***Laboratoire de l'Informatique du Parallélisme***

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 8512

# *Multiplication by an Integer Constant*

Vincent Lefèvre

January 1999

Research Report N° 1999-06

**École Normale Supérieure de Lyon**
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

# Multiplication by an Integer Constant

Vincent Lefèvre

January 1999

**Abstract**

We present an algorithm allowing to perform integer multiplications by constants. This algorithm is compared to existing algorithms. Such algorithms are useful, as they occur in several problems, such as the Toom-Cook-like algorithms to multiply large multiple-precision integers, the *approximate* computation of consecutive values of a polynomial, and the generation of integer multiplications by compilers.

**Keywords:** multiplication, addition chains

**Résumé**

Nous présentons un algorithme permettant de faire des multiplications entières par des constantes. Cet algorithme est comparé à d'autres algorithmes existants. De tels algorithmes sont utiles, car ils interviennent dans plusieurs problèmes, comme les algorithmes du style Toom-Cook pour multiplier des entiers à grande précision, le calcul approché de valeurs consécutives d'un polynôme et la génération de multiplications entières par les compilateurs.

**Mots-clés:** multiplication, chaînes d'additions

# 1 Introduction

The multiplication by integer constants occurs in several problems, such as the Toom-Cook-like algorithms to multiply large multiple-precision integers [3], the *approximate* computation of consecutive values of a polynomial (we can use an extension of the finite difference method [2] that needs multiplications by constants), and the generation of integer multiplications by compilers (some processors do not have an integer multiplication instruction, or this instruction is relatively slow). We look for an algorithm that will generate shift, add and sub instructions to perform such a multiplication, which would be faster than a general purpose integer multiplication. We assume that the constant may have several hundreds of bits.

Here we are allowed to do shifts (i.e., multiplications by powers of 2) as fast as additions. So, this is a more difficult problem than the well-known *addition chains* problem [2].

This problem has already been dealt with, to have an algorithm for compilers, but for shorter constants (e.g., 32 bits). Most compilers implement an algorithm from Robert Bernstein [1] or a similar algorithm. But this algorithm is too slow for large constants. We will present a completely different algorithm, that is suitable to large constants. But first, a simpler algorithm and Bernstein's algorithm will be presented.

# 2 Formulation of the Problem

A positive odd integer $n$ is given. One looks for a sequence of positive integers $u_0, u_1, u_2, \ldots, u_q$ such that:

- $u_0 = 1$;
- for $i > 0$, $u_i = |s_i\, u_j + 2^{c_i}\, u_k|$, with $j < i$, $k < i$, $s_i \in \{-1, 0, 1\}$, $c_i \in \mathbb{N}$;
- $u_q = n$.

The problem is to find an algorithm that yields a minimal sequence $(u_i)_{0 \leq i \leq q}$. But this problem is very complex (it is believed to be NP-complete). So, we have to find heuristics.

Note: here, we restrict to positive integers. We could change the formulation to accept negative integers (i.e., remove the absolute value and allow the sign to be applied to either $u_j$ or $u_k$), but this would be an equivalent formulation.

# 3 The Binary Method

The simplest heuristic consists in writing the constant $n$ in binary and generating a shift and an add for each 1 in the binary expansion (e.g., starting from the left): for instance, consider $n = 113$, that is, we want to compute $113x$. In binary, $113 = 1110001_2$. We generate the following operations:

$$
\begin{array}{rcl}
3x & \leftarrow & (x << 1) + x \\
7x & \leftarrow & (3x << 1) + x \\
113x & \leftarrow & (7x << 4) + x
\end{array}
$$

The number of operations is the number of 1's in the binary expansion, minus 1.

This method can be improved using Booth's recoding, which consists in introducing signed digits ($-1$ denoted $\overline{1}$, 0 and 1) and performing the following transform:

$$\underbrace{1111\ldots1111}_{k \text{ digits}} \rightarrow 1\underbrace{0000\ldots000}_{k-1 \text{ digits}}\overline{1}.$$

This transform is based on the formula:

$$2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2^1 + 2^0 = 2^k - 1.$$

For instance, 11011 would be first transformed to $1110\overline{1}$, then to $100\overline{1}0\overline{1}$. Thus, Booth's recoding allows to decrease the number of non-zero digits.

