Logarithmic Speed Modular Multiplication

Colin D. Walter

Department of Computation
UMIST
PO Box 88
Manchester M60 1QD, UK
www.co.umist.ac.uk

Abstract. A design for logarithmic speed modular multiplication is given and a comparison made with the best implementations of existing, more standard algorithms by Brickell and Montgomery. A 40-fold increase in speed is reported by using chip area at the limit of current technology.

Key words: Computer Arithmetic, Modular Multiplication, RSA cryptography.

1 Introduction

Most modular multiplication algorithms in current use perform their arithmetic on n-digit inputs in $O(n \log n)$ time and $O(n)$ space using what amounts to the standard paper and pencil method with a redundant representation of the numbers [1]. Much work has gone into improving the efficiency of these algorithms and good estimates of their speeds in terms of gate delays are now available [3], [9].

Many cryptographic applications, such as decryption of RSA [5], require high speed modular arithmetic on large numbers. It is presumably well understood that modular multiplication in particular can be performed on inputs with $O(n)$ digits in $O(\log n)$ time at the price of $O(n^2)$ space. Here we give details for this and quantify upper bounds on the time and space more precisely in order to see if a change of algorithm can reasonably be made to trade space for speed and yet remain within feasible technology. Pipelining the algorithm increases the throughput to 1 modular multiplication per $O(1)$ clock cycles, although the latency is still $O(\log n)$.

In a previous work [10] the author showed how a systolic array could be built for modular multiplication with a throughput of one product per clock cycle on a very fast clock ($O(1)$ time), also using $O(n^2)$ space. However, it requires $O(n)$ time between the input and output of corresponding digits. Using pipelining as noted above, the algorithm here would provide essentially the same performance as the systolic array $-$ the same throughput for the same cost in area - but the much lower latency of $O(\log(n))$ time instead of $O(n)$.

In a related work [6], Shand et al. describe a programmable array implementation of Montgomery's algorithm [4]. The details of their implementation are not given, but the performance would appear to be similar to that of the systolic array [10] if they have the ability to pipeline a number of modular multiplications to use the hardware simultaneously. Thus the hardware here should again provide a significant advantage. Their measure of time is an absolute one using a particular technology and gives an $O(n)$ time for latency, an order of magnitude slower than what is described here.

Upper and lower bounds on the area/time efficiency of multiplication are given by Brent and Kung [1]. By employing the discrete Fourier transform, they are able to describe a method that would provide modular multiplication using $O(n \log n)$ area and $O(\sqrt{n} \log n)$ time. This is intermediate in terms of both area and time between the method here and the standard methods of [2] and [3]. With its smaller product of $Area{\times}Time$, as well as an absolute area that suits current technology, it is probably an over neglected alternative. It is worth noting that minimum $Area{\times}Time$ is believed to increase as time is reduced to its asymptotic minimum, and the area here still appears to be the best for the logarithmic time it takes [7].

It turns out that cryptographic applications requiring around 500-bit numbers are well served by logarithmic methods as far as speed is concerned, although not so well space-wise as far as current technology is concerned. Without effort, a 40-fold increase in speed is obtained here for a 500-fold increase in area, resulting in the requirement for $5{\times}10^6$ XOR gates or equivalent for a H/W implementation. Such hardware is appropriate for heavy centralised RSA encryption and decryption using the same key or keys continuously, and is just about feasible nowadays using redundancy in wafer-scale integration techniques.

2 The Algorithm

We split the computation of $(A{\times}B) \bmod M$ into six distinct phases, namely the various function applications in $rnd(fract((A{\times}B){\times}(1/M)){\times}M)$ where $fract$ discards the integer part of a real number, and retains the non-negative fractional part, and rnd rounds a real to the nearest integer. Here $fract$ is discarding $(A{\times}B)$ div M so that multiplication by M leaves $(A{\times}B) \bmod M$. Various approximations are made, which leave a small fractional part to be rounded off. In particular, we assume that M^{-1} is already known to just over $3m$ places after the point where m is the number of digits in M, A and B. Of course, the first m (approximately) of these digits are zero. Then $(A{\times}B){\times}(1/M)$ has an accuracy to over m places after the point, so that multiplying its fractional part by M will give a result that is accurate to within $\frac{1}{4}$, say. Rounding to the nearest integer will then give the correct answer. The precise accuracy needed in the calculations is not important for the discussion here, but is easy to find.

The only requirement is that sufficient accuracy is employed to perform exact rounding by inspecting a small constant number of fractional digits.

