Optimizing Multiprecision Multiplication for Public Key Cryptography

Michael Scott[1] and Piotr Szczechowiak[2]

[1] School of Computing
mike@computing.dcu.ie
[2] School of Electronic Engineering
Dublin City University
Ballymun, Dublin 9, Ireland.
piotr@eeng.dcu.ie

Abstract. In this paper we recall the hybrid method of Gura et al. for multi-precision multiplication [4] which is an improvement on the basic Comba method and which exploits the increased number of registers available on modern architectures in order to avoid duplicated loads from memory. We then show how to improve and generalise the method for application across a wide range of processor types, setting some new records in the process.
Keywords: Implementation. Multi-precision arithmetic.

1 Introduction

Multiprecision multiplication is the time-critical requirement in the great majority of number-theoretic based methods for public key cryptography. For example the RSA method [8], the El Gamal method [8], methods based on elliptic curves [5], and the new methods of pairing-based cryptography [10] are all dependent on it. In fact at a higher level the requirement is actually for modular arithmetic, where multiplications are typically carried out modulo a certain fixed modulus. A typical calculation in the context of implementing the RSA algorithm might be $y = a \cdot b \bmod p$, where p is a fixed 512-bit prime number. However using Montgomery's method [9] the modular reduction can be carried out without using division, and hence it is the multi-precision multiplication (or squaring) which is significant. Sometimes it is advantageous to integrate the multiplication and reduction steps into a single algorithm [6]. Alternatively, and particularly in elliptic curve cryptography, the modulus may have a special simple form that can be exploited for fast modular reduction. Since the modulus is of a fixed size in bits, the code for the multiplication (and squaring) and for the modular reduction can be written in assembly language, and the loops completely unrolled for maximum speed. Then the cryptographic system of choice can be built on top of these primitives, and benefit from their efficient implementation.

Most efficient implementations use the method of Comba [1], which is a column-wise implementation of the basic school-boy method for long multiplication (also known as the "product scanning" method). This is a little awkward to

program as the columns are of different lengths, although this is not a concern if we unroll the code. The alternative row-wise method (also known as "operand scanning") is also often used, as it is much easier to implement as a short looped program – an n digit by n digit multiplication can be carried out using a simple pair of nested "for" loops. For more details see Chapter 14 of [8].

Recently [4] it has been suggested that a hybrid method which combines features of both methods might be preferable, particularly in a setting where the processor has many available general purpose registers, which is the case with most modern architectures which often have at least 16 (for example x86-64 architectures), and perhaps as many as 128 registers (e.g. the IBM/Sony Cell architecture). This method was first described in the context of the ATmega-128L 8-bit processor, which has 32 registers. In this same setting Uhsadel et al. [12] have recently obtained improved results.

In this paper we simplify and attempt to generalise the method and show its application to a wide range of processors. We also demonstrate the superiority of our modified scheme to the earlier proposals.

2 Improved Hybrid Method

Consider a processor with a natural word length of w bits, and hence equipped with r registers of w bits. Typically r will be 8, 16 or 32. Typically w will be 8, 16, 32 or 64. Assume that the processor architecture supports simple indexed memory load and store instructions, an integer multiply instruction which from an input of any two registers produces their product in a pair of registers, and finally a simple add-with-carry instruction.

We accept that not all of the r registers may be available to us. Indeed some will certainly be required to store the memory addresses of the operands and of the result. It is also sometimes helpful if a register fixed with the constant value of zero is available to us.

The basic column-wise algorithm for multi-precision multiplication, also known as the "Comba" method, is illustrated for $n = 4$ in Figure 1 (i). As each w-bit pair of digits are multiplied together to create one $2w$ bit partial product in a pair of registers, this partial product is accumulated in a triple register (which we will sometimes refer to as the "column registers"). The third register (here called a "carry-catcher") is required to catch the carries that can arise as the full column is added up. The diagram shows the addition of a particular partial product, and the arrows indicate the possible carries that will arise. Note that the maximum number that can arise in the third and most significant register is bounded by n, the number of digits in the multi-precision multiplication. This is likely to be much smaller than $2^w - 1$, the maximum number that can be stored in a register, and so sometimes a smaller register (if available, for example the 8-bit cl register in the 32-bit x86 architecture) can be used here.

The column-wise algorithm is to be preferred as memory writes only occur at the foot of each column. Here the least significant register of the triple register is written to memory, and the other two registers shifted down, representing the

carry to the next column. On a computer with only 8 registers like most of the early x86 family, Comba [1] found that there were just enough registers available to implement this method successfully.

The alternative row-wise algorithm requires more memory writes, but less memory reads of the operands. A disadvantage of the column-wise algorithm is that it will reload the same data many times as it is required in the calculation of more than one partial product. This is clearly a little foolish if there are enough registers available to store some of these values for later use. So it is not surprising to observe that we could do better.

