Multi-Precision Arithmetic

Ivor Page[1]

6.1 Multi-Length Operations

Consider the problem of writing functions to perform arithmetic on integers that are larger than the precision of the computer. Such packages are said to implement *multi-length* or *multi-precision* operations. For the sake of clarity in our example, we will keep the largest available integer to only 4 bits (64 bits would be more appropriate for modern PCs).

6.1.1 Using Unsigned Numbers

First we have to decide how to represent values larger than the 4 bit integers provided in the hardware. Say we just want to use 8 bits for each value. In our first attempt, we consider using two unsigned ints to represent each value, where the most significant bit of the left-half of a number is the sign bit.

Here are some examples where two 4 bit unsigned ints are used to represent an 8 bit 2's complement value:

Value in Decimals	Double-length 2's complement
-128	1000 0000
-127	1000 0001
-1	1111 1111
0	0000 0000
1	0000 0001
126	0111 1110
127	0111 1111

This representation has the advantage of using all the available bits. How can we program the arithmetic operations on double length numbers? Consider addition. We will denote a double length value A by the concatenation of its two halves, A_H and A_L (A_H is A high and A_L is A low). A_H is the most significant half.

To add two values A and B, we first add A_L and B_L giving the least significant half of the answer, X_L. The bit-pattern of X_L is correct no matter whether we declare the values to be signed or unsigned. But how can we tell when a carry should take place into the left-half of the result?

[1]University of Texas at Dallas

With unsigned values, where all 4 bits of the right-half of each number are employed, there is no simple way of telling when a carry should occur, unless the hardware provides a carry-flag that can be tested from within the high-level language being used.

Alternatively, we could make use of only 3 of the bits in the right-half of each number, keeping the most significant bit of that half as a kind of carry flag.

The representation is then as follows:

Value in Decimals	Double-length 2's complement
-64	1000 0000
-63	1000 0001
-1	1111 0111
0	0000 0000
1	0000 0001
62	0111 0110
63	0111 0111

Say, in this scheme, we add the two right-half values, 0011 and 0111. The result would be 1010. We extract most significant bit and use it as a carry into the sum of the left-hand components. The right-half of the result is then 0010.

Here are some examples:

$$\begin{array}{rr} \multicolumn{2}{c}{26 + 11 = 37} \\ \hline & 0011\ 0010 \\ + & 0001\ 0011 \\ \hline & 0100\ 0101 \end{array} \qquad \begin{array}{rr} \multicolumn{2}{c}{30 + 15 = 45} \\ \hline & 0011\ 0110 \\ + & 0001\ 0111 \\ \hline & 0101\ 0101 \end{array}$$

$$\begin{array}{rr} \multicolumn{2}{c}{-34 + 15 = -19} \\ \hline & 1011\ 0110 \\ + & 0001\ 0111 \\ \hline & 1101\ 0101 \end{array} \qquad \begin{array}{rr} \multicolumn{2}{c}{-34\ +\ -15 = -49} \\ \hline & 1011\ 0110 \\ + & 1110\ 0001 \\ \hline & 1001\ 0111 \end{array}$$

The scheme easily extends to any length.

6.1.2 Using Signed Numbers

We return to the possibility of using all the 8 bits. The representation is again as follows:

Value in Decimals	Double-length 2's complement
-128	1000 0000
-127	1000 0001
-1	1111 1111
0	0000 0000
1	0000 0001
126	0111 1110
127	0111 1111

Some right-half values appear negative if signed ints are used, even though they may be components of positive values.

To make this scheme work, we would need to use the overflow flag as a kind of carry flag (and we would need to be able to detect an overflow from within our high-level language). If signed values are used, does an overflow in the addition of two right-half values correspond to the desired carry-out? The answer is NO.

Consider the example with the right-half sum $1100 + 1100$.

Here the result is 1000 but no overflow is generated because the result fits within the 4-bit scheme. In Decimals, the sum represents $-4\ +\ -4$ and the result fits within the 4-bit 2's complement system. At the hardware level, there is a carry-into and out of the sign-bit position.

There is no simple way to employ all the available bits of the right-half parts of each number, unless the hardware provides a carry-flag that can be tested from within the high-level language being used.

There are solutions using only 3 bits of the right-half of each number, both with signed and unsigned representations.

6.1.3 Using two different lengths

A good way to implement the multi-precision package is to store the long numbers in a vector of unsigned ints but to do the basic arithmetic operations with longs (where longs are twice the length of ints). For example, if an int is 32 bits and a long is 64 bits, then the digits of our multi-precision number can have range 0 to $2^{32} - 1$, using all the bits of the unsigned ints. This creates a radix 2^{32} system. When two of these multi-precision values are added, the algorithm takes two ints from corresponding positions in the two numbers, converts them to longs, and adds the longs. The carry bit is easily masked out from the 64 bit answer and the low-order 32 bits of the answer are stored as an int in the result vector.

