt bliska zeru: dokładniej obowiązuje ona, gdy

$$x \in [-x_{max}, -x_{min}] \cup [x_{min}, x_{max}].$$

W pozostałych przypadkach, tzn. dla $x < -x_{max}$ lub $x > x_{max}$ lub $x \in (-x_{min}, x_{min}) \setminus \{0\}$ mówimy o nadmiarze (ang. *overflow*) lub niedomiarze (ang. *underflow*). Sytuacje te na ogół powodują wygenerowanie błędu (szczegóły zależy od implementacji).

Podstawowa własność arytmetyki zmiennopozycyjnej jest następująca:

jeżeli $x \in \mathbb{R}$ *jest taka, że* $x_{min} \le |x| \le x_{max}$, *to*

$$fl(x) = x(1+\delta), \quad \text{przy czym } |\delta| \le 2^{-t}, \tag{0.9}$$

gdzie t oznacza liczbę bitów mantysy.

Wielkość $u = 2^{-t}$ jest nazywana precyzją arytmetyki (ang. *roundoff unit*) lub precyzja maszynową (ang. *machine precision*). Na przykład dla standardu pojedynczej precyzji (normy IEEE

754) mamy $t = 23$, zatem $u = 2^{-23} \approx 1{,}19 \cdot 10^{-7}$, a dla podwójnej precyzji $t = 52$, zatem $u = 2^{-52} \approx 2{,}22 \cdot 10^{-16}$.

Jak już wcześniej stwierdzono, konieczne jest zdefiniowanie na zbiorze liczb maszynowych F arytmetyki, która będzie analogiczna – na ile jest to możliwe – do arytmetyki w zbiorze $\mathbb{R}$. Na przykład dla dodawania $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ odpowiednik zmiennopozycyjny oznaczymy przez $\oplus$, tzn. $\oplus : \mathbb{R} \times \mathbb{R} \to F$. Jego definicja jest następująca

$$x \oplus y := fl(fl(x) + fl(y)). \tag{0.10}$$

Ze względu na właściwości liczb maszynowych – (0.9) – oczekujemy, że operator maszynowy $\oplus$ będzie miał podobną własność: $\forall x, y \in F$, $\exists \delta \in \mathbb{R}$ taka, że

$$x \oplus y = (x + y)(1 + \delta) \quad \text{przy czym } |\delta| \le 2^{-t}. \tag{0.11}$$

Własność ta (również dla pozostałych operatorów $-$, $\cdot$, $/$) jest spełniona pod warunkiem, że procedury realizujące definicję (0.10) są odpowiednio zdefiniowane (szczegóły pomijamy, gdyż wykracza to poza zakres tego skryptu).

Na koniec zauważmy, że wiele własności arytmetyki liczb rzeczywistych nie zachodzi w arytmetyce zmiennopozycyjnej. Na przykład prawo łączności nie jest spełnione, gdyż na ogół mamy

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z.$$

Z drugiej strony pewne prawa są zachowane, np. przemienność dodawania i mnożenia – co jest prostą konsekwencją definicji (0.10) oraz przemienności działań w $\mathbb{R}$. Mamy bowiem

$$x \oplus y = fl(fl(x) + fl(y)) = fl(fl(y) + fl(x)) = y \oplus x.$$

Zatem

$$x \oplus y = y \oplus x,$$
$$x \odot y = y \odot x.$$