Backtracking-Assisted Multiplication

Houda Ferradi, Rémi Géraud, Diana Maimuț,
David Naccache, and Hang Zhou

École normale supérieure
Département d'Informatique
45 rue d'Ulm, F-75230, Paris CEDEX 05, France
{surname.name}@ens.fr

Abstract. This paper describes a new multiplication algorithm, particularly suited to lightweight microprocessors when one of the operands is known in advance. The method uses backtracking to find a multiplication-friendly encoding of the operand known in advance.
A 68HC05 microprocessor implementation shows that the new algorithm indeed yields a twofold speed improvement over classical multiplication for 128-byte numbers.

1 Introduction

A number of applications require performing long multiplications in performance-restricted environments. Indeed, low-end devices such as the 68HC05 or the 80C51 microprocessors have a very limited instruction-set, very limited memory, and operations such as multiplication are rather slow: a `mul` instruction typically claims 10 to 20 cycles.

General multiplication has been studied extensively, and there exist algorithms with very good asymptotic complexity such as the Schönhage-Strassen algorithm [18] which runs in time $O(n \log n \log\log n)$ or the more recent Fürer algorithm [13], some variants of which achieve the slightly better $O(2^{3 \log^* n} n \log n)$ complexity [14]. Such algorithms are interesting when dealing with extremely large integers, where these asymptotics prove faster than more naive approaches.

In many cryptographic contexts however, multiplication is performed between a variable and a *pre-determined constant*:

- During Diffie-Hellman key exchange [9] or El-Gamal [10] a constant g must be repeatedly multiplied by itself to compute $g^x \bmod p$.
- The essential computational effort of a Fiat-Shamir prover [11, 12] is the multiplication of a subset of fixed keys (denoted s_i in [11]).
- A number of modular reduction algorithms use as a building-block multiplications (in $\mathbb{N}$) by a constant depending on the modulus. This

is for instance the case of Barrett's algorithm [2] or Montgomery's algorithm [17].

The main strategy to exploit the fact that one operand is constant consists in finding a decomposition of the multiplication into simpler operations (additions, subtractions, bitshifts) that are hardware-friendly [3]. The problem of finding the decomposition with the least number of operations is known as "single constant multiplication" (SCM) problem. SCM $\in$ **NP**-complete as shown in [4], even if fairly good approaches exist [1, 7, 8, 20] for small numbers. For larger numbers, performance is unsatisfactory unless the constant operand has a predetermined format allowing for *ad hoc* simplifications.

In this paper, we propose a completely different approach: the constant operand is encoded in a computation-friendly way, which makes multiplication faster. This encoding is based on linear relationships detected amongst the constant's digits (or, more generally, subwords), and can be performed offline in a reasonable time for 1024-bit numbers and 8-bit microprocessors. We use a graph-based backtracking algorithm [16] to discover these linear relationships, using recursion to keep the encoder as short and simple as possible.

2 Multiplication Algorithms

We now provide a short overview of popular multiplication methods. This summary will serve as a baseline to evaluate the new algorithm's performance.

Multiplication algorithms usually fall in two broad categories: general divide-and-conquer algorithms such as Toom-Cook [6, 19] and Karatsuba [15]; and the generation of integer multiplications by compilers, where one of the arguments is statically known. We are interested in the case where small-scale optimizations such as Bernstein's [3] are impractical, but general purpose multiplication algorithms à la Toom-Cook are not yet interesting.

Throughout the paper we will assume unsigned integers, and denote by w the word size (typically, $w = 8$), a_i, b_i and r_i the binary digits of a, b and r respectively:

$$a = \sum_{i=0}^{n-1} 2^{wi} a_i, \quad b = \sum_{i=0}^{n-1} 2^{wi} b_i, \quad \text{and} \quad r = a \times b = \sum_{i=0}^{2n-1} 2^{wi} r_i.$$

2.1 Textbook Multiplication

A direct way to implement long multiplication consists in extending textbook multiplication to several words. This is often done by using a MAD[1] routine.

A MAD routine takes as input four n-bit words $\{x, y, c, \rho\}$, and returns the two n-bit words c', ρ' such that $2^n c' + \rho' = x \times y + c + \rho$. We write

$$\{c', \rho'\} \leftarrow \text{MAD}(x, y, c, \rho).$$

If such a routine is available then multiplication can be performed in n^2 MAD calls using Algorithm 1. The MIRACL big number library [5] provides such a functionality.

