УДК 004.4 (043.2)
ББК Д 973.20-018+Б183.4

В.П. Гамаюн, Л.О.Шалаш

АЛГОРИТМІЧНИЙ БАЗИС БЕЗПОМИЛКОВОЇ АРИФМЕТИКИ

Описано алгоритми реалізації операцій додавання, віднімання, множення, ділення і логічних І-АБО, які складають алгоритмічний базис безпомилкових обчислень на ЕОМ. Наведені лістинги відповідних модулів з аналізом загальних закономірностей їх побудови.

Одним із способів представлення даних, що забезпечують прискорення й підвищення точності обчислень, є розрядно-логарифмічне (РЛ) представлення [1]. Застосування розрядно-логарифмічного представлення орієнтовано, насамперед, на розробку алгоритмів виконання мультиплікативних операцій з використанням множення-зведення і ділення-добування кореня, але також може бути арифметико-логічним, арифметико-алгоритмічним базисом обчислень в ЕОМ.

Розглянемо основні положення розрядно-логарифмічного представлення [1]. Нехай А – двійкове число розрядністю n представлено у форматі з фіксованою комою:

$$A=\sum a_i p^1,$$

де a_i ={ 0,1 }, p = 2 – основа системи числення.

Кожний розряд a_i , який не дорівнює нулю ($a_i p^1 \neq 0$) , можна представити як номер позиції (розряду) N_i . Таким чином, двійкове число А представимо набором номерів ненульових розрядів:

$$A \rightarrow A^N = \{ N_1 , N_2 , \ldots, N_i , \ldots, N_n\},$$

де N_i – номер ненульового розряду ($a_i p^1 \neq 0$).

Структура узагальненого розрядно-логарифмічного представлення для плаваючої та фіксованої коми така:

sign A	Q	$N_1 + p$	$N_2 + p$	...	$N_i + p$	...	$N_n + p$

де p – порядок.

Правила виконання розрядних операцій (N_i, N_k - ненульові розряди операндів):

додавання – $N_i + N_k = N_i$, N_k – якщо $N_i > N_k$;

$N_i + N_k = N_i + 1$ – якщо $N_i - N_k$;

віднімання – $Ni - N_k = N_{i-1}, N_{i-2}, \ldots, N_k$, якщо $N_i > N_k$,

$N_i - N_k = 0$, якщо $N_i - N_k$;

множення – $N_i * N_k = N_i + N_k$;

зсув числа на *c* розрядів – $(N_1, N_2, \ldots, N_n) \rightarrow (N_1+c, N_2+c, \ldots, N_n+c)$;

логічне І - $N_i \times N_k = 0$; якщо $N_i = N_k$; $N_i \times N_k = N_i$, якщо $N_i = N_k$;

логічне АБО - $N_i V N_k = N_i$, якщо $N_i = N_k$; $N_i V N_k = N_i, N_k$; якщо $N_i = N_k$.

При розрядно-логарифмічному представленні збільшується діапазон оброблюваних чисел, що забезпечує коректне виконання операцій і виключення процедур заокруглення і нормалізації. У табл. 1 наведені кількості двійкових розрядів n_2 і розряди розрядно-логарифмічного представлення $n_{РЛ}$, одержувані при кодуванні у відповідних n_2, а також діапазони зміни чисел D при розрядних сітках $n_{РЛ}$.

Розширення діапазону зміни даних є основою для підвищення точності обчислень у засобах обчислювальної техніки. На відміну від традиційного представлення чисел, розрядно-логарифмічні коди дозволяють скоротити вплив фактора заокруглення, тому що

Таблиця 1

n_2	7	8	9	10
$n_{РЛ}$	254	510	1022	2046
D	$2^{-127} \leq x \leq 2^{+127}$	$2^{-255} \leq x \leq 2^{+255}$	$2^{-511} \leq x \leq 2^{+511}$	$2^{-1023} \leq x \leq 2^{+1023}$

поряд з найбільш значущими розрядами числа можливо зберігати найменш значущі розряди, що звичайно відкидалися. Зазначена властивість має важливе практичне значення. Наприклад, у цифровій обробці сигналів з'являється можливість комплексної обробки сигналів з великими і малими амплітудами. Наступним позитивним фактором є забезпечення застосування більшого числа обчислювальних алгоритмів і методів, що не знаходили застосування через похибки, обумовлені структурною організацією ЕОМ. Розрядно-логарифмічне представлення є основою розробки нового програмного забезпечення, що гарантує одержання точних результатів при розв'язанні задач.