The basic operation within a multi-precision multiply is to take two "digits", one from each of the numbers being multiplied, convert them to longs, and multiply them, giving a 64 bit result. The result forms two digits that must be added into the developing answer vector.

Similarly, during a division, the main division step is to take a two digit value (64 bits) from the dividend and divide it by a single digit (32 bits) of the divisor.

The use of ints and longs matches exactly what we need to do in these main arithmetic operations.

6.1.4 Very Large Integers

Using the scheme developed above, it is possible to string many integers together to make up a mult-precision value as large as required. Addition requires the sum of corresponding pairs of components of the two multi-length values, starting at the least significant end, and propagating carries between them.

Subtraction is performed by negating of the subtrahend and adding the result to the minuend.

Multiplication is more complicated. Consider the product of two operands of size m and n ints respectively. We can think of the numbers as being m and n digits in a radix $\beta = 2^{k-1}$ system, where each int has k bits. The product is formed by multiplying each digit of the multiplicand by each digit of the multiplier and summing the results after appropriate scaling. The ideas are demonstrated by a decimal example:

```
     1 9 3 6                  |     1 3 5 5 2  = multiplicand*7
   x  2 8 4 7                 |        2 4    6*4
 ------------                 |     ---------
         4 2  6*7             |     1 3 7 9 2
     + 2 1    3*7             |       1 2     3*4
      ------                  |     ---------
       2 5 2                  |     1 4 9 9 2
   + 6 3      9*7             |     3 6       9*4
     -------                  |     ---------
     6 5 5 2                  |     5 0 9 9 2
   + 7        1*7             |     4         1*4
    --------                  |     ---------
   1 3 5 5 2  = multiplicand*7 |     9 0 9 9 2 = multiplicand*47

  continued in column to right:
```

The example is only 1/2 finished. Each iteration consists of multiplying one digit of the multiplicand by one digit of the multiplier and adding the appropriately shifted double-length result to the developing answer. In each of the addition steps the carry could propagate throughout the digits of the developing answer. Therefore the time to compute the product comprises $m \times n$ iterations, each of which is a digit-by-digit product, giving a double-length result, followed by an addition that could carry along all the digits of the answer. The answer will finally be of length $m \times n$ digits. The time is then $O(nm(m+n))$, where the multiplier has length n digits and the multiplicand has m digits.

It is possible to prevent the carry ripple during the addition step. Consider the point in the above example where the second digit (4) of the multiplier is used.

The developing answer is

```
1 3 5 5 2  = multiplicand * 7
```

We set the variable `carry` to 0 and compute the product 6*4 = 24. The right-hand part of the product (4) is added, together with the `carry`, to the correct digit of the developing result, and the carry from the result of that add, if any, is saved in `carry`.

```
1 3 5 5 2  = multiplicand * 7
      4
      0    = carry
---------
1 3 5 9 2
```

There was no carry during this addition, so `carry` is still zero. We add the left-hand part of the product into `carry` giving `carry` = 2.

Now we compute the next product, 3*4 = 12.

The right-hand part of the product (2) is added, together with the `carry`, to the correct digit of the developing result, and the carry from the result of that add, if any, is saved in `carry`.

```
1 3 5 9 2  = multiplicand * 7
    2
    2      = carry
---------
1 3 9 9 2
```

There was no carry during this addition, so `carry` is zero. We add the left-hand part of the product into `carry` giving `carry` = 1.

Now we compute the next product, 9*4 = 36.

The right-hand part of the product (6) is added, together with the `carry`, to the correct digit of the developing result, and the carry from the result of that add, if any, is saved in `carry`.

```
1 3 9 9 2  = multiplicand * 7
  6
  1        = carry
---------
1 0 9 9 2
```

There was a carry of 1 during this addition, so `carry` is set to 1. We add the left-hand part of the product into `carry` giving `carry` = 4.

Now we compute the next product, 1*4 = 04.

The right-hand part of the product (4) is added, together with the `carry`, to the correct digit of the developing result, and the carry from the result of that add, if any, is saved in `carry`.

```
1 0 9 9 2  = multiplicand * 7
4
4          = carry
---------
9 0 9 9 2
```

There was no carry during this addition, so `carry` is 0. We add the left-hand part of the product into `carry` giving `carry` = 0.

Since this is the last iteration with this digit of the multiplier, the final carry is added to the appropriate digit of the developing answer. In this case, the developing answer doesn't change.