Algorithm 1: MAD-based computation of $r = a \times b$.

Input: $a, b \in \mathbb{N}$.
Output: $r \in \mathbb{N}$ such that $r = a \times b$.

```
for i ← 0 to 2n − 1 do
    r_i ← 0
end for
for i ← 0 to n − 1 do
    c ← 0
    for j ← 0 to n − 1 do
        {c, r_{i+j}} ← MAD(a_i, b_j, c, r_{i+j})
    end for
    r_{i+n} ← c
end for
return r
```

This approach is unsatisfactory: it performs more computation than often needed. Assuming a constant-time MAD instruction, Algorithm 1 runs in time $O(n^2)$.

2.2 Karatsuba's Algorithm

Karatsuba [15] proposed an ingenious divide-and-conquer multiplication algorithm, where the operands a and b are split as follows:

$$r = a \times b = (2^L \bar{a} + \underline{a}) \times (2^L \bar{b} + \underline{b}),$$

[1] An acronym standing for "Multiply Add Divide"

where typically $L = nw/2$. Instead of computing a multiplication between long integers, Karatsuba performs multiplications between shorter integers, and (virtually costless) multiplication by powers of 2. Karatsuba's algorithm is described in Algorithm 2.

Algorithm 2: Karatsuba's algorithm to compute $r = a \times b$.

Input: $a, b \in \mathbb{Z}$.
Output: $r \in \mathbb{Z}$ such that $r = a \times b$.

1 $u = \bar{a} \times \bar{b}$
2 $v = \underline{a} \times \underline{b}$
3 $w = (\bar{a} + \underline{a})(\bar{b} + \underline{b}) - u - v$
4 $r = 2^{2L} \times u + 2^{L} \times w + v$
5 **return** r

This approach is much faster than naive multiplication – on which it still relies for multiplication between short integers – and runs[2] in $\Theta(n^{\log_2 3})$.

2.3 Bernstein's Multiplication Algorithm

When one of the operands is constant, different ways to optimize multiplication exist. Bernstein [3] provides a branch-and-bound algorithm based on a cost function.

The minimal cost, and an associated sequence, are found by exploring a tree, possibly using memoization to avoid redundant searches. More elaborate pruning heuristics exist to further speedup searching. The minimal cost path produces a list of operations which provide the result of multiplication.

Because of its exponential complexity, Bernstein's algorithm is quickly overwhelmed when dealing with large integers. It is however often implemented by compilers for short (32 to 64-bit) constants.

3 The Proposed Algorithm

3.1 Intuitive Idea

The proposed algorithm works with an alternative representation of the constant operand a. Namely, we wish to express some a_i as a linear com-

[2] When repeated recursively.

bination of other a_js with small coefficients. It is then easy to reconstruct the whole multiplication $b \times a$ from the values of the $b \times a_j$ only.

The more linear combinations we can find, the less multiplications we need to perform. Our algorithm therefore tries to find the longest sequence of linear relationships between the digits of a. We call this sequence's length the *coverage* of a.

Yet another performance parameter is the number of registers used by the multiplier. Ideally at any point in time two registers holding intermediate values should be used. This is not always possible and depends on the digits of a.

As an example, consider the set of relations of Table 1. All words are expressed ultimately in terms of the values of a_3 and a_7. In Table 1, we express a as a subset of words $A \in \{a_0, \ldots, a_{n-1}\}$ and build a sparse table U where $U_{i,j} \in \{-1, 0, 1, 2, =\}$, which encodes linear relationships between individual words. During multiplication, U describes how the different a_i can be derived from each other.

