Tietokoneen binääriaritmetiikasta

.0001 1001 1001 1001 1001 1001 …

Dokumentissa kuvataan lukujärjestelmiä ja niiden välisiä muunnoksia erityisesti binääriaritmetiikan osalta. Lisäksi tarkastellaan tietokoneen kokonaisluku- ja liukulukuesityksiä sekä niihin liittyviä käyttäjästä riippumattomia pyöristysvirheitä havainnollisten esimerkkien avulla

Kari Peisa
RAMK 2012

Lukujärjestelmät ja lukujen paikkajärjestelmät

Kokonaislukujen k-kantaiset esitykset

10-järjestelmän (eli kantaluku **on 10) paikkajärjestelmä:**

Käytettävissä numerot **0, 1, 2, 3, 4, 5, 6, 7, 8** ja **9**

. . .	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
0	0	0	0	0	0	**3**	**0**	**9**

Kaikki kokonaisluvut voidaan esittää summana

$z_{10} = \sum_{k=0}^{n} a_k \cdot 10^k$, missä $a_k \in \{0,1,2,3,4,5,6,7,8,9\}$ ja $n \in \boldsymbol{N_0}$

Esimerkiksi **$309_{10} = 3*10^2 + 0*10^1 + 9*10^0$**

2-järjestelmä

Käytettävissä numerot **0** ja **1**

. . .	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	**1**	**0**	**1**

Kaikki 2-järjestelmän kokonaisluvut voidaan esittää 10-järjestelmässä summana

$z_{10} = \sum_{k=0}^{n} a_k \cdot 2^k$, missä $a_k \in \{0,1\}$ ja $n \in \boldsymbol{N_0}$

Esimerkiksi **$101_2 = 1*2^2 + 0*2^1 + 1*2^0 = 5_{10}$.**

8-järjestelmä eli oktaalijärjestelmä

Käytössä numerot **0**, **1**, **2**, **3**, **4**, **5**, **6** ja **7**

. . .	8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0
0	0	0	0	**7**	**0**	**5**	**0**	**1**

Kaikki 8-järjestelmän kokonaisluvut voidaan esittää 10-järjestelmässä summana

$z_{10} = \sum_{k=0}^{n} a_k \cdot 8^k$, missä $a_k \in \{0,1,2,3,4,5,6,7\}$ ja $n \in \boldsymbol{N_0}$

Esimerkiksi **$70501_8 = 7*8^4+5*8^2+1*8^0 = 28993_{10}$**.

16-järjestelmä eli heksadesimaalijärjestelmä

Käytössä numerot **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **A**, **B**, **C**, **D**, **E** ja **F**

. . .	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	**A**	**0**	**F**	**9**

Kaikki 16-järjestelmän kokonaisluvut voidaan esittää 10-järjestelmässä summana

$z_{10} = \sum_{k=0}^{n} a_k \cdot 16^k$, missä $a_k \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ ja $n \in \boldsymbol{N_0}$

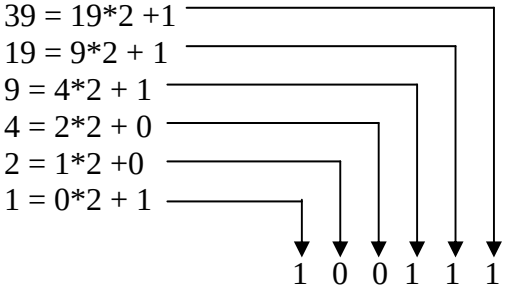
Esimerkiksi **$A0F9_{16} = 10*16^3+15*16^1+9*16^0 = 41209_{10}$**.

Kokonaisluvun muunnos 10-järjestelmästä k-järjestelmään

Edellä on kuvattu lukujen paikkajärjestelmään perustuvat muunnokset eri lukujärjestelmistä 10-järjestelmään. Vastaavasti 10-järjestelmän kokonaisluvun muuntaminen toiseen lukujärjestelmään saadaan peräkkäisten jakolaskujen avulla. Luku jaetaan ensin kantaluvulla k ja laitetaan jakojäännös k-järjestelmän luvun ”ykkösten” paikalle eli potenssin k^0 kertoimeksi. Sen jälkeen jaetaan edellisen jakolaskun osamäärän kokonaisosa luvulla k ja jakojäännös tulee k-järjestelmän luvun potenssin k^1 kertoimeksi jne…