Although we have only demonstrated the process with the 2nd digit of the multiplier, it is used throughout the entire process.

Consider the range of possible results that can occur during the addition. For the above scheme to work, that range must be (in decimal notation) no more than 19 or a carry of more than 1 could occur. We are adding an arbitrary valued digit of the developing answer to the right-hand half of the product of 2 digits, together with a carry value that (we hope) is no more than 1. The result can be no more than 9+9+1, and our hopes are secure.

This scheme is known as carry-save addition and is widely used in multipliers. The time to multiply the m and n digit numbers using carry-save addition is $O(mn)$.

6.2 Negative Multi-Precision Values

Although this note began with examples using 2's complement, sign-magnitude representation may be of interest since it greatly simplifies multiplication. As we shall see later in the module on multiplication, multiplication of 2's complement multi-precision values presents several challenges. To illustrate the difficulties, consider the product of two values, both of k bits, $A \times B$. We call A the *multiplicand* and B the *multiplier*. If both are positive, there is no problem.

If only the multiplicand is negative, $A = -X$, and the multiplier is positive, $B = Y$ we represent A by $2^k - X$. The product is then $(2^k - X) \times Y = 2^kY - XY$. The required answer is $2^{2k} - XY$. One solution is to sign-extend A to $2k$ bits, so that $A = 2^{2k} - X$. Then the result is $2^{2k}Y - XY$. Since 2^{2k} is beyond the range of the number system, its term is discarded.

If only the multiplier is negative, $B = -Y$, B is represented by $2^k - Y$ and the product is $X \times (2^k - Y) = 2^kX - XY$. The required result is $2^{2k} - XY$. One solution is to add a the multiplicand, left shifted k places, 2^kX, to the result.

If both operands are negative, the result is $(2^k - X) \times (2^k - Y) = 2^{2k} - 2^k(X + Y) + XY$. A combination of the previous two solutions gives the correct result, XY.

We shall discuss alternative solutions in the module on integer multiplication.

These problems seem easy to fix when both numbers are of the same length. They create a few more difficulties for programmers when the values are of different lengths (numbers of ints). We shall leave these problems as implementation details at this point since, even if 2's complement is used, conversion to and from sign-magnitude representation before and after each multiplication operation would not greatly reduce the speed of the operation.

6.3 Division

It is only necessary to consider an example in decimals to understand the complexity of the problem:

$$\frac{u}{v} = \frac{Dividend}{Divisor} = \frac{7315982649963}{28453}$$

At each iterative step, we guess a value for the next digit of the answer, q_i, multiply the divisor by this guess, scale the result by left shifting it, and subtract the result from the reducing remainder (initially remainder = dividend). Unfortunately the guess can be wrong. If so, the guess will need adjusting and further arithmetic will be necessary. The guess is obtained by solving a sub-problem.

Consider the simplest way of guessing q_i, $q_i = \lfloor 7/2 \rfloor = 3$. The result is 1 too large. If the most significant digit of the divisor had been 1, the result would be 7, that's 5 larger than the correct answer. And if the most significant digit of the divisor had been 9, the answer would be 0. Similar problems arise if we use $q_i = \lfloor 73/28 \rfloor$, and we appear to be in even worse trouble is we use $q_i = \lfloor 73/2 \rfloor$.

The solution is to scale the divisor before beginning the division process such that its most significant digit, v_1, has value $5 \leq v_1 \leq 9$. If necessary, we can do this by multiplying v by 2, 4, or 8. In binary, the scaling will be a left shift. A corresponding left shift of the dividend, u, will keep the quotient correct, although it may extend the length of the dividend by one digit. With this adjustment, it is possible to show that the guess is never too small and, only in rare circumstances, will it be 2 too large. A further refinement will ensure that the guess will only rarely be 1 too large.

6.4 Knuth's Division Algorithm

Knuth Vol II[2] contains the following algorithm, which is a modification of an algorithm by Pope and Stein[3]

Assume that the multi-precision system is based on integers of k bits such that the range of each *digit* in the system is $[0, 2^k - 1]$. This system is said to have radix $\beta = 2^k$ (using all the bits of each int).

The division sub-problem is then, find $q = \lfloor u/v \rfloor$, given the n+1 digit value $u = (u_0, u_1, \cdots, u_n)$, and the n digit value $v = (v_1, v_2, \cdots, v_n)$, where u and v are radix β numbers, and $u/v < \beta$.

The dividend has the extra int, u_0, produced by the scaling operation. We only include $n+1$ digits of the scaled dividend and all n digits of the scaled divisor.