Accurate application of $fract$ takes time. A change of 1 in the value of $A\times B$ changes $(A\times B)\times(1/M)$ by about 1 in the position m places after the point. Hence $fract$ may need to examine just more than the first m places after the point for complete accuracy. By examining only one or two such places, $fract$ may be out by ± 1, so that the end result differs by M from $(A\times B) \bmod M$. The output is then in the range 0 to $2M$ rather than 0 to M. This is a typical penalty of saving time, but is not a problem in RSA cryptography where a large number of consecutive modular multiplications are performed. In such cases, the correct range 0 to M need only be achieved after the last operation. We will assume this approximation to $fract$ is used.

3 The Notation

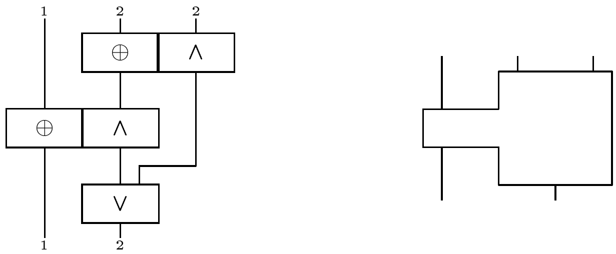
Using a redundant number system enables addition to be done with bounded carry propagation. We assume A and B and all intermediate results have such forms, but not M or $1/M$, which we will suppose to be in binary form. We will also assume that each multiplication is done sequentially on the same hardware. The prohibitive cost of the extra hardware does not seem to warrant the small extra speed achievable by trying to do more at once.

Let b be the base of our redundant representation. We may assume $A < 2M$ and $B < 2M$. Then $m = \lceil \log_b 2M \rceil$ is the maximum number of (redundant) digits in A or B, and $n = \lceil \log_2 M \rceil$ the number of bits in the modulus M. The largest multiplication involved here is that by $1/M$. So we need hardware to perform a $2m\, digit\times 2n\, bit$ multiplication. The product $A\times B$ of m *digit* redundant numbers can be split fairly easily to use this hardware if the degree of redundancy is not unreasonable.

4 The Largest Multiplication

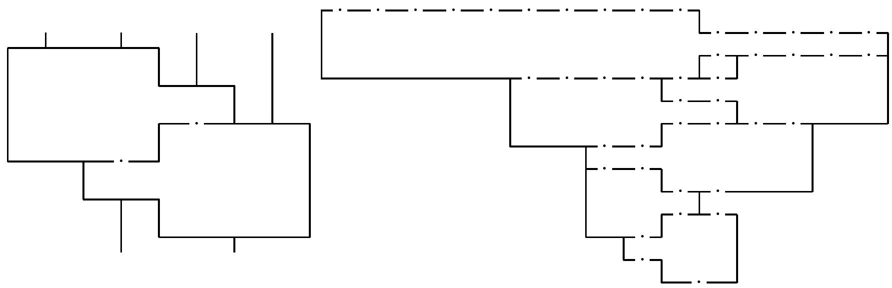
The $2m\, digit\times 2n\, bit$ multiplication generates $4mn$ digits, with up to $2n$ of these representing the same power of b. Here $\frac{1}{8}^{th}$ of them will be discarded when $fract$ is applied because they overflow into the integer part. Also, almost half of them represent such low powers of b that they can only influence the rounding process through carry propagation: those with indices below about $-m-\log_b m$ will make a total difference of less than $\frac{1}{2}$ to the final outcome and so can be ignored. Thus we can assume that only about $2mn$ digits need adding together. These can be reduced to 2 redundantly represented numbers using a tree structure of 3-to-2 redundant number adders built from 3-to-2 bit adders (Figure 1). This tree has a maximum depth of about $\log_{3/2} n$ 3-to-2 bit adders and involves an area of one 3-to-2 bit adder for every bit removed in transforming the input digits into output digits. This is essentially Wallace's construction [8].

In fact, the 3-to-2 adders can be more closely packed than counting a depth of 3 gates each as illustrated schematically in Figure 2. Some inputs are not needed



(a) Full Adder for Bits (b) Full Adder for Numbers

Fig. 1. 3-to-2 Adder



(a) 4-to-2 Adder (b) 15-to-2 Adder with depth of 12 gates

Fig. 2. Packing of Adders

immediately and some outputs are generated early. Hence adders like the 15-to-2 adder in the figure can be constructed with depths of close to $\log_{6/5} n/2$ for an n-to-2 adder. In RSA cryptography we can expect n to be a little in excess of 512 and so about 32 gates depth will suffice to reduce the $2n$ summands of the multiplication to 2 redundant numbers. A further 2 or at most 3 gates will reduce this to a single redundant number (e.g. consider using 2 binary registers to hold a redundant number).