Esimerkkinä muunnos 2-järjestelmään:

(”2 menee 19 kertaa 39:ään ja jakojäännös = 1”)



Muunnos 2-järjestelmästä 8- tai 16-järjestelmään

Muunnos saadaan hyvin yksinkertaisesti ryhmittelemällä binääriesitys 3 tai 4 numeron ryhmiin

16 / 2 / 8																
	4_{16}			**D_{16}**				**1_{16}**				**A_{16}**				
	1	0	0	1	1	0	1	0	0	0	1	1	0	1	0	
	4_8			**6_8**				**4_8**				**3_8**			**2_8**	

$100110100011010_2 = 46432_8 = 4D1A_{16}$

Tietokoneen kokonaislukuaritmetiikka

Kokonaislukuesitykselle varataan tietty lukumäärä vierekkäisiä muistipaikkoja, jotka vastaavat 2-järjestelmäluvun paikkoja. Ne voivat sisältää joko 0 tai 1.

Etumerkiksi C-kielen pienin kokonaislukutyyppi signed char (8 bittiä)
Sisäisessä esitysmuodossa $E \in \{0,1\}$ on etumerkkiä varten ja $b \in \{0,1\}$

E	b	b	b	b	b	b	b

suurin luku$_{10}$ = **$+111\ 1111_2 = +2^6+2^5+2^4+2^3+2^2+2^1+2^0 = 127$**

Negatiiviset luvut saadaan vastaavista positiivisista **kahden komplementtimuodossa** seuraavasti:

1. vaihdetaan luvussa esiintyvät 1:t 0:ksi ja päinvastoin.
2. Lisätään saatuun lukuun 1

2-järjestelmässä voidaan käyttää samoja peruslaskutoimitusten laskualgoritmeja kuin 10-järjestelmässä

Yhteenlasku

Laske 11001 + 1011 $=(25)_{10} + (11)_{10} = (36)_{10}$

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

Vähennyslasku tehdään lisäämällä vastaava negatiivinen luku

Esimerkiksi $12_{10} – 5_{10} = 12_{10} + (-5_{10})$

5_{10} = 0000 0101, -5_{10} = 1111 1010 + 0000 0001 = 1111 1011

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

Tuloksena 111 = 7_{10}.

Mitä tapahtuu, kun lasketaan $127_{10} + 2_{10}$?

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

Tuloksena on negatiivinen luku, koska etumerkki on 1. Arvoalueen ylivuoto!!

Vastaava 10-järjestelmän arvo saadaan kahden komplementtina

1000 0001 → 0111 1110 + 0000 0001 = 0111 1111 = -127_{10}

Asiaa voi tutkia esimerkiksi oheisella c-kielisellä ohjelmalla

```
#include <stdio.h>
#include <limits.h>
void main(void)
{   int a=INT_MAX;
    int b = INT_MIN;
    printf("\n INT_MAX = %i INT_MIN = %i",a, b);
    printf("\n INT_MAX +1 = %i, INT_MAX +2 = %i",a+1, a+2 );
    printf("\n INT_MIN -1 = %i, INT_MIN -2 = %i",b-1, b-2);
    fflush(stdin);
    printf("\n paina jotain...");
    getchar();
}
```

Tietokoneen liukulukuaritmetiikka

Reaalilukujen tieteellinen esitysmuoto

Tavanomainen desimaalilukujen (reaalilukujen) tieteellinen esitysmuoto on

$$x = \pm r\,E \pm n \quad \left(x = \pm r \cdot 10^{\pm n}\right)$$

missä **mantissa *r*** on välillä $10^{-1} \le r < 1$ $(0.1_{10} \le q < 1_{10})$ ja **eksponentti** $n \in \boldsymbol{N_0}$

Esimerkiksi

$$412.537 = 0.412537E + 3 \qquad \left(0.412537 \cdot 10^{3}\right)$$
$$0.001002 = 0.1001E - 2 \qquad \left(0.1002 \cdot 10^{-2}\right)$$

2-järjestelmien desimaalilukujen tieteelliset esitysmuodot noudattavat samaa periaatetta

$$x = \pm q\,E \pm m \quad \left(= \pm q \cdot 2^{\pm m}\right)$$

missä **mantissa *q*** on välillä $2^{-1} \le q < 1$ $(0.1_2 \le q < 1_2)$ ja **eksponentti *m*** on kokonaisluku.

