Missing a trick: Karatsuba revisited

Michael Scott

MIRACL Labs
mike.scott@miracl.com
January 1st, 2016

Abstract. There are a variety of ways of applying the Karatsuba idea to multi-digit multiplication. These apply particularly well in the context where digits do not use the full word-length of the computer, so that partial products can be safely accumulated without fear of overflow. Here we re-visit the "arbitrary degree" version of Karatsuba and show that the cost of this little-known variant has been over-estimated in the past. We also attempt to definitively answer the question as to the cross-over point where Karatsuba performs better than the classic method.

1 Introduction

As is well known the Karatsuba idea for calculating the product of two polynomials can be used recursively to significantly reduce the number of partial products required in a long multiplication calculation, at the cost of increasing the number of additions. A one-level application can save 1/4 of the partial products, and a two-level application can save 7/16ths etc. However application of Karatsuba in this way is quite awkward, it attracts a large overhead of extra additions, and the ideal recursion is only available if the number of digits is an exact power of two.

One way to make Karatsuba more competitive is to use a number base radix that is somewhat less than the full register size, so that additions can be accumulated without overflow, and without requiring immediate carry propagation. Here we refer to this as a "reduced-radix" representation.

Multi-precision numbers are represented as an array of computer words, or "limbs", each limb representing a digit of the number. Each computer word is typically the same size as the processor's registers, that are manipulated by its instruction set. Using a full computer word for each digit, or a "packed-radix" representation [7], intuitively seems to be optimal, and was the method originally adopted by most multi-precision libraries. However to be efficient this virtually mandates an assembly language implementation to handle the flags that catch the overflows that can arise, for example, from the simple act of adding two digits.

The idea of using a reduced-radix representation of multi-precision numbers (independent of its suitability for Karatsuba) has long been championed by Bernstein and his co-workers. See for example [2] for a discussion of the relative merits of packed-radix and reduced-radix representation. This approach is supported

by the recent experience of Hamburg in his implementation of the Goldilocks elliptic curve [7]. A reduced-radix representation is sometimes considered to be more efficient [7] – and this is despite the fact that in many cases it will require an increased number of limbs.

In [1] is described an elliptic curve implementation that uses this technique to demonstrate the superiority of using the Karatsuba idea in a context where each curve coordinate is represented in just 16 32-bit limbs. In fact there is much confusion in the literature as to the break-even point where Karatsuba becomes superior. One confounding factor is that whereas Karatsuba trades multiplications for additions, in modern processors multiplications may be almost as fast as additions.

Elliptic curve sizes have traditionally been chosen to be multiples of 128-bits, to provide a nice match to standard levels of security. On the face of it, this is fortuitous, as for example a 256-bit curve might have its x and y coordinates fit snugly inside of 4 64-bit or 8 32-bit computer words using a packed-radix representation. Again, as these are exact powers of 2, they might be thought of as being particularly suitable for the application of the Karatsuba idea. However somewhat counter-intuitively this is not the case – if Karatsuba is to be competitive the actual number base must be a few bits less than the word size in order to facilitate addition of elements without carry processing (and to support the ability to distinguish positive and negative numbers). So in fact a competitive implementation would typically require 5 64-bit or 9 32-bit words, where 5 and 9 are not ideal polynomial degrees for the application of traditional Karatsuba.

What is less well known is that there is an easy to use "arbitrary degree" variant of Karatsuba (ADK), as it is called by Weimerskirch and Paar [11], which can save nearly $1/2$ of the partial products and where the polynomial degree is of no concern to the implementor. In fact this idea has an interesting history. An earlier draft of the Weimerskirch and Paar paper from 2003 is referenced in [9]. But it appears to have been discovered even earlier by Khachatrian et al. [8], and independently by David Harvey, as reported in Exercise 1.4 in the textbook [3]. Essentially the same idea was used by Granger and Scott [4], building on earlier work from Granger and Moss [5] and Nogami et al. [10], in the context of a particular form of modular arithmetic.

Here we consider the application of this variant to the problem of long integer multiplication. Since the number of partial products required is the same as that required by the well known squaring algorithm, squaring is not improved, and so is not considered further here. We restrict our attention to the multiplication of two equal sized numbers, as arises when implementing modular arithmetic as required by common cryptographic implementations.