With the above example: $113 = 100\overline{1}0001_2$. This gives 2 operations only:

$$
\begin{array}{rcl}
7x & \leftarrow & (x << 3) - x \\
113x & \leftarrow & (7x << 4) + x
\end{array}
$$

# 4 Bernstein's Algorithm

Bernstein's algorithm is based on arithmetic operations. It doesn't explicitely use the binary expansion of $n$. It consists in restricting the operations to $k = i-1$ and $j = 0$ or $i-1$ (in the formulation) and it can be used with different costs for the addition, the subtraction and the shifts. It is a *branch-and-bound* algorithm, with the following formulas:

$$
\begin{array}{lcl}
\mathrm{Cost}(1) & = & 0 \\
\mathrm{Cost}(n \text{ even}) & = & \mathrm{Cost}(n/2^{\,c} \text{ odd}) + \texttt{ShiftCost} \\
\mathrm{Cost}(n \text{ odd}) & = & \min \left\{ \begin{array}{l}
\mathrm{Cost}(n+1) + \texttt{SubCost} \\
\mathrm{Cost}(n-1) + \texttt{AddCost} \\
\mathrm{Cost}(n/(2^{\,c}+1)) + \texttt{ShiftCost} + \texttt{SubCost} \\
\mathrm{Cost}(n/(2^{\,c}-1)) + \texttt{ShiftCost} + \texttt{AddCost}
\end{array} \right.
\end{array}
$$

An advantage of Bernstein's algorithm is that there is no extra memory (registers or RAM) needed for temporary results, in the generated code. But extra memory is not always a problem.

# 5 A Pattern-based Algorithm

## 5.1 The Algorithm

This algorithm is based on the binary method: after Booth's recoding, we regard the number $n$ as a vector of signed digits 0, +1, −1, denoted `0`, `P` and `N` (and sometimes, 0, 1, $\bar{1}$). The idea (that is recursively applied) is as follows: we look for repeating (non necessarily adjacent) digit-patterns, to have the most digits `P` and `N` disappeared in one operation. To simplify, one only looks for patterns that repeat twice (though, in fact, they may repeat more often). For instance, $20061 = 100111001011101_2$, recoded to `P0P00N0P0N00N0P`, contains the pattern `P000000P0N` twice (the first one in the positive form and the second one in the negative form `N000000N0P`). Thus, considering this pattern allows to have 3 nonzero digits disappeared in one operation, and we now need to compute `P000000P0N` and the remaining `00P000000000000`. This can be summarized by:

```
  P000000P0N
-      P000000P0N
+ 00P000000000000
-----------------
  P0P00N0P0N00N0P
```

On this example, 4 operations are obtained (`P000000P0N` is computed with 2 operations thanks to the binary method), whereas Bernstein's algorithm generates 5 operations.

Now, it is important to find a good repeating pattern quickly enough. The number of nonzero digits of a pattern is called the *weight* of the pattern. We look for a pattern having a maximal weight. To do this, we take into account the fact that, in general, there are much fewer nonzero digits than zero digits, in particular near the leaves of the recursion tree, because of the following relation: $w(\text{parent}) = w(\text{child 1}) + 2\,w(\text{child 2})$. The solution is to compute all the possible distances between two nonzero digits, in distinguishing between identical digits and opposite digits. This gives an upper bound on the pattern weight associated with each distance. For instance, with `P0P00N0P0N00N0P`:

| distance | upper bound | weight |
|---|---|---|
| 2 (P-N / N-P) | 3 | 2 |
| 5 (P-N / N-P) | 3 | 3 |
| 7 (P-P / N-N) | 3 | 2 |

The distances are sorted according to the upper bounds, then they are tried the one after the other until the maximal weight and a corresponding pattern are found.

## 5.2 Comparison with Bernstein's Algorithm

This algorithm has been compared to Bernstein's and we found that on average, it is slightly better than Bernstein's for small constants. Comparisons couldn't be performed on large constants because Bernstein's algorithm would be too slow: the complexity of Bernstein's algorithm is exponential, whereas the pattern-based algorithm is polynomial (it seems to be in $O(n^3)$ on average: $O(n^2)$ for each recursion height).