Esimerkiksi

$$1101.101 = 0.1101101E + 4.$$
$$0.001011 = 0.1011E - 2.$$

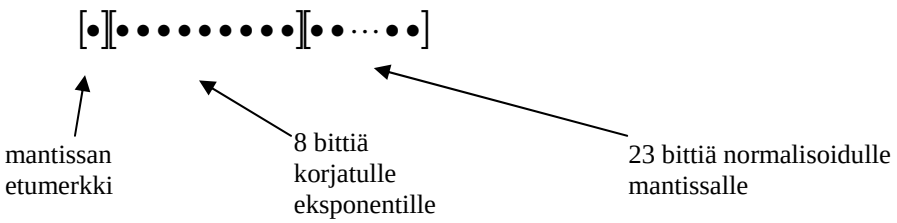
Kaikki tieteellisen esitysmuodon reaaliluvut saadaan laskettua 10-järjestelmän lukuna kantajärjestelmästä riippumatta samalla periaatteella, esimerkiksi

$$12.5_8 = 1 \cdot 8^1 + 2 \cdot 8^0 + 5 \cdot 8^{-1} = \frac{81}{8} = 10.125_{10}$$

$$1101.101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3} = \frac{109}{8} = 13.625_{10}.$$

Liukulukujen binääriesitykset

Tietokoneen sisäisesti käyttämät reaalilukuesitykset eli liukuluvut perustuvat 2-järjestelmän tieteelliseen esitysmuotoon. Reaaliluvun 32-bittinen yksinkertaisen tarkkuuden liukulukuesitys muistissa koostuu 32 bitistä, jotka on jaettu kolmeen kenttään seuraavasti:



Mantissan pituus vaikuttaa luvun esitystarkkuuteen, eksponentti määrää luvun suuruusluokan. Normalisoitu mantissa tarkoittaa sitä, että ensimmäinen bitti asetetaan ykköseksi, jolloin eksponentti korjataan vastaamaan oikeaa suuruusluokkaa. Itse asiassa ensimmäistä mantissan bittiä (1) ei tarvitse edes merkitä muistiin, jolloin käytännössä saadaan yksi ylimääräinen bitti (24.) mantissaan eli luvun tarkuuteen.

Eksponentin arvoa ei esitetä kuitenkaan (tehokkuussyistä) normaalina 7-bittisenä negatiivisena kokonaislukuna, vaan 6-bittisenä mukautettuna lukuna, jossa on suoritettu eräänlainen siirto alueelta [-126, +127] alueelle [1, 254], jolloin negatiivisten eksponettiarvojen vertailu on tehokkaampaa. Kun luvun arvo tulkitaan muistissa olevien bittien perusteella, tämä korjaus tulee huomioida.

Liukulukujen arvo voidaan laskea muistissa olevien bittien perusteella seuraavasti

$$(-1)^{[b]} \cdot 2^{[eeeeeeee]-127} \cdot 1.[mm\ldots m],$$

missä **[*b*]** on etumerkkibitti (0→ + ja 1→ -), **[*eeeeeeee*]** on binääriluku eksponenttikentässä ja **[*mm…m*]** on binääriluku mantissan kentässä.
Käytännössä yksinkertaisen liukuluvun (**float**) tapauksessa saadaan eräänlaiset äärirajat lukujen suuruuksille

itseisarvoltaan pienin luku $\approx 1.2 \cdot 10^{-38}$ sekä
itseisarvoltaan suurin luku $\approx 3.4 \cdot 10^{38}$.

Kun käytetään kaksinkertaisen tarkkuuden liukulukuja (**double**), lukujen tarkkuus (mantissa 52 bittiä) sekä myös suuruusluokka (eksponentti 11 bittiä) kasvavat. Tällöin kuitenkin tehokkuus kärsii, sillä ohjelma kuluttaa sekä muistia että laskenta-aikaa enemmän.
Jos eksponettin arvo on yli sallitun rajan, sanotaan, että tapahtuu *ylivuoto*. Mikäli arvo alittaa sallitun minimiarvon (negatiivinen arvo), sanotaan, että tapahtuu *alivuoto*.