The hybrid method [4] keeps the advantages of the column-wise method, while exploiting any extra registers to avoid unnecessary reloads of data. The idea is conceptually a simple one – basically perform the multiplication as if the word length of the computer were actually $m \cdot w$, and perform the $m \times m$ multiplication that arises in the calculation of the larger partial product using the row-wise algorithm. In Figure 1 (ii) we see an illustration of the hybrid method for the case $m = 2$. As each large 2×2 partial product is calculated (represented by the large outer boxes), it must be accumulated into registers. As before the diagram illustrates the acccumulation of a particular partial product, and the arrows indicate the carries. A naive implementation would then require 5 registers in this case to store this sum. However in the worst case this might require up to 5 add-with-carry instructions, as in integer addition a carry-out is always a possibility that must be catered for.

The original hybrid method [4], and the improved variant described in [12], use a rather complex method to deal with this carry propagation problem, based on the fact that the product of two words, plus two words, cannot overflow a double-word-sized register since

$$(2^w - 1)(2^w - 1) + (2^w - 1) + (2^w - 1) = 2^{2w} - 1 < 2^{2w}$$

In contrast our main idea is to simply employ even more registers to suppress the carry propagation. As well as using $2m$ column registers to hold the sum of each now wider column, an extra $2m - 1$ registers are deployed as "carry catchers". These registers are initialised to zero at the top of the column and catch any carries that may arise without the possibility of further carry propagation. Then when the column is finished, these carry catchers are simply added with carry to the main set of column registers. The case for $m = 2$ is shown in Figure 1 (iii). Observe that the most significant carry-catcher register performs exactly the same role as the fifth register in the naive implementation.

The $m \times m$ row-wise multiplication requires $m + 1$ registers, m registers to hold the entire row, each element of which is then multiplied by another register (which stores the current multiplier) to create m partial products. There are $2m$ column registers and $2m-1$ carry-catchers, for a total of $5m$ registers. Note that for $m = 1$ this is the original Comba method, which requires 5 registers, just about possible if $r = 8$. The choice $m = 2$ requiring 10 registers is a good fit for a processor with a total of $r = 16$ registers, and $m = 4$ will take 20 registers for

the case $r = 32$. These figures are approximate and may require modification for a particular architecture.

An important general point – implementation will always be processor specific, as processors differ in small but significant ways that can not be described in a one-size-fits-all way. Indeed the prior art that we compare our method with, is described only in the specific contexts of particular processors [1], [4], [12]. However as evidence of the generality of the idea we describe it in the contexts of three quite different processors, and for these processors we do achieve very useful speed-ups compared with the prior art, of from 8-20%. Nonetheless this cannot be taken as proof that the method will always work as well as reported here.

We remark that the use of the hybrid method as described here and also in [4] and [12], rather conflicts with the type of instruction set and architectural enhancements as proposed by Großschädl and Savas [2], and as implemented in the SmartMIPS^{TM} variant of the popular RISC MIPS32 architecture [11]. Those enhancements, incorporating as they do a special 72-bit "triple register" are specific to the original Comba method, and cannot easily be modified to support the hybrid method. It may be that adequate performance can be achieved at lower cost on certain architectures by simply incorporating more registers into the architecture, or perhaps fully using the registers that are already there, rather than by including specialised instruction set extensions.

3 The Atmel AVR ATmega-128L processor

The Atmel AVR ATmega-128L is an 8-bit RISC processor, very popular for use in Wireless Sensor Networks (WSNs). On the face of it, it is an unpromising candidate for public key cryptography. However the authors of [4] demonstrated that elliptic curve cryptography in particular can be implemented quite successfully on this platform. In particular a 160-bit point multiplication on a standard elliptic curve can be carried out in just 0.81 seconds when the device is clocked at 8MHz.

The processor has 32 registers, denoted r0 to r31. There are three memory addressing registers, denoted X, Y and Z, which actually refer to the 16-bit register pairs r26:27, r28:29 and r30:31.

Now consider the calculation of a typical single partial product on this processor using the standard Comba method. See Figure 2. The Y and Z registers are pointing at 8-bit digits of the multiplicand and the multiplier respectively. The X register is pointing at the address which will eventually contain the sum of this column. Using standard indexed addressing and the `LDD` instruction, the multiplicand and multiplier are loaded into the r0 and r1 registers. The `MUL` instruction always places the 16-bit result in the register pair r0:1. The "triple register" in this case is r6:7:8 and the register r5 is kept as a constant zero. Then an `ADD` and two `ADC` instructions are used to add the new product into the triple register, taking care of the carries.