При програмній реалізації арифметичних операцій додавання, віднімання, множення, ділення були використані алгоритми. Оскільки розрядно-логарифмічне представлення є розширенням двійкового представлення даних, тому алгоритми арифметичних операцій подібні операціям двійкової арифметики і не вимагають елементно-технологічної перебудови ЕОМ. Також було проведене дослідження залежності кількості звертань до циклів арифметичних операцій від імовірності розподілу значущих розрядів. Дані прикріплені до відповідних операцій.

Операція додавання. Доданки представлені у вигляді файлів (наборів) розрядно-логарифмічних кодів. Операнди обробляються, починаючи з молодшого розряду, з виконанням наступних етапів. Виконується поелементне порівняння чисел A→{sign, Q(A), N, N_2,..N_i, …N_m} і B→{sign, Q(B), N_1, N_2,.. N_j, …N_k }: якщо $N_i \neq N_j$, то в результат C записуються обидва елементи в порядку зростання. Інакше ($N_i=N_j$) відбувається таке: обчислюється значення N_i+1, після чого отримане число порівнюється з наступними елементами чисел A та B, тобто з N_{i+1} і N_{j+1}, якщо рівності не виявлено, то в результат записується N_i+1. У противному разі виконуються порівняння наступних (старших) цифр. Слід зазначити випадок, коли доданки дорівнюють один одному: A=B. У цьому випадку результат C дорівнює одному з доданків, зсунутих на одиницю вправо, тобто

$$C \rightarrow N_1+1,\ N_2+1,\ N_3+1,\ldots,N_i+1,\ \ldots\ N_m+1.$$

Приклад. Нехай дані A({0,5,7.6.3.2.0} і B({0,3,6.4.1}. Необхідно знайти їхню суму. Порівнюючи поелементно A та B, визначимо, що в обох числах є елемент {6}, збільшуємо його на 1→ 7 і порівнюємо з наступними елементами A та B: в A елементів уже немає, а в B є елемент, що дорівнює 7, значить ще раз збільшуємо на 1→8. Ні A, ні B не містять більше елементів, значить остаточний результат буде таким:

$$C \rightarrow \{0,6,8.4.3.2.1.0\}.$$

У програмній реалізації операції додавання окремою функцією виділена перевірка на знак доданків. Наведений фрагмент програми основного циклу додавання побудовано за допомогою описаного алгоритму.