Now let us consider the initialisation time to create the summands for adding. Each input bit needs to be distributed to $2n$ different positions for ANDing with a bit from the other multiplicand. This can be done by a tree of multiplexers of depth no worse than, say, $\log_4 2n$, which is about 5 in our case. However, only one digit position has as many as $2n$ digits to sum, so that digits can be routed preferentially there in order to cut the overall depth. Adding the time of 2 gates for setup and hold times of registers, plus one or two gates to enable the hardware to be used for the various multiplications leads to a grand total of about 40 gates along a critical path. With 3 multiplications being done to perform the modular multiplic- ation, we obtain 120 gates delays altogether over the three clock cycles needed.

5 Comparison with Standard Implementations

The standard algorithms when pushed to their limits, require a constant depth of 9 or 10 gates per clock cycle (see [9] and [3]), with marginally over n clock cycles per modular multiplication. For $n = 512$ or so, this means the method above provides a *forty*-fold increase in speed, which almost doubles every time n is doubled. Indeed, repeating the calculations shows a speed advantage for word lengths right down to $n = 8$, at which point a non-redundant representation would be employed to do the whole multiplication in a single clock tick.

What is the cost? The area of the standard algorithm is linear, as is the time. However, the logarithmic time for the new algorithm requires $0(n^2)$ area. Specifically, about $20n$ gates for the standard algorithms (more for the fastest implementations), and $20n^2$ here if two binary numbers provide the redundancy, with linear changes to both for other representations. Overall, the logarithmic method is therefore n times more expensive in area, requiring some 5×10^6 gates for typical choices of $n \approx 5\times10^2$. This is just about possible with current technology.

We have neglected the effect of long wire lengths. Along the critical path this will typically be comparable with the edge length of the area of the chip being utilised in both algorithms because information needs to be transported right across the chip. For the classical algorithms, this will be of length $O(\sqrt{n})$, and so adds $O(n\sqrt{n})$ time to the multiplication, whereas for the scheme here the length is $O(n)$, which adds just $O(n)$ to the total time. Thus wire length eventually dominates the time in both algorithms, but is always much greater for the standard algorithms.

Finally, we note that some of the hardware stands idle at the end of each multiplication cycle. It is only the middle digits in the output which required the full depth of the hardware for their calculations. The end digits can be produced with almost zero depth. However, on average a typical digit requires the summation of half the number of inputs that the middle digits have. The tree structure of the addition means that, although only half the hardware is need for this, its depth is just one gate less than for a middle digit, so that it takes almost the same time as the worst case. Hence, overall the hardware is running at almost full capacity. The only improvement in use could be through pipelining, as mentioned earlier.

6 Conclusion

It is now possible to consider logarithmic time (Wallace tree) VLSI implementations for modular multiplication. This stretches current technology to the limit, but provides nearly two orders of magnitude increase in speed when inputs have 500 or so bits, as typically in RSA cryptography.

References

[1] R.P. Brent & H.T. Kung, *The Area-Time Complexity of Binary Multiplication*, J.ACM, vol. **28**, 1981, pp 521-534.

[2] E. F. Brickell, *A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography*, Advances in Cryptology - CRYPTO '82, ed. Chaum et al., Plenum, 1983, pp. 51-60.

[3] S.E. Eldridge & C.D. Walter, *Hardware Implementation of Montgomery's Modular Multiplication Algorithm*, IEEE Trans. Computers, vol. **42**, 1993, pp. 693-699.

[4] P.L. Montgomery, *Modular Multiplication without Trial Division*, Mathematics of Computation, vol. **44**, 1985, pp. 519-521.

[5] R. L. Rivest, A. Shamir & L. Adleman, *A Method for obtaining Digital Signatures and Public-Key Cryptosystems*, Comm. ACM, vol. **21**, 1978, pp. 120-126.

[6] M. Shand, P. Bertin & J. Vuillemin, *Hardware speedups in long integer multiplication*, ACM SigArch, vol. **19** no. 1, March 1991, pp 106-113.

[7] B. Sugla & D.A. Carlson, *Extreme Area-Time Tradeoffs in VLSI*, IEEE Trans. Comput., vol. **39**, 1990, pp 251-257.

[8] C.S. Wallace, *A suggestion for a fast multiplier*, IEEE Trans. Elec. Comput., vol. **EC-13**, 1964, pp 14-17.

[9] C.D. Walter, *Faster Modular Multiplication by Operand Scaling*, Advances in Cryptology - CRYPTO '91, Lecture Notes in Computer Science, vol. **576**, Springer-Verlag, 1992, pp. 313-323.

[10] C.D. Walter, *Systolic Modular Multiplication*, IEEE Trans Computers, vol. **42**, 1993, pp 376-8.