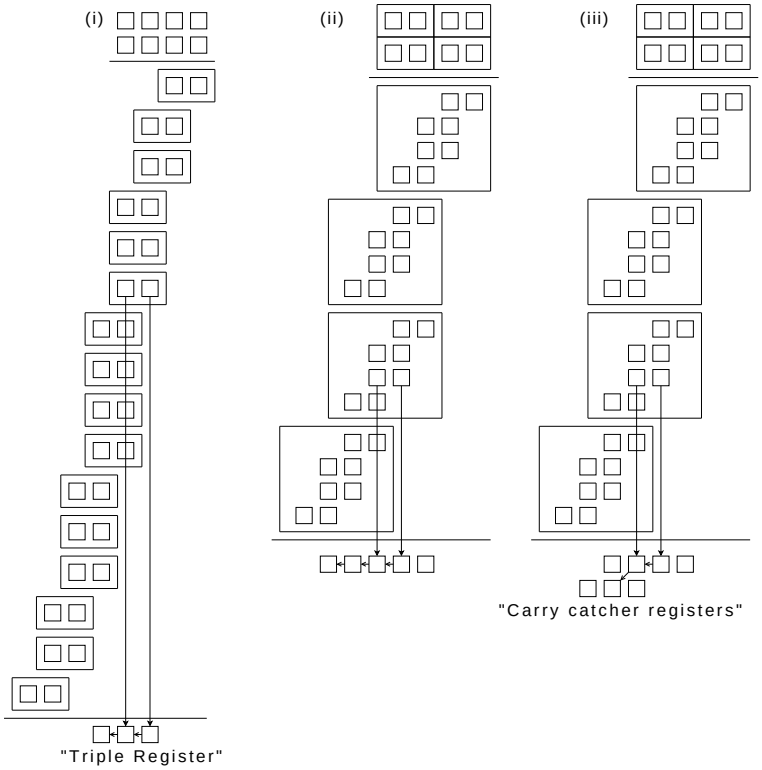


Fig. 1. The processing of a two word partial product using (i) The standard Column-wise Comba implementation, (ii) A naive implementation of the hybrid method and (iii) Our improved hybrid implementation

```
LDD r0,Y+2
LDD r1,Z+1
MUL r1,r0
ADD r6,r0
ADC r7,r1
ADC r8,r5
```

Fig. 2. The processing of a single partial product on the ATmega-128

Now try again, this time using our hybrid method with $m = 2$. In this case 4 partial products are calculated together in each step, requiring more registers. See Figure 3. In this case the column registers are r6:7:8:9, and r10:11:12 are used as the carry-catchers. This time r25 is used to hold the constant zero.

A simple counting exercise now reveals the advantage of the hybrid method. To calculate 4 partial products using the simple Comba method would require 4 times the operations of figure 2, that is 8 loads, 4 multiplies, 4 additions and 8 additions-with-carry. However for the same work carried out, the hybrid method requires 4 loads, 4 multiplies, 4 additions and 8 additions-with-carry, from figure 3. Clearly we have saved 4 load instructions, which is particularly significant as the load instruction takes 2 clock cycles, the same as the multiply instruction, but more than the addition instructions which require only 1 clock cycle each. For this architecture at least, it can be seen that using the proposed method we are able to take maximum advantage of the "hybrid" idea.

Furthermore we still have enough unused registers to comfortably extend the idea to $m = 4$ with further savings. Using this method the number of clock cycles for a full 160×160 bit multiplication is reduced to 2651 clock cycles, which compares favourably with the 2881 clock cycles recently reported in [12] (which is claimed as "the fastest implementation world wide of a modular multiplication of a 160-bit standardized elliptic curve for an 8-bit micro controller"), and the 3106 clock cycles reported in [4] (we are approximately 15% faster). This is despite the fact that these last two papers use $m = 5$ rather than the $m = 4$ maximum that we are constrained to use due to our extra requirement for carry-catcher registers. See Table 1.

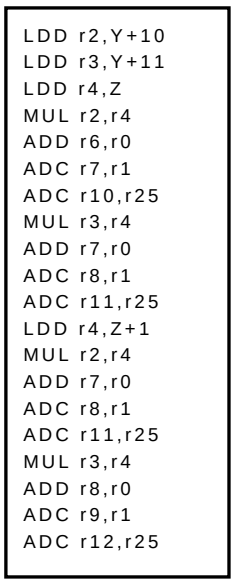
We have omitted details of the processing required at the foot of each column, as for an $n \times n$ digit multiplication this is required just $2n$ times, compared with the n^2 partial products which must be calculated and accumulated. However in the hybrid case there is admittedly some extra work involved here as the "carry-catcher" registers must be added-with-carry to the column registers. Then m registers are written to memory via the X register, and the carry value for the next column calculated. For this particular architecture advantage can be taken here of the `MOVW` instruction which moves two registers in a single clock cycle.