The setup corresponds to the following adjustment of our original division problem in preparation for finding the first digit of the quotient:

$$\frac{u}{v} = \frac{146319}{56906}$$

Note that the condition $u/v < \beta$ is equivalent to the condition $\lfloor u/\beta \rfloor < v$, which means $(u_0, u_1, \cdots, u_{n-1}) < (v_1, v_2, \cdots, v_n)$. Also note that, if we write $r = u - qV$, then q is the unique integer such that $0 \leq r < v$.

The first attempt at a solution to the sub-problem is to make a guess at q based on the most significant digits of u and v, as we did in our decimal example:

$$\hat{q} = min\left(\left\lfloor \frac{u_0\beta + u_1}{v_1} \right\rfloor, \beta - 1\right)$$

We choose $\hat{q}$ to be the floor of the two most significant digits of the dividend divided by the

[2]The Art of Computer Programming Vol II, Seminumerical Algorithms, Addison Wesley

[3]David A Pope and Marvin L. Stein, CACM 3 1960, pp652-654

most significant digit of the divisor. If this value is larger than $\beta - 1$, we replace it by $\beta - 1$. We shall not cover the proofs here (consult Knuth Vol II), but it is easy to show that $\hat{q} \geq q$ and if $v_1 \geq \lfloor \beta/2 \rfloor$, then $\hat{q} - 2 \leq q \leq \hat{q}$.

This result states that *normalization* of the divisor will ensure that our guess, $\hat{q}$ is never more than 2 too large. Normalization can be accomplished by pre-multiplying dividend and divisor by $\lfloor \beta/(v_1 + 1) \rfloor$. This does not alter the result nor does it change the number of digits in v.

The algorithm follows:

Given radix β non negative values $u = (u_1, u_2, \cdots, u_{m+n})$, $v = (v_1, u_2, \cdots, u_n)$, where $v_1 > 0$ and $n > 1$ compute quotient $q = u/v = (q_0, q_1, \cdots, q_m)$ and remainder $r = u \ mod \ v = (r_1, r_2, \cdots, r_n)$

D1 Normalize:

$d \leftarrow \lfloor \beta/(v_1 + 1) \rfloor$.

$(u_0, u_1, \cdots, u_{m+n}) \leftarrow (u_1, u_2, \cdots, u_{m+n}) \times d$.

$(v_1, v_2, \cdots, v_n) \leftarrow (v_1, v_2, \cdots, u_n) \times d$.

If $d = 1$ choose any value of d that results in $v_1 \geq \lfloor \beta/2 \rfloor$.

The simplest way to normalize in a binary computer is to left-shift the int v_1 p places until its most significant bit is 1 and then left shift both u and v by that amount.

Note that these are multi-precision left-shifts.

D2 Initialize counter j:

$j \leftarrow 0$

D3 Calculate $\hat{q}$:

If $u_j = v_1$ set $\hat{q} \leftarrow \beta - 1$, otherwise set $\hat{q} \leftarrow \lfloor (u_j\beta + u_{j+1})/v_1 \rfloor$.

This results in $\hat{q}v_1 + q_{rem} = u_j\beta + u_{j+1}$, or $q_{rem} = u_j\beta + u_{j+1} - \hat{q}v_1$.

Now test if $v_2\hat{q} > (u_j\beta + u_{j+1} - \hat{q}v_1)\beta + u_{j+2}$, or equivalently, $v_2\hat{q} > q_{rem}\beta + u_{j+2}$

If so, decrease $\hat{q}$ by 1, increase q_{rem} by v_1 and repeat this test. This test determines at high speed if $\hat{q}$ is 1 too large and eliminates all cases in which it is 2 too large.

D4 Multiply and subtract:

Compute $\hat{q} \times (v_1, v_2, \cdots, v_n)$ and subtract the result from $(u_j, u_{j+1}, \cdots, u_{j+n})$.

The multiplication by the single precision value $\hat{q}$ and the subtraction can be combined into one operation.

The result of this step should be positive. If not, it means that $\hat{q}$ was one too large. Leave the results as a β complement negative value.

D5 Test remainder:

$q_j \leftarrow \hat{q}$.

If the result of step D4 was negative, goto step D6, otherwise goto step D7.

D6 Add back:

The probabilty of having to perform this step is $3/\beta$.

Decrease q_j by 1 and add $(0, v_1, v_2, \cdots, v_n)$ to $(u_j, u_{j+1}, \cdots, u_{j+n})$. A carry out of the u_j position will occur and should be ignored since it cancels a *borrow* that was generated in step D4.

D7 Loop on j:

Increase j by 1. If $j \leq m$ goto step D3.

D8 Denormalize:

$(q_0, q_1, \cdots, q_m)$ is the quotient and the remainder is obtained by dividing $(u_{m+1}, u_{m+2}, \cdots, u_{m+n})$ by d (or by right shifting p places if a shift operation was used in step D1).

The step D4 is known as a *restoring* step. Note that it is needed only on very rare occasions. For a realistic implementation using 16 bit integers, the probability of having to perform this step is 3/32,768 = 0.01%.

The running time of the algorithm is as follows:

There are m iterations involving a constant time step to compute q_j, the product of $(v_1, v_2, \cdots, v_n)$ and q_j, an n bit subtraction, and some constant time testing and loop control operations. The product and subtraction each take $O(n)$ time, so the overall running time is $O(mn)$.

Decimal Example:

$$19134786 \div 241$$

To normalize, multiply both operands by $d = 4$ giving:

$$76539144 \div 964$$

Calculate $\hat{q} = 76 \div 9 = 8, \quad q_{rem} = 4$

$$\begin{aligned} \text{Test } \; v_2\hat{q} \quad &> \quad q_{rem}\beta + u_{j+2} \quad ? \\ 6 \times 8 \quad &> \quad 4 \times 10 + 5 \quad ? \end{aligned}$$

The test is positive, so $\hat{q}$ is reduced by 1, q_{rem} is increased by v_1, giving $\hat{q} = 7$, $q_{rem} = 13$, and the test is repeated:

$$6 \times 7 \quad > \quad 13 \times 10 + 5 \quad ?$$

This time the test failes.

Set $q_1 = \hat{q} = 7$

Multiply v by q_1 and subtract:

```
76539144
6748
--------
 9059144
```

Calculate the new $\hat{q} = 90 \div 9 = 10$.

Set $\hat{q} = \beta - 1 = 9$, and $q_{rem} = 9$.

$$\begin{array}{rcl} \text{Test } \; v_2\hat{q} & > & q_{rem}\beta + u_{j+2} \quad ? \\ 6 \times 9 & > & 9 \times 10 + 5 \quad ? \end{array}$$

The test fails.

Set $q_2 = \hat{q} = 9$.

Multiply and subtract:

```
9059144
8676
-------
 383144
```

Calculate the new $\hat{q} = 38 \div 9 = 4$ and $q_{rem} = 2$.

$$\begin{array}{rcl} \text{Test } \; v_2\hat{q} & > & q_{rem}\beta + u_{j+2} \quad ? \\ 6 \times 4 & > & 2 \times 10 + 3 \quad ? \end{array}$$

The test is positive, reduce $\hat{q}$ by 1, giving 3, add v_1 to q_{rem} giving 11, and test again:

$$6 \times 3 \quad > \quad 11 \times 10 + 3 \quad ?$$

The test fails.

Set $q_3 = \hat{q} = 3$.

Multiply and subtract:

```
383144
2892
------
 93944
```

Calculate the new $\hat{q} = 93 \div 9 = 10$.

Set $\hat{q} = \beta - 1 = 9$, and $q_{rem} = 12$.

$$\begin{array}{rcl} \text{Test } \ v_2\hat{q} & > & q_{rem}\beta + u_{j+2} \quad ? \\ 6 \times 9 & > & 12 \times 10 + 3 \quad ? \end{array}$$

The test fails.

Set $q_4 = \hat{q} = 9$, and multiply and subtract:

```
 93944
 8676
 -----
  7184
```

Calculate the new $\hat{q} = 71 \div 9 = 7$, and $q_{rem} = 8$.

$$\begin{array}{rcl} \text{Test } \ v_2\hat{q} & > & q_{rem}\beta + u_{j+2} \quad ? \\ 6 \times 7 & > & 8 \times 10 + 8 \quad ? \end{array}$$

The test fails.

Set $q_5 = \hat{q} = 7$.

Multiply and subtract:

```
 7184
 6748
-------
  436
```

Denormalize the remainder by dividing by $d = 4$, giving remainer 109.

The answer is 79397 remainder 109.

Each iteration of the division process reduces the length of the dividend by one digit and produces one digit of the quotient. It is therefore possible to store the digits of the quotient in the vacated digits of the dividend. At the end of the process, the digits of the old dividend will be replaced by the $m+1$ digits of the quotient, followed by the n digits of the remainder.

6.5 Components of the Multi-Precision Package

Here is a list of several support functions necessary for multi-precision arithmetic.

Sign extend a negative value of length a ints to length a, $a > b$.

Left or right shift a positive value by p places.

Multiply a multi-precision value by a single precision value.

Study the BigInteger package provided in the Java language.