2 The ADK algorithm

This algorithm is described in mathematical terms in [3], [8] and [11]. However here we have used the subtractive variant of Karatsuba to some advantage to get a simpler formula, as pointed out to us by [12].

$$\boldsymbol{xy} = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1}(x_i - x_j)(y_j - y_i)b^{i+j} + \sum_{i=0}^{n-1} b^i \sum_{j=0}^{n-1} x_j y_j b^j$$

In fact we find the mathematical description unhelpful in that it makes the method look more complex than it is. It also makes it difficult to determine its exact complexity. To that end an algorithmic description is more helpful.

Algorithm 1 The ADK algorithm for long multiplication

INPUT: Degree n, and radix $b = 2^t$
INPUT: $\boldsymbol{x} = [x_0, \ldots, x_{n-1}], \boldsymbol{y} = [y_0, \ldots, y_{n-1}]$ where $x_i, y_i \in [0, b-1]$
OUTPUT: $\boldsymbol{z} = [z_0, \ldots, z_{2n-2}]$, where $z_i \in [0, b^2 - 1]$ and $\boldsymbol{z} = \boldsymbol{xy}$

```
1: function ADKMUL(x, y)
2:     for i ← 0 to n − 1 do
3:         d_i ← x_i y_i
4:     end for
5:     s ← d_0
6:     z_0 ← s
7:     for k ← 1 to n − 1 do
8:         s ← s + d_k
9:         t ← s
10:        for i ← 1 + ⌊k/2⌋ to k do
11:            t ← t + (x_i − x_{k−i})(y_{k−i} − y_i)
12:        end for
13:        z_k ← t
14:    end for
15:    for k ← n to 2n − 2 do
16:        s ← s − d_{k−n}
17:        t ← s
18:        for i ← 1 + ⌊k/2⌋ to n − 1 do
19:            t ← t + (x_i − x_{k−i})(y_{k−i} − y_i)
20:        end for
21:        z_k ← t
22:    end for
23:    return z
24: end function
```

The number of multiplications and additions required can be confirmed by a simple counting exercise. For clarity we have not included the final carry propagation, which reduces the product z to a radix b representation. A fully unrolled example of the algorithm in action for the case $n = 4$ is given in the next section.

3 Comparing Karatsuba variants

As an easy introduction consider the product of two 4 digit numbers, $z = xy$. The School-boy method (SB) requires 16 multiplications (muls) and 9 double precision adds, which is equivalent to 18 single precision adds. In the sequel when comparing calculation costs a "mul" M is a register-sized signed multiplication resulting in a double register product. An "add" A is the addition (or subtraction) of two registers. We also make the reasonable assumption that while add, shift or masking instructions cost the same on the target processor, an integer multiply instruction may cost more. So the cost of the SB method here is 16M+18A.

$$\begin{aligned}
z_0 &= x_0y_0 \\
z_1 &= x_1y_0 + x_0y_1 \\
z_2 &= x_2y_0 + x_1y_1 + x_0y_2 \\
z_3 &= x_3y_0 + x_2y_1 + x_1y_2 + x_0y_3 \\
z_4 &= x_3y_1 + x_2y_2 + x_1y_3 \\
z_5 &= x_3y_2 + x_2y_3 \\
z_6 &= x_3y_3
\end{aligned} \tag{1}$$

A final "propagation" of carries is also required. Assuming that the number base is a simple power of 2, this involves a single precision masking followed by a double precision shift applied to each digit of the result. The carry must then be added to the next digit. If multiplying two n digit numbers the extra cost is equivalent to $(10n - 7)A$ adds. Here we will neglect this extra contribution, as it applies independent of the method used for long multiplication.

Using arbitrary-degree Karatsuba (or ADK), the same calculation takes 10 muls and 11 double precision adds and 12 single precision subs. The total cost is 10M+34A. So overall 6 muls are saved at the cost of 16 adds

$$\begin{aligned}
z_0 &= x_0y_0 \\
z_1 &= (x_1 - x_0)(y_0 - y_1) + (x_0y_0 + x_1y_1) \\
z_2 &= (x_2 - x_0)(y_0 - y_2) + [x_0y_0 + x_1y_1] + x_2y_2 \\
z_3 &= (x_3 - x_0)(y_0 - y_3) + (x_2 - x_1)(y_1 - y_2) + [x_0y_0 + x_1y_1] + [x_2y_2 + x_3y_3] \\
z_4 &= (x_3 - x_1)(y_1 - y_3) + x_1y_1 + [x_2y_2 + x_3y_3] \\
z_5 &= (x_3 - x_2)(y_2 - y_3) + (x_2y_2 + x_3y_3) \\
z_6 &= x_3y_3
\end{aligned} \tag{2}$$

Here square brackets indicate values already available from the calculation. Hopefully the reader can see the pattern in this example in order to easily extrapolate to higher degree multiplications.