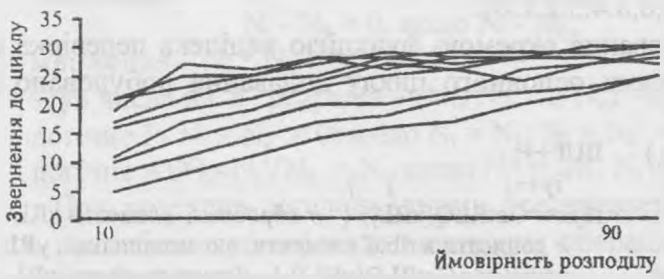
```
while( !((ix==GetRLQ(xRL)) && (iy==GetRLQ(yRL))) )
{
  if(!flag) // якщо не було переносу   {
   if(*(xRL+ix) < *(yRL+iy))
  *(rlbuf+ibuf)=*(xRL+ix); // записуємо його в буфер
    ix++;                  // збільшуємо індекс xRL
    ibuf++;      }  // збільшуємо індекс буфера
   else if(*(xRL+ix) > *(yRL+iy)) {
       *(rlbuf+ibuf)=*(yRL+iy);
       iy++;               // збільшуємо індекс yRL
       ibuf++;       }  // збільшуємо індекс буфера
      else { // елементи рівні - буде перенос
       BUF=*(xRL+ix);
       BUF++;   // запам у BUF значення елемента +1
       ix++; iy++; // зсуваємося на наступні елементи
       flag=true; }}// установлюємо прапор переносу
  else {    // був перенос
   if((BUF<*(xRL+ix)) && (BUF<*(yRL+iy))) {
      BUF++;
       iy++;              }   }
if(ix==GetRLQ(xRL)) {   // оброблені  елементи xRL
   // дописати в rlbuf елементи, що залишилися, yRL
   while(iy!=GetRLQ(yRL)) {   // поки не кінець yRL
    if(!flag) { // якщо немає переносу
     *(rlbuf+ibuf)=*(yRL+iy); // у буфер  yRL
     ibuf++;
     iy++;    }  // переходимо до елемента yRL
    else { // якщо є перенос
     if(BUF==*(yRL+iy)) { // перенос у BUF yRL
      BUF=*(yRL+iy);  // перенос  BUF yRL[iy]+1
      BUF++;
      iy++;     }       // зсуваємося до елемента yRL
     else { // перенос у BUF не дорівнює yRL
*(rlbuf+ibuf)=BUF; // пишемо BUF у буфер суми rlbuf
      ibuf++;
      flag=false; }      }     }   }// знімаємо прапор
```

```
*(rlbuf+ibuf)=BUF; //  буфер суми rlbuf
ibuf++;
flag=false; }   // знімаємо прапор переносу
else if((BUF==*(xRL+ix)) && (BUF==*(yRL+iy))){
*(rlbuf+ibuf)=BUF;
ibuf++;
BUF=*(xRL+ix);
BUF++;       // запам'ятовуємо в BUF  +1
ix++; iy++; } //зміщаємося на наступні елементи
else if(BUF==*(xRL+ix)) { // BUF дорівнює  xRL
    BUF=*(xRL+ix);
    BUF++;
    ix++;          }
  else { // BUF дорівнює тільки yRL
    BUF=*(yRL+iy);
else if(iy==GetRLQ(yRL)) { // оброблені всі  yRL
 while(ix!=GetRLQ(xRL)) {   // поки не кінець xRL
 if(!flag) { // якщо немає переносу
  *(rlbuf+ibuf)=*(xRL+ix); // у буфер елемент xRL
  ibuf++;
  ix++;  } // переходимо до елемента xRL
 else { // якщо є перенос
  if(BUF==*(xRL+ix)) { //  BUF дорівнює xRL
    BUF=*(xRL+ix);    // перенос  BUF xRL[ix]+1
    BUF++;
    ix++;        }   // зсуваємося до елемента xRL
   else { // перенос у BUF не дорівнює елементу
xRL      *(rlbuf+ibuf)=BUF; // пишемо BUF у буфер
суми rlbuf
       ibuf++;
       flag=false; }}}}}
```

В табл. 2 наведені дані дослідження: у першому стовпці та рядку розташовані ймовірності розподілу значущих розрядів у доданках, в інших – кількість звернень до основного циклу операції додавання.

Таблиця 2

Розподіл, %	10	20	30	40	50	60	70	80	90
10	5	8	10	11	16	17	21	21	19
20	8	11	14	17	19	21	25	23	27
30	11	14	17	19	21	24	25	25	26
40	14	17	20	23	24	25	28	27	28
50	16	18	20	25	27	28	26	29	28
60	17	21	22	26	26	27	28	28	29
70	20	22	26	24	28	28	28	28	29
80	22	23	27	27	30	30	28	30	29
90	25	25	28	29	30	30	27	32	32



Залежність ймовірності розподілу від кількості звернень

Дані з табл. 2 проілюстровані на рисунку, з якого видно, що залежність між ймовірністю розподілу та кількістю звернень до циклу близька до лінійної.

Операція віднімання. Операція віднімання реалізується як додавання кодів зменшуваного і від'ємника, при цьому від'ємник представлений у зворотному коді.

Приклад. Нехай дані A→{0,3,9.8.1} і B→{0,4,8.7.5.0}. Необхідно знайти їхню різницю. Числа A та B позитивні й A>B. Записуємо від'ємник у зворотному коді: старший розряд зменшуваного 9,9-1=8; $B_{зв}$ →{0,5,6.4.3.2.1}. Складаємо A без старшого розряду {0,2,8.1} і $B_{зв}$, результат C→{0,4,8.6.5.0}.