When squaring advantage can be taken of the fact that the off-diagonal partial products need only be calculated once, and then accumulated twice, with some savings. We omit the details. We also note that an unrolled and optimized version of the Montgomery **REDC** modular reduction function [9] can also exploit the same basic hybrid method.

One major advantage of the proposed method is that, due to its simplicity, it is relatively easily extended to other architectures.

4 The TI MSP430 processor

We choose the Texas Instruments MSP430 processor as our second architecture on which to test the efficiency of our improved hybrid multiplication method. The TI MSP430 like the ATmega-128L is also widely used in WSNs and many



```
LDD r2,Y+10
LDD r3,Y+11
LDD r4,Z
MUL r2,r4
ADD r6,r0
ADC r7,r1
ADC r10,r25
MUL r3,r4
ADD r7,r0
ADC r8,r1
ADC r11,r25
LDD r4,Z+1
MUL r2,r4
ADD r7,r0
ADC r8,r1
ADC r11,r25
MUL r3,r4
ADD r8,r0
ADC r9,r1
ADC r12,r25
```

Fig. 3. The processing of 4 partial products using the hybrid method on the ATmega-128

Table 1. Comparison of instruction counts for ATmega-128

		This Work		Uhsadel et al.		Gura et al.		Classic Comba	
Instruction	CPI	Instructs	Cycles	Instructs	Cycles	Instructs	Cycles	Instructs	Cycles
add/adc	1	1263	1263	986	986	1360	1360	1200	1200
mul	2	400	800	400	800	400	800	400	800
ld	2	200	400	238	476	167	334	800	1600
st	2	40	80	40	80	40	80	40	80
mov/movw	1	70	70	355	355	355	355	81	81
other			38		184		197	44	44
Totals			**2651**		**2881**		**3106**		**3805**

other embedded applications. But there the similarity ends. The MSP430 is a 16-bit RISC processor with a memory-memory architecture rather than the more classic RISC load-store architecture of the Atmel chip. The TI product offers 27 instructions in 7 addressing modes and uses a hardware multiplier for 8 and 16-bit integer multiplication. This orthogonal architecture allows every instruction to be used with every addressing mode. Only 12 from total number of 16 registers are available for general use. Registers r0-3 are used for Program Counter, Stack Pointer, Status Register and Constant Generator respectively. Registers r4-15 are general purpose and available for use at all times. A very nice feature of the MSP430 is its ultra low power consumption which is especially important on tiny devices in distributed environments, where battery usage is a big issue.

In order to achieve best performance for multiprecison multiplication on the MSP430 it is important to use the hardware multiplier. This device is a peripheral and is not implemented in every member of the large MSP430 family of microprocessors. It is accessed via memory mapped I/O registers. We used the MSP430F1611 platform in our research, which does include this hardware multiplier.

Figure 4 shows the calculation of a typical partial product on this architecture using the standard Comba method. Registers r13 and r14 store the memory addresses of the first 16-bit digits of the multiplicand and the multiplier. Indexed addressing mode is used to find the address of a particular 16-bit digit. In this case the third digit of the multiplicand and the second digit of multiplier are loaded into the hardware multiplier registers using the memory-to-memory `MOV` instruction. Writing the first operand to the `MPY` register selects unsigned multiply mode but does not start any operation. Loading the second operand to `OP2` initiates the multiply operation. The lower and the higher 16-bit parts of the 32-bit result are stored in `RESLO` and `RESHI` registers respectively and can be read by the next instruction. One `ADD`, one `ADDC` and one `ADC` (add carry to destination in the MSP430 instruction set) instructions are used to add the new product to the "triple register" which consists of r9:10:11.

```
MOV    4(r13),&MPY
MOV    2(r14),&OP2
ADD    &RESLO,r9
ADDC   &RESHI,r10
ADC    r11
```

Fig. 4. The processing of a single partial product on MSP430F1611

Due to the limited number of available registers we could only implement our hybrid method with $m = 2$ on the MSP430. See Figure 5. Eleven registers are required to calculate the 4 partial products in every step of this algorithm. This

time r4:5:6:7 are the column registers, and r13:14:15 are the "carry-catchers". Registers r10:11 are used as pointers to particular digits in the multiplicand and multiplier. In the MSP430 instruction set the number of clock cycles per instruction depends both on the instruction mnemonic and the type of operands being used. Because memory-to-memory `MOV` instructions require the most cycles, we employ two extra registers r8 and r9 as temporary storage to save on the overall cycle count (memory-to-register operations are much less expensive).

```
MOV    20(r10),r8
MOV    22(r10),r9
MOV    12(r11),&MPY
MOV    r8,&OP2
ADD    &RESLO,r4
ADDC   &RESHI,r5
ADC    r13
MOV    r9,&OP2
ADD    &RESLO,r5
ADDC   &RESHI,r6
ADC    r14
MOV    14(r11),&MPY
MOV    r8,&