It is an interesting exercise to repeat this calculation using one level of "regular" Karatsuba, and simplifying the result. As can be seen in equation 3 the

same calculation requires 12 muls, 10 double precision adds and 4 single precision subs, or equivalently 12M+24A, so 4 muls are saved at the cost of 6 adds.

$$
\begin{aligned}
z_0 &= x_0y_0 \\
z_1 &= (x_1y_0 + x_0y_1) \\
z_2 &= (x_2 - x_0)(y_0 - y_2) + x_0y_0 + (x_1y_1 + x_2y_2) \\
z_3 &= [(x_2 - x_0)](y_1 - y_3) + (x_3 - x_1)[(y_0 - y_2)] + [x_1y_0 + x_0y_1] + [x_3y_2 + x_2y_3] \\
z_4 &= [(x_3 - x_1)][(y_1 - y_3)] + [x_1y_1 + x_2y_2] + x_3y_3 \\
z_5 &= (x_3y_2 + x_2y_3) \\
z_6 &= x_3y_3
\end{aligned}
\tag{3}
$$

Observe that only z_1, z_3 and z_5 are calculated differently. Using two levels of Karatsuba (equation 4), requires 9M+38A, so 7 muls are saved at the cost of 20 adds.

$$
\begin{aligned}
z_0 &= x_0y_0 \\
z_1 &= (x_1 - x_0)(y_0 - y_1) + (x_0y_0 + x_1y_1) \\
z_2 &= (x_2 - x_0)(y_0 - y_2) + [x_0y_0 + x_1y_1] + x_2y_2 \\
z_3 &= ([x_3 - x_1] - [x_2 - x_0])([y_0 - y_2] - [y_1 - y_3]) + [(x_2 - x_0)(y_0 - y_2)] + [(x_3 - x_1)(y_1 - y_3)] \\
&\quad + [(x_1 - x_0)(y_0 - y_1)] + [(x_3 - x_2)(y_2 - y_3)] + [x_0y_0 + x_1y_1] + [x_2y_2 + x_3y_3] \\
z_4 &= (x_3 - x_1)(y_1 - y_3) + x_1y_1 + [x_2y_2 + x_3y_3] \\
z_5 &= (x_3 - x_2)(y_2 - y_3) + (x_2y_2 + x_3y_3) \\
z_6 &= x_3y_3
\end{aligned}
\tag{4}
$$

Now only z_3 is calculated differently from the ADK approach. It is noteworthy that in [1] the authors deployed two levels of standard Karatsuba, and apparently did not consider the ADK method. However since the ADK approach works on a digit-by-digit basis, and thus applies seemlessly independent of the number of digits, it would appear to offer a nice easily applied compromise solution that extracts a big part of the Karatsuba advantage, without causing an explosion in the number of additions.

In terms of the number of partial products required, its performance is always at least as good as that obtained by applying one level of regular Karatsuba. This may represent an easily achieved "sweet spot" of relevance to applications involving medium sized numbers, as may for example apply in the context of Elliptic Curve Cryptography.

In passing we observe, as also noted in [3], that the ADK method can be used as an amusing alternative algorithm for pencil-and-paper long multiplication. We would not be surprised to learn of its use in the recreational mathematics literature.

4 Numerical stability

Before proceeding we need to address the problem of numerical stability. We start by assuming that both numbers to be multiplied are fully normalized, that is each digit of x is in the range $0 \leq x_i < b$. If they are not, they can be quickly normalized using a fast mask and shift operation (which works even if some of the digits are temporarily negative). For numerical stability of the long multiplication it is important that the sum of double-precision products that form each row of equation 1, do not cause a signed integer overflow. Assume that $b = 2^t$ where $t < w$ on a w-bit wordlength computer. Then the product of two such numbers could be as big as $2^{2t} - 2^{t+1} + 1$. The longest row consists of n such numbers, plus a carry from the previous row. So each row could not be larger than $(n+1)(2^{2t} - 2^{t+1} + 1)$. Since it must be possible to distinguish the sign of each partial product this must be strictly less than 2^{2w-1}.