Table 1. An example showing how linear relationships between individual words are encoded and interpreted.

step	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	meaning	reg1	reg2	reg3
0				1		=		1			$a_5 \leftarrow a_3 + a_7$	a_5	a_3	a_7
1					=	2					$a_4 \leftarrow a_5 + a_5$	a_5	a_4	a_7
2	=				-1			1			$a_0 \leftarrow a_7 - a_4$	a_0	a_4	a_7
3					2					=	$a_9 \leftarrow a_4 + a_4$	a_0	a_4	a_9
4	1	=								-1	$a_1 \leftarrow a_0 - a_9$	a_0	a_1	a_9
5		1							=	1	$a_8 \leftarrow a_1 + a_9$	a_8	a_1	a_9
6		1	=					1			$a_2 \leftarrow a_1 + a_8$	a_8	a_1	a_2
7			2				=				$a_6 \leftarrow a_2 + a_2$	a_8	a_6	a_2

Hence it suffices to compute $b \times a_3$ and $b \times a_7$ to infer all other $b \times a_i$ by long integer additions. Note that the algorithm only needs to allocate three $(n+1)$-word registers `reg1`, `reg2` and `reg3` to store intermediate results.

The values allowed in U can easily be extended to include more complex relationships (larger coefficients, more variables, etc.) but this immediately impacts the algorithm's performance. Indeed, the corresponding search graph has correspondingly many more branches at each node.

Operations can be performed without overflowing (*i.e.* so that results fit in a word), or modulo the word size. In the latter case, it is necessary

to subtract $b \ll w$ from the result, where w is the word size, to obtain the correct result. This incurs some additional cost.

3.2 Backtracking Algorithm

Algorithm 3: macro $\mathsf{Step}(u, w)$.

```
1 (p_{d+1,0}, p_{d+1,1}) ← (u, w)
2 Backtrack(d + 1)
```

Algorithm 4: macro $\mathsf{EncodeDep}(c, \text{opcode})$.

```
1 if c < 256 then
2 |   if v_c = False then
3 |   |   (v_c, p_{d,2}, p_{d,3}) ← (True, c, opcode)
4 |   |   Step(a, b)
5 |   |   Step(a, c)
6 |   |   Step(b, c)
7 |   |   v_c ← False
8 |   end if
9 end if
```

Algorithm 5: macro $\mathsf{Backtrack}(d)$.

```
1 if d > d^max then
2 |   (d^max, p^max) ← (d, p)
3 end if
4 (a, b) ← (p_{d,0}, p_{d,1})
5 for opcode ∈ C do
6 |   EncodeDep(opcode(a, b), opcode)
7 end for
```

Linear combinations amongst words of a are found by backtracking [16], the pseudocode of which is given in Algorithm 5. Our implementation focuses on linear dependencies amongst 8-bit words, as our main recommendation for applying the proposed multiplication algorithm is exactly an 8-bit microprocessor.

Algorithm 6: Main Backtracking Program.

Input: $A = \sum_{i=0}^{N-1} 256^i A_i$.
Output: $U_{i,j}$

```
// Initialization
for i = 0 to 255 do
    v_i ← True
end for
for i = 0 to N − 1 do
    v_{A_i} ← False
end for
d^max ← −1
// Backtracking
for i = 0 to 255 do
    for j = i + 1 to 255 do
        if v_i = v_j = False then
            (p_{0,0}, p_{0,1}, v_i, v_j) ← (i, j, True, True)
            Backtrack(0)
            (v_i, v_j) ← (False, False)
        end if
    end for
end for
// U-matrix reconstruction
U_{i,j} ← 0
for i = 0 to 255 do
    (in_1, in_2, out, opcode) ← p_i^max
    (U_{i,in_1}, U_{i,in_2}, U_{i,out}) ← (1, 1, opcode)
end for
return U_{i,j}
```

We take advantage of recursion and macro expansion (see Algorithms 3 to 5) to achieve a more compact code. In this implementation, p encodes the current depth's three registers of Table 1 as well as the current operation. With suitable listing, Algorithm 6 outputs a set of values being related, along with the corresponding relation. The dependencies that we take into account in our C code (given in Appendix A) don't go beyond depth 2. Thus, the corresponding operations are $\mathcal{C} = \{+, -, \times 2\}$. We also add these operations performed modulo 256, to obtain more solutions. The alternative to this approach is to consider a bigger depth, which naturally leads to more possibilities.

Our program takes as an input an integer p that represents the percentage of a being covered (*i.e.* the coverage is $p/100$ times the length of the a). In a typical lightweight scenario, a 128-byte number is involved in the multiplication process. Our software attempts to backtrack over

a coverage-related number of values out of 256. It follows immediately that at most a 50% coverage would be required for performing such a multiplication (as byte collisions are likely to happen).

The program takes as parameter the list of bytes of a. If some bytes appear multiple times, it is not necessary to re-generate each of them individually: generation is performed once, and the value is cloned and dispatched where needed.