Основному циклу віднімання передує перевірка зменшуваного і від'ємника на знаки і перевірка на "більше–менше". У наведеному далі фрагменті показане одержання зворотного коду від'ємника і сам алгоритм віднімання.

```
if ((yRLREV=(RLType) malloc(sizeof(int)* sz)) == NULL) {
   printf("Not enough memory to allocate buffer Minus\n");
   exit(1);  /* terminate program if out of memory */  }
```

```
iyr=sz-1;       // на старший елемент yRLREV
 // Початкове значення r = Старший розряд xRL-1.

 for(r=(*(xRL+ix)-1); iy!=1; r--) {
  if(r!=*(yRL+iy)) {
   *(yRLREV+iyr)=r;
   iyr--;   }
  else
   iy--; }
 *(yRLREV+iyr)=*(yRL+2); // last elem of yRL
 *(yRLREV)=0;
 *(yRLREV+1)=sz;

 if(ix==2) {
  if(rezRL==xRL) FreeRL(xRL);
  if(rezRL==yRL) FreeRL(yRL);
 return(yRLREV); }
// xRLwithoutFirst
 if ((xRLwoFirst=(RLType) malloc(sizeof(int)*(ix))) == NULL) {
  printf("Not enough memory to allocate buffer Minus\n");
  exit(1);  /* terminate program if out of memory */  }

 *(xRLwoFirst)=0;
 *(xRLwoFirst+1)=ix;
 for(;ix>2;ix--)
   *(xRLwoFirst+ix-1)= *(xRL+ix-1);
rezRL=Plus (rezRL, xRLwoFirst, yRLREV);
```

Дослідження щодо залежності ймовірності розподілу значущих розрядів та кількості звернень до основного циклу не мають сенсу, оскільки віднімання зводиться до додавання, тобто дані цілком ідентичні.

Операція множення. Алгоритм множення полягає в поелементному додаванні масиву, що представляє один співмножник, з елементами масиву, що представляє інший співмножник. Одержані часткові добутки складаються за алгоритмом розрядно-логарифмічного додавання.

Приклад. Нехай дані A({0,2,2.1} і B({0,2,3.1}. Необхідно знайти їхній добуток. Часткові суми: {3.2} і {5.4}. Результат C дорівнює сумі {3.2} і {5.4}, тобто {0,4,5.4.3.2}. Дали наведено повний лістинг функції множення.

```
RLType RLMult(RLType rezRL, RLType xRL, RLType yRL) {
 int SIGN;
 int ix, iy;
 RLType BUF;
 RLType rez=rezRL;

 if((GetRLSign(xRL)*GetRLSign(yRL))>0) SIGN=0;
 else SIGN=1;

 if ( (BUF=(RLType) malloc(sizeof(int)*GetRLQ(yRL))) == NULL) {
  printf("Not enough memory to allocate buffer RLMult\n");
  exit(1);  /* terminate program if out of memory */  }

 *(BUF)=0;
 *(BUF+1)=GetRLQ(yRL);
 rezRL=x10xRL(0); // початкове значення rezRL=0;

 for(ix=2; ix<GetRLQ(xRL); ix++) {
  for(iy=2; iy<GetRLQ(yRL); iy++) {
```

```
    *(BUF+iy)= *(xRL+ix) + *(yRL+iy);   }
  rezRL=Plus(rezRL, rezRL, BUF); }

 FreeRL(BUF);
 if(rez==xRL)
  FreeRL(xRL);

 if(rez==yRL)
  FreeRL(yRL);

 *(rezRL)=SIGN;
return(rezRL);}
```

Дослідження залежності ймовірності розподілу значущих розрядів та кількості звернень до основного циклу множення показало, що кількість звернень залежить лише від імовірності заповнення першого множника (табл. 3).