Merkitsevien numeroiden menetys liukulukujen aritmetiikassa

Kun tietokoneen kanssa työskennellään, me syötämme numeerista tietoa 10-järjestelmän lukuina. Tavallisesti luemme tulosteina myös 10-järjestelmän mukaisia esityksiä. Sisäisesti laskennassa tietokone käsittelee kuitenkin vain 2-järjestelmän lukuja. Tietokoneen käyttäjän ei yleensä tarvitse huomioida tätä asiaa mitenkään, mutta järjestelmien muunnoksissa voi joskus joutua tekemisiin erilaisten **pyöristysvirheiden** kanssa.

Pyöristysvirheet johtuvat siitä, että tietokone voi säilyttää vain **kiinteän suuruisen lukumäärän** luvussa esiintyvistä numeroista (biteistä). Lisäksi monissa hyvin tavanomaistenkin desimaalilukujen muunnoksissa kuten 10-järjestelmän luvulla 0.1_{10} on **jaksollinen ääretön binäärilukuesitys**

$$0.1_{10} = (0.0001100110011001\cdots)_2 .$$

Muunnos voidaan tehdä samantapaisella algoritmilla kuin kokonaislukujen tapauksessa, nyt jakolaskun sijasta käytetään kertolaskua. Reaaliluvun desimaaliosa kerrotaan kohdejärjestelmän kantaluvulla k (tässä 2) ja merkitään näin saadun luvun kokonaisosa k-järjestelmän ensimmäiseksi desimaaliksi eli potenssin k^{-1} kertoimeksi. Algoritmia jatketaan siten, että saadun luvun desimaaliosaa kerrotaan edelleen kantaluvulla k ja otetaan syntyneen luvun kokonaisosa seuraavaksi desimaaliksi jne.

0.1*2 = 0.2 → 0
0.2*2 = 0.4 → 0
0.4*2 = 0.8 → 0
0.8*2 = 1.6 → 1
0.6*2 = 1.2 → 1
0.2*2 = 0.4 → 0

$0.4*2 = 0.8 \rightarrow 0$

jne…

Tässä voimme todeta, että algoritmi ei pääty koskaan, vaan saamme yo jaksollisen äärettömän binääridesimaalikehitelmän.
Tietokone joutuu katkaisemaan esityksen sovitun kiinteän bittimäärän kohdalta, jolloin muodostunut binääriluku onkin enää vain alkuperäisen likiarvo. Tällainen pyöristysvirhe voi syntyä molempiin suuntiin tehdyissä muunnoksissa.
Esimerkiksi luvun 0.1_{10} katkaistun 32-bittisen binääriluvun todellinen suuruus on

$$0.10000\ 00014\ 90116\ 11938\ 47656\ 25_{10}\ .$$

Yleensä käyttäjä ei huomaa virhettä, koska luvun esitystarkkuus on sellainen, että virhe ei tule esille.

Liukulukujen yhteen- ja vähennyslaskuissa tapahtuu myös määrätyissä tapauksissa merkitsevien numeroiden menetyksiä. Oletetaan, että ***x*** ja ***y*** ovat liukulukuja siten, että $x >> y$. Yhteenlaskussa $x + y$ luvun ***y*** **nollasta eroavien bittien paikkaa on siirrettävä binääriesityksen mantissassa oikealle**, jotta yhteenlasku luvun ***x*** vastaavien bittien kanssa voidaan suorittaa. Tämä merkitsee joidenkin tai mahdollisesti myös kaikkien luvun ***y*** merkitsevien bittien menetystä summassa, jos ***x*** ja ***y*** eroavat riittävän paljon.

Oletetaan, että laskentasysteemissä on 7 numeron tarkkuus 10-järjestelmän desimaaliluvuille. Tällöin summassa

$$.1210121_{10} + .4257318_{10}\ \ 10^{-6} = .\ 1210125257318_{10},$$

tulos joudutaan katkaisemaan ja esittämään muodossa $.1210125_{10}$. Tällöin tapahtuu 6 merkitsevän numeron katoaminen.

Samoin, jos ***x*** ja ***y*** ovat liukulukuja siten, että $x \approx y$, muodostuu lukujen erotuksessa $x - y$ tai $y - x$ mantissan alkuun pitkä jono nollia. Kun tietokone normalisoi erotuksen, **siirtyvät mantissassa oikealla olevat nollasta eroavat bitit vasemmalle ja samalla sen loppuun lisätään nollia**. Näillä nollilla ei kuitenkaan ole mitään todellista vastaavuutta oikean lopputuloksen suhteen, vaan ne edustavat merkitsevien numeroiden menetystä.