For the common wordlengths of $w = 32$ and $w = 64$-bits, and for numbers of the sizes relevant to elliptic curve cryptography, we would expect t to be 28 or 29 on a 32-bit computer, and 61 or 62 for a 64-bit computer. Too large and the stability criteria will not be met. Too small and too many words will be required to represent our numbers, with a loss of efficiency.

We would assume that normally the largest radix possible would be used that is compatible with this stability condition. However there may be other factors at play which might dictate a slightly smaller choice for t – for example if reduction were merged with multiplication [4], or if it were regarded as desirable that field elements could be added without normalization prior to multiplication [7].

5 The true cost of ADK multiplication

In [11] the number of multiplications and additions required for the application of the ADK method is calculated, in the context of polynomial arithmetic. There it is worked out that its performance compared to the school-boy method, given a multiplication to addition cost ratio of r, is such that they are equivalent for $r = 3$ irrespective of the degree of the polynomials. The rather neat conclusion might be that unless multiplication takes more than 3 times as long as an addition, the method brings no advantage. And for many real-world processors with hardware support for integer multiplication, this may not be the case.

However here we are interested in multi-precision arithmetic, which is a little different. While all of the additions in the SB method are double precision, the subtractions required by the ADK method are only single precision. Furthermore the cost function used for ADK in [11] appears to be incorrect. The true cost in terms of single precision additions is actually only $2n^2 + 2n - 6$ for the ADK method, compared to the $2n^2 - 4n + 2$ required by SB. In [11] the number of additions is calculated as being of the order of $2.5n^2$. This dramatically changes the balance between the two contenders. Recall that for SB the number of muls is n^2, while for the ADK method it is $n(n+1)/2$. An immediate and striking

conclusion is that for $n \geq 12$ the total number of muls and adds for ADK becomes less than the total required for SB.

Interestingly Khachatrian et al. [8] appear to have got it wrong as well, over-estimating the number of additions required to an even greater extent, as always requiring 50% more additions than the SB method. However these previous over-estimates may be explained by the authors considering only a packed-radix representation.

	muls	adds
SB	n^2	$2n^2 - 4n + 2$
ADK	$n(n+1)/2$	$2n^2 + 2n - 6$

Table 1. Complexity

Processor designers go to great lengths to cut the cost in cycles of a mul instruction, even getting it down to 1 clock cycle, the same as that required for an add. However a mul will always require more processor resources, and thus a hidden extra cost will probably show up in actual working code. For example in a multi-scalar architecture only one processor pipeline might support hardware multiplication, whereas all available pipelines will allow simultaneous execution of adds, so whereas one mul can execute in 1 cycle, two or more adds might execute simultaneously. The actual break-even point between ADK and SB can only be determined on a case-by-case basis via an actual implementation. In this next table we calculate the ratio r between the costs of muls and adds that mark the expected break-even between SB and ADK.

n	SB muls	SB adds	ADK muls	ADK adds	r
5	25	32	15	54	2.2
9	81	128	45	174	1.28
12	144	242	78	306	0.97
16	256	450	136	538	0.69

Table 2. Operation Counts

This would appear to settle the matter: A variant of Karatsuba should be used for all multi-precision multiplications that involve numbers with 12 or more limbs. A caveat might be that the simplicity of the SB method might favour a compiler in terms of the number of memory accesses and register move instructions (not considered here) which it might require. However we suspect that any such advantage would be outweighed by the hidden resource consumption of even the fastest integer multiply.

On the other hand it remains a real possibility that a packed-radix implementation of the School-Boy method written in carefully hand-crafted assembly language might prove superior on particular processors, even beyond the 12 limb

limit (bearing in mind that a packed-radix representation may actually require less limbs). This could only be established experimentally. A useful resource for comparison purposes would be the well known GMP multi-precision library [6].