Note that if precomputation takes too long, the list of a_i can be partitioned into several sub-lists on which backtrackings are run independently. This would entail as many initial multiplications by the online multiplier but still yield appreciable speed-ups.

3.3 Multiplication Algorithm

Algorithm 7: Virtual Machine

Input: $b, \text{instr} = (\text{opcode}, i, j, t, p)_k, R$
Output: r

$r \leftarrow 0$
foreach $(i, v) \in R$ **do**
    $\texttt{reg}[i] \leftarrow v \times b$
    $\mathsf{PlaceAt}(v, \texttt{reg}[i])$
end foreach
foreach $(\text{opcode}, i, j, t, p) \in \text{instr}$ **do**
    $\texttt{reg}[t] \leftarrow \text{opcode}(\texttt{reg}[i], \texttt{reg}[j])$
    $\mathsf{PlaceAt}(p, t)$
end foreach
return r

With the encoding of a generated by Algorithm 6, it is now possible to implement multiplication efficiently.

To that end we make use of a specific-purpose multiplication virtual machine (VM) described in Algorithm 7. The VM is provided with instructions of the form

```
opcode t, i, j, p
```

that are extracted offline from U. Here, `opcode` is the operation to perform, `i` and `j` are the indices of the operands, `t` is the index of the result, and $\texttt{p} \leftarrow w \times \texttt{t}$ is the position in r where to place the result, w being the word size. The value of `p` is pre-computed offline to allow for a more efficient implementation.

We store the result in a $2n$-byte register initialized with zero. We also make use of a long addition procedure $\mathsf{PlaceAt}(\mathtt{p}, i)$ which "places" the contents of the $(n+1)$-byte register $\mathtt{reg}[i]$ at position $\mathtt{p}$ in r. PlaceAt performs the addition of register $\mathtt{reg}[i]$ starting from an offset $\mathtt{p}$ in r, propagating the carry as needed.

Finally, we assume that the list $R = (i, v)_k$ of root nodes (position and value) of U is provided.

After executing all the operations described in U, Algorithm 7 has computed $r \leftarrow a \times b$.

Remark 1 (Karatsuba Multiplication).

Using the notations of Algorithm 2 one can see that in settings where a is a constant, the numbers u, v, w all result from the multiplication of $\bar{b}, \underline{b}$ and $\bar{b} + \underline{b}$ (which are variable) by $\bar{a}, \underline{a}$ and $\bar{a} + \underline{a}$ (which are constant). Hence our approach can independently be combined with Karatsuba's algorithm to yield further improvements.

4 Performance

The algorithm has an offline step (backtracking) and an online step (multiplication), which are implemented on different devices.

The offline step is naturally the longest; its performance is heavily dependent on the digit combination operations allowed and on how many numbers are being dealt with. More precisely, results are near-instant when dealing with 64 individual bytes and operations $\{+, -, \times 2\}$. It takes much longer if operations modulo 256 are considered as well, but this gives a better coverage of a, hence better *online* results. That being said, modulo 256 operations are slightly less efficient than operations over the integers ($\simeq 1.5$ more costly), since they require a subtraction of b afterwards.

Table 2 provides comparative performance data for a multiplication by the processed constant $\lfloor \pi 2^{1024} \rfloor$. Backtracking this constant took 85 days on an Altix UV1000 cluster.

Table 2. Performance on a 68HC05 clocked at 5 MHz

	Time	RAM	Code Size
Usual Algorithm	188 ms	395 bytes	1.1 kilobytes
New Algorithm	72 ms	663 bytes	1.7 kilobytes

As a final remark, note that one can also reverse the idea and generate a key by which multiplication is easy. This can be done by progressively

picking VM operations until an operand (key) with sufficient entropy is obtained. While this is not equivalent to randomly selecting keys, the authors conjecture that, in practice, the existence of linear relations[3] between key bytes should not significantly weaken public-key implementations.