Таблиця 3

Розподіл, %	10	20	30	40	50	60	70	80	90
10	3	6	9	12	15	18	21	27	30
20	3	6	9	12	15	18	21	27	30
30	3	6	9	12	15	18	21	27	30
40	3	6	9	12	15	18	21	27	30
50	3	6	9	12	15	18	21	27	30
60	3	6	9	12	15	18	21	27	30
70	3	6	9	12	15	18	21	27	30
80	3	6	9	12	15	18	21	27	30
90	3	6	9	12	15	18	21	27	30

Проаналізувавши наведені дані, можна зробити висновок, що операцію множення можливо оптимізувати. Необхідно порівнювати кількість елементів в розрядно-логарифмічних числах. Якщо кількість елементів першого множника більша ніж у другого, тоді треба поміняти множники місцями, що і зменшить кількість звернень до основного циклу операції множення.

Операція ділення. Ділення чисел у розрядно-логарифмічному представленні виконується за алгоритмом, подібним «ділення кутом». Спробна цифра частки визначається як різниця між старшою одиницею діленого (залишку) і старшою значущою одиницею дільника. Якщо залишок негативний, то цифра частки коректується на –1. У випадку повторення цифри частки виконується також корекція на –1. Ділення зупиняється, якщо отримано нульовий залишок або стільки цифр частки, скільки обумовлено заздалегідь.

Приклад. Нехай дані A→{0,5,9.6.5.4.0} і B→{0,2,2.0}. Необхідно знайти результат їхнього ділення.

```
_9.6.5.|4.0    2.0
 8.6   |      6.5.4.3.2.0
  _ 8.5.4.0
     7.5
   _ 7.4.0
     6.4
   _ 6.0
     5.3
    _ 4.3.0
      4.2
     _2.0
      2.0
     NULL
```

Далі наведена програмна реалізація описаного алгоритму ділення.

```
RLType RLDiv(RLType rezRL, RLType xRL, RLType yRL)
if ( (r=(RLType) malloc(sizeof(int)*3)) == NULL) {
  printf("Not enough memory to allocate buffer RLDiv\n");
  exit(1); /* terminate program if out of memory */
       }
       *(r)=0;
       *(r+1)=3;
      razn=x10xRL(0);
razn=Plus(razn, razn, xRL); // початкове значення razn=xRL
       do {
        *(r+2)=*(razn+GetRLQ(razn)-1)-*(yRL+GetRLQ(yRL)-1);
        vich=RLMult(vich, r, yRL);
        if(CompModRL(vich, razn)==1) {
         FreeRL(vich);
         *(r+2)=*(r+2)-1;
         vich=RLMult(vich, r, yRL); }
        if ( (rez=(RLType) realloc(rez, sizeof(int)*(irez+1))) == NULL) {
       printf("Not enough memory to allocate buffer RLDiv\n");
         exit(1); /* terminate program if out of memory */ }
if(*(r+2)<0) kolotr++;
  *(rez+irez)=*(r+2);
  irez++;
  *(rez+1)=irez;
razn=Minus(razn, razn, vich);
 FreeRL(vich); }
 while((!isNULRL(razn))&&kolotr!=TT);
 FreeRL(razn);
 FreeRL(r);
 if(rezRL==xRL)
  FreeRL(xRL);
 if(rezRL==yRL)
  FreeRL(yRL);
if ( (rezRL=(RLType) malloc(sizeof(int)*irez)) == NULL) {
 printf("Not enough memory to allocate buffer RLDiv\n");
exit(1); /* terminate program if out of memory */ }
 *(rezRL)=SIGN;
 *(rezRL+1)=irez;
 irez--;
 for(i=2;i<*(rezRL+1);i++, irez--)
  *(rezRL+i)=*(rez+irez);
 FreeRL(rez);
 return(rezRL);}
```

Дослідження щодо залежності ймовірності розподілу значущих розрядів та кількості звернень до основного циклу ділення не несуть ніякого змісту, оскільки при діленні кількість звернень до основного циклу залежить від заданої точності або кількості значущих розрядів після коми.

Операція АБО. Логічна операція АБО здійснюється аналогічно операції арифметичного додавання. Розрядно-логарифмічні коди аналізуються, починаючи з молодших розрядів, з виконанням наступних процедур. Виконується поелементне порівняння й в результат записуються всі розряди обох чисел без повторень. Наведений фрагмент функції ілюструє порозрядне порівняння.

```
while( !((ix==GetRLQ(xRL)) && (iy==GetRLQ(yRL))) ) { // закінчити, якщо в обох
// числах досягнутий кінець
  if(*(xRL+ix) < *(yRL+iy)) { // елемент числа xRL менше yRL
   *(rlbuf+ibuf)=*(xRL+ix); // записуємо його в буфер
   ix++;                    // збільшуємо індекс xRL
   ibuf++;                  } // збільшуємо індекс буфера
  else if(*(xRL+ix) > *(yRL+iy)) { // елемент числа yRL менше xRL
     *(rlbuf+ibuf)=*(yRL+iy); // записуємо його в буфер
     iy++;                    // збільшуємо індекс yRL
     ibuf++;                  } // збільшуємо індекс буфера
    else { // елементи рівні
     *(rlbuf+ibuf)=*(xRL+ix); // записуємо його в буфер
     ibuf++;                  // збільшуємо індекс буфера
     ix++; iy++; } // зсуваємося в обох числах на наступні елементи
```

Операція I. Логічна операція I також заснована на порозрядному порівнянні чисел A→{sign, Q(A), N_1, N_2,.. N_i, …N_m} і B→{sign, Q(B), N_1, N_2,.. N_j, …N_k }... Результат складається тільки з елементів, що входять і в число A, і в число B (N_i=N_j). Лістинг функції наведений далі

```
while( (ix!=GetRLQ(xRL)) && (iy!=GetRLQ(yRL)) )  {
  //закінчити, коли в одному з чисел досягнуто кінець
  if(*(xRL+ix) < *(yRL+iy))  // елемент числа xRL менше yRL
    ix++;                    // збільшуємо індекс xRL
  else if(*(xRL+ix) > *(yRL+iy))  // елемент числа yRL менше xRL
       iy++;                      // збільшуємо індекс yRL
     else { // елементи рівні
       *(rlbuf+ibuf)=*(xRL+ix);  // записуємо його в буфер
       ibuf++; ix++; iy++;       } }
 *(rlbuf)=*(xRL)&*(yRL);
```

На основі реалізованих математичних операцій можливе виконання більш складних функцій. Приклад програмної реалізації обчислення функції $\sum_{i=1}^{n} a_i b_i$ наведений далі.

```
RLType MAdd(int A[SM] ,int B[SM]) {
 int i;
 RLType Ai;
 RLType Bi;

 RLType S=x10xRL(0);
 for(i=0;i<SM;i++) {
  Ai=RLMult(Ai, Ai=x10xRL(A[i]), Bi=x10xRL(B[i]));
  S=RLPlus(S,S,Ai);
  FreeRL(Ai);FreeRL(Bi);  }
 return(S);}
```

Приклад. Нехай у формулі $\sum_{i=1}^{n} a_i b_i$ n=16, масив a_i →{1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000, 12000, 13000, 14000, 15000, 16000}, а масив b_i →{1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999, 10000, 11111, 12222, 13333, 14444, 15555, 16666}. Результат виконання функції:

$A_{РЛ}$→{+,17, 4.5. 6.7. 8.9. 11. 13. 15. 16. 17. 19. 24. 26. 27. 28. 30.};

A_{10}=1 561 046.

Загальними процедурами, використовуваними у всіх операціях, є зведення подібних і сортування даних, що відрізняє дану арифметику від класичної. При апаратній реалізації розглянутого базису необхідно виконати зазначені процедури. При цьому архітектура машини не відрізняється від відомих рішень, наприклад, від структури асоціативно-логічних процесорів. Обробка з використанням розглянутого базису реалізується без заокруглень і операцій нормалізації – за винятком обробки періодичних дробів. Комп'ютерна технологія на основі розрядно-логарифмічного представлення може бути застосована для високоточних розрахунків складних виробів, моделювання на якісно новому рівні багатофакторних систем, розробки високопродуктивних рівнобіжних комп'ютерних засобів.

Список літератури

1. *Гамаюн В.П.* Макрооператорные методы вычисления многоместных произведений// Микропроцессорные системы и их применение. – К.: Ін-т кібернетики ім. В.М.Глушкова АН УРСР , 1990. – С.23-28.

Стаття надійшла до редації 21.06.01