6 Some results

We tested our results on an industry standard Intel i3-4025U 1.9GHz 64-bit processor running in Windows. This is a simple head-to-head comparison of the reduced-radix SB and ADK methods. The test code was written in C, and compiled using the GCC compiler (version 5.1.0) with maximum optimization. It includes the carry propagation code. The multiplication code was fully unrolled, as a compiler cannot always be trusted to do this automatically. Our experience would be that optimized compiler output like this for Intel processors is very hard to improve upon, even using hand-crafted assembly language.

n	SB cycles	ADK cycles
5	75	78
9	234	185
12	397	324
16	687	577

Table 3. Intel i3-4025U Cycle Counts

These results more than support the conclusion to be drawn from table 2: In fact on this processor the cross-over point occurs already with just 9 limbs. On examining the generated code, it was observed that the number of mul and add-equivalent instructions were as predicted in the analysis above. However an inspection of the generated code also confirmed our suspicion that the ADK code generated more register-register move instructions and memory accesses. On some processors this could offset the ADK advantage.

7 Conclusion

In this note we have dusted off an old oft-rediscovered trick that we would suggest has not received sufficient attention from those interested in efficient cryptographic implementations. We have demonstrated that it is much more efficient than previously thought. We have established a concrete break-even point where Karatsuba variants should be considered ahead of the classic school-boy method for long multiplication.

Of course we are not claiming that the ADK method is necessarily the best choice in all circumstances. A classic recursive Karatsuba may well be superior in particular cases. For example Hamburg [7] uses a modulus that chimes particularly well with 1-level of classic Karatsuba. And Bernstein et al. [1] may well be correct in applying 2-level Karatsuba in their particular context.

The fact that a multiplication now requires the calculation of the same number of partial products as a squaring, might encourage implementors to use this multiplication algorithm for both squaring and multiplication, so that multiplications and squarings cannot be easily distinguished by some simple kinds of side-channel attack, like for example a timing attack.

8 Acknowledgements

The author would like to thank Rob Granger, Billy Bob Brumley and Paul Zimmermann for helpful comments on an earlier draft of this paper.

References

1. Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. Cryptology ePrint Archive, Report 2014/526, 2014. `http://eprint.iacr.org/2014/526`.
2. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Report 2011/368, 2011. `http://eprint.iacr.org/2011/368`.
3. R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.
4. R. Granger and M. Scott. Faster ECC over $\mathbb{F}_{2^{521}-1}$. In *Public-Key Cryptography – PKC 2015*, volume 9020 of *Lecture Notes in Computer Science*, pages 539–553. Springer Berlin Heidelberg, 2015.
5. Robert Granger and Andrew Moss. Generalised Mersenne numbers revisited. *Mathematics of Computation*, 82:2389–2420, 2013. `http://arxiv.org/abs/1108.3054`.
6. Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.0 edition, 2015. `http://gmplib.org/`.
7. Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. `http://eprint.iacr.org/2015/625`.
8. G. Khachatrian, M. Kuregian, K. Ispiryan, and J. Massey. Faster multiplication of integers for public-key applications. In *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 245–254. Springer Berlin Heidelberg, 2001.
9. P. Montgomery. Five, six and seven term karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.
10. Y. Nogami, A. Saito, and Y. Morikawa. Finite extension field with modulus of all-one polynomial and representation of its elements for fast arithmetic operations. *IEICE Transactions on Fundementals of Electronics, Communications and Computer Sciences*, E86-A(9):2376–2387, 2003.
11. Andre Weimerskirch and Christof Paar. Generalization of the Karatsuba algorithm for efficient implementations. Cryptology ePrint Archive, Report 2006/224, 2006. `http://eprint.iacr.org/2006/224`.
12. P. Zimmermann. Personal communication, January 2015.

More results

We carried out further tests on a variety of platforms. In all cases we used the GCC compiler tools. Where the well known GMP library could be installed, we provide a comparison with its assembly language `mpn_mul_basecase()` packed-radix SB implementation. However it should be noted that whereas the GMP code is only partially unrolled, ours is fully unrolled.