Oletetaan seuraavassa tarkkuus on 10 numeroa 10-järjestelmän reaaliluvuille *x* and *y* siten, että $x = .12103411225634_{10}$ ja $y = .\ 1210341115412_{10}$

$$x - y = .1210341122\,5634_{10}\ \text{--}\ .\ 1210341111\,5412_{10} = .0000000011\,0222_{10}.$$

Tulos liu'utetaan tietokoneessa vastaamaan muotoa

$$.11\,00000000_{10},$$

Viimeiset 8 nollaa eivät kuitenkaan edusta todellisia merkitseviä numeroita, vaan ne on lisätty täyttämään puuttuvat numerot.

Liukulukujen kertolaskussa joudutaan myös jättämään merkitseviä bittejä pois, kun lopputulos katkaistaan. Periaatteessa binääristen liukulukujen kertolasku voidaan tehdä samanlaisella "alakkain kertomisen" algoritmilla kuin 10-järjestelmän desimaaliluvuillakin. Bittien kertominen voidaan toteuttaa vertailuoperaattorilla **AND**: 1 **AND** 1 → 1, 0 **AND** 0 → 0, 0 **AND** 1 → 0, 1 **AND** 0 → 0.

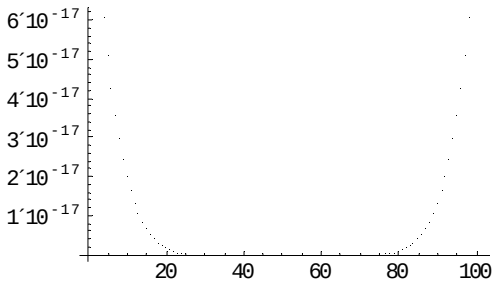
Esimerkki

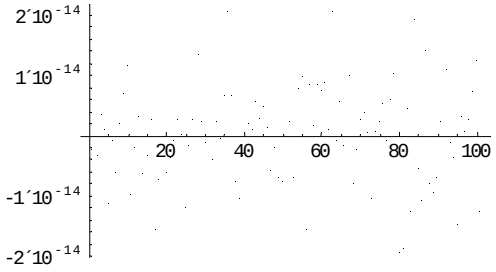
```
10.1        (2.5)
1.01        (1.25)
  101
 000
101
11.001      (3.125)
```

Kertominen palautuu siten useiden binäärilukujen yhteenlaskuksi. Tietokoneen binäärilukujen kertomisessa normalisoidut mantissat kerrotaan ja eksponentit lasketaan yhteen tavallisen potenssikaavan $m_1 Ee_1 \times m_2 Ee_2 = m_1 \cdot m_2 Ee_1 + e_2$ mukaisesti.

32-bittisten lukujen kertomisessa yhteenlaskuja sekä bittien siirtämisiä vasemmalle syntyisi aivan liikaa, jotta lukujen kertominen voitaisiin tehdä tehokkaasti. Todellisuudessa prosessoreiden liukulukujen aritmetiikka on pitkälle kehitetty ja optimoitu järjestelmä, jota ei voi yksinkertaisesti kuvata

Merkitsevien numeroiden katoamisen vaikutus näkyy esimerkiksi seuraavalla tavalla. Funktiot
$f(x) = (x-1)^8$ ja $f(x) = x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1$,
ovat täsmälleen samat, vain esitysmuoto on eri.





$(x-1)^8$ $\qquad$ $x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1$

Yllä olevissa kuvissa on laskettuna ja kuvattuna molemmille esitysmuodoille 100 arvoa lukujen 0.99 ja 1.01 väliltä (huom. vaaka-akselilla pisteiden indeksit ei x:n arvot). Merkitsevien numeroiden menetys ei tule esille potenssimuotoisessa lausekkeessa, mutta tulee polynomimuotoisessa lausekkeessa, koska siinä joudutaan tekemään paljon epäedullisia ja merkitseviä numeroita kadottavia kertolaskuja.
Kaikkein edullisin polynomien esitysmuoto laskennan kannalta saadaan *Hornerin algoritmilla*, jota voidaan soveltaa kaikille polynomilausekkeille. Tässä tapauksessa algoritmi tuottaa edellä olevalle funktiolle $f(x)$ seuraavan esitysmuodon

$$f(x) = 1 + x(-8 + x(28 + x(-56 + x(70 + x(-56 + x(28 + (-8 + x)x))))))$$