References

1. Avižienis, A.: Signed-digit number representations for fast parallel arithmetic. IRE Transactions on Electronic Computers (3), 389–400 (1961)
2. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology - CRYPTO'86, Santa Barbara, California, USA. pp. 311–323 (1986)
3. Bernstein, R.: Multiplication by integer constants. Software: Practice and Experience 16(7), 641–652 (1986)
4. Cappello, P., Steiglitz, K.: Some complexity issues in digital signal processing. IEEE Transactions on Acoustics, Speech and Signal Processing 32(5), 1037–1041 (1984)
5. Certivox: The MIRACL big number library, see https://www.certivox.com/miracl
6. Cook, S.A.: On the minimum computation time of functions. Ph.D. thesis (1966)
7. Dempster, A.G., Macleod, M.D.: Constant integer multiplication using minimum adders. IEE Proceedings – Circuits, Devices and Systems 141(5), 407–413 (1994)
8. Dempster, A.G., Macleod, M.D.: Use of minimum-adder multiplier blocks in FIR digital filters. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 42(9), 569–577 (1995)
9. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
10. El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Advances in Cryptology – CRYPTO'84, Santa Barbara, California, USA. pp. 10–18 (1984)
11. Feige, U., Fiat, A., Shamir, A.: Zero knowledge proofs of identity. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, USA. pp. 210–217 (1987)
12. Feige, U., Fiat, A., Shamir, A.: Zero-knowledge proofs of identity. J. Cryptology 1(2), 77–94 (1988)
13. Fürer, M.: Faster integer multiplication. SIAM Journal on Computing 39(3), 979–1005 (2009)
14. Harvey, D., Van Der Hoeven, J., Lecerf, G.: Even faster integer multiplication. arXiv preprint arXiv:1407.3360 (2014)
15. Karastuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Doklady Akad. Nauk SSSR 145(293-294) (1962)
16. Knuth, D.: The Art of Computer Programming (1968)
17. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation 44(170), 519–521 (1985)
18. Schönhage, D.D.A., Strassen, V.: Schnelle Multiplikation grosser Zahlen. Computing 7(3-4), 281–292 (1971)
19. Toom, A.L.: The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. Soviet Mathematics Doklady 3, 714–716 (1963)

[3] These linear relations are unknown to the attacker.

20. Wu, H., Hasan, M.A.: Closed-form expression for the average weight of signed-digit representations. IEEE Transactions on Computers 48(8), 848–851 (1999)

A Source Code

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

int False = 0;
int True = 255;

#define step(u,v) path[dep+1][0]=u; path[dep+1][1]=v; backtracking(dep+1);
#define steps(x,n) c=x; if(c<256) {if (visited[c]==False) {visited[c]=True;\
        path[dep][2]=c;path[dep][3]=n;step(a,b);step(a,c);step(b,c);visited[c]=False;}}

int visited[2 * 256];
int maxdep,path[256][4],maxpath[256][4];
int size;

void backtracking(int dep){

    if (dep > maxdep){
        maxdep = dep;
        memcpy(maxpath, path, sizeof(path));
    }

    int a = path[dep][0];
    int b = path[dep][1];
    int c;

    steps(a+b,         1);
    steps(a<<1,        2);
    steps(b<<1,        3);
    steps((a<<1)%256, 4);
    steps((b<<1)%256, 5);
    steps((a+b)%256,  6);
    steps(abs(a-b),   7);
}

int main() {
    int i,j;
    FILE *Fin, *Fout;
    int *A;

    Fin = fopen("input.txt", "r");
    fscanf(Fin, "%d", &size);
    A = (int*)malloc(sizeof(int)*size);
    for (i=0; i <size; ++i) {
        fscanf(Fin, "%d", &A[i]);
    }

    for (i=0; i<256; ++i)  visited[i] = True;
    for (i=0; i<size; ++i) visited[A[i]] = False;
```

```
    free(A);

    maxdep = -1;
    for (i=0; i<256; ++i) {
        for (j=i+1; j<256; ++j) {
            if (visited[i]==True || visited[j]==True) continue;
            path[0][0] = i;
            path[0][1] = j;
            visited[i] = visited[j] = True;
            backtracking(0);
            visited[i] = visited[j] = False;
        }
    }

    Fout = fopen("output.txt","w");
    for (i=0; i < maxdep; ++i)
        fprintf(Fout,"%3d %3d %3d %3d\n",
                maxpath[i][0],  // a_i
                maxpath[i][1],  // a_j
                maxpath[i][2],  // a_p
                maxpath[i][3]); // op

    return EXIT_SUCCESS;
}
```