First up is a rather old Intel Core i5 chip running under the Ubuntu OS, and using GCC version 5.2.1.

n	SB cycles	ADK cycles	GMP cycles
5	87	106	99
9	234	248	289
12	400	380	506
16	691	626	921

Table 4. 64-bit Intel i5-M520 Cycle Counts

Next a more modern i5 variant, running on an Apple Mac Mini.

n	SB cycles	ADK cycles	GMP cycles
5	66	60	64
9	195	154	172
12	368	250	286
16	658	491	495

Table 5. 64-bit Intel i5-4278U Cycle Counts

Finally results for an old 32-bit Intel Atom processor, using GCC version 4.8.4

n	SB cycles	ADK cycles
5	373	313
9	1068	888
12	1824	1441
16	3193	2459

Table 6. 32-bit Intel Atom N270 Cycle Counts

Example Code

Here we present an example of the loop unrolled C code for the SB and ADK methods that we used in our tests. In this small example the number of limbs n in x and y is 5. Code for carry propagation is included. In practise this code is automatically generated by a small utility program for any value of n.

```
typedef int64_t small;
typedef __int128 large;
#define B 61 // bits in radix
#define M (((small)1<<B)-1) //Mask

void sbmul5(small *x,small *y,small *z)
{
 large t,c;
 t=(large)x[0]*y[0]; z[0]=(small)t&M; c=t>>B;
 t=c+(large)x[1]*y[0]+(large)x[0]*y[1]; z[1]=(small)t&M; c=t>>B;
 t=c+(large)x[2]*y[0]+(large)x[1]*y[1]+(large)x[0]*y[2]; z[2]=(small)t&M; c=t>>B;
 t=c+(large)x[3]*y[0]+(large)x[2]*y[1]+(large)x[1]*y[2]+(large)x[0]*y[3];
                                                  z[3]=(small)t&M; c=t>>B;
 t=c+(large)x[4]*y[0]+(large)x[3]*y[1]+(large)x[2]*y[2]+(large)x[1]*y[3]
                                     +(large)x[0]*y[4]; z[4]=(small)t&M; c=t>>B;

 t=c+(large)x[4]*y[1]+(large)x[3]*y[2]+(large)x[2]*y[3]+(large)x[1]*y[4];
                                                  z[5]=(small)t&M; c=t>>B;
 t=c+(large)x[4]*y[2]+(large)x[3]*y[3]+(large)x[2]*y[4]; z[6]=(small)t&M; c=t>>B;
 t=c+(large)x[4]*y[3]+(large)x[3]*y[4]; z[7]=(small)t&M; c=t>>B;
 t=c+(large)x[4]*y[4]; z[8]=(small)t&M; c=t>>B;
 z[9]=(small)c;
}

void adkmul5(small *x,small *y,small *z)
{
 large t,s,c,d[5];
 d[0]=(large)x[0]*y[0];
 d[1]=(large)x[1]*y[1];
 d[2]=(large)x[2]*y[2];
 d[3]=(large)x[3]*y[3];
 d[4]=(large)x[4]*y[4];

 s=d[0]; t=s; z[0]=(small)t&M; c=t>>B;
 s+=d[1]; t=c+s+(large)(x[1]-x[0])*(y[0]-y[1]); z[1]=(small)t&M; c=t>>B;
 s+=d[2]; t=c+s+(large)(x[2]-x[0])*(y[0]-y[2]); z[2]=(small)t&M; c=t>>B;
 s+=d[3]; t=c+s+(large)(x[3]-x[0])*(y[0]-y[3])+(large)(x[2]-x[1])*(y[1]-y[2]);
                                                  z[3]=(small)t&M; c=t>>B;
 s+=d[4]; t=c+s+(large)(x[4]-x[0])*(y[0]-y[4])+(large)(x[3]-x[1])*(y[1]-y[3]);
                                                  z[4]=(small)t&M; c=t>>B;

 s-=d[0]; t=c+s+(large)(x[4]-x[1])*(y[1]-y[4])+(large)(x[3]-x[2])*(y[2]-y[3]);
                                                  z[5]=(small)t&M; c=t>>B;
 s-=d[1]; t=c+s+(large)(x[4]-x[2])*(y[2]-y[4]); z[6]=(small)t&M; c=t>>B;
 s-=d[2]; t=c+s+(large)(x[4]-x[3])*(y[3]-y[4]); z[7]=(small)t&M; c=t>>B;
 s-=d[3]; t=c+s ; z[8]=(small)t&M; c=t>>B;
 z[9]=(small)c;
}
```