If we consider the number of generated operations[1] by these algorithms for the numbers up to $2^{20}$, the largest difference is obtained for 543413:

With the pattern-based algorithm, one obtains:

$$\begin{array}{rrcl}
1. & 255x & \leftarrow & (x << 8) - x \\
2. & 3825x & \leftarrow & (255x << 4) - 255x \\
3. & 19125x & \leftarrow & (3825x << 2) + 3825x \\
4. & 543413x & \leftarrow & (x << 19) + 19125x
\end{array}$$

With Bernstein's algorithm, one obtains:

$$\begin{array}{rrcl}
1. & 9x & \leftarrow & (x << 3) + x \\
2. & 71x & \leftarrow & (9x << 3) - x \\
3. & 283x & \leftarrow & (71x << 2) - x \\
4. & 1415x & \leftarrow & (283x << 2) + 283x \\
5. & 11321x & \leftarrow & (1415x << 3) + x \\
6. & 33963x & \leftarrow & (11321x << 2) - 11321x \\
7. & 135853x & \leftarrow & (33963x << 2) + x \\
8. & 543413x & \leftarrow & (135853x << 2) + x
\end{array}$$

## 5.3 Results on random numbers

An implementation of the algorithm has been tested on random numbers (an exhaustive test would have been too slow). Here is the average number of generated operations as a function of the number of bits (the first and the last bits must be 1):

| | |
|---|---|
| 32 | 8.0 |
| 64 | 14.5 |
| 128 | 26.3 |
| 256 | 47.6 |
| 512 | 86.5 |
| 1024 | 157.4 |
| 2048 | 289.4 |

The ratio between two consecutive numbers is almost a constant. From this results, we can conjecture that the average number of operations generated for an $n$-bit integer is $O(n^k)$, where $k \approx 0.85$.

## 5.4 Possible Improvements

Our algorithm can still be improved. Here are some ideas, which have not been implemented yet:

[1] An operation is a shift, then an addition or a subtraction, i.e., the value $q$ in the formulation.

- One can look for common digit-patterns. For instance, consider `PONONOOPONONOOOPON`, with pattern `PONON`. `PON` appears both in the pattern and in the remaining digits; thus, it needs to be computed only once (under some conditions). A solution is to stop the recursion when the maximal weight is equal to 1 (here, only the binary method can be used); looking for common patterns would be easier. Note that common patterns should be looked for before using the binary method: with the above example, if we start with `NON` in `PONON`, the common pattern `PON` cannot be used; we need to start with `PON` in `PONON`.

- Sometimes, there are several choices that correspond to the maximal weight. Instead of taking only one, one can try several patterns, and keep the shortest operation sequence.

- One can consider the following transform, which does not change the weight: `PON` $\leftrightarrow$ `OPP` (and `NOP` $\leftrightarrow$ `ONN`). For instance, $11010101001_2$ has the default code `PONOPOPOPOOP`, but the equivalent code `POONONNOPOOP` is better (with the pattern `POONOON`). As the number of equivalent codes is exponential, we cannot test all of them; so, we have to look for a method to find the best transforms.

- Instead of defining a pattern of maximal weight that appears twice, one can define a new digit consisting of two old nonzero digits. For instance, consider $10\bar{1}0\bar{1}0010\bar{1}0\bar{1}00010\bar{1}$ and the pattern $10\bar{1}0\bar{1}$. One defines a new digit: $\mathrm{A} = 10000001$, and obtains: $\mathrm{A0\bar{A}0\bar{A}00010\bar{1}}$. Then, one defines $\mathrm{B} = \mathrm{A00000001}$, and finally obtains: $\mathrm{\bar{A}000B0\bar{B}}$. This leads to 4 operations, like the common-pattern method.

## 6 Conclusion

Thanks to the algorithm presented here, we will be able to perform fast multiplications by integer constants, which may have several hundred bits. Future work will consist in improving this algorithm, doing some experiments to find the complexity, and trying to prove some results.

## References


[1] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[2] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1973.

[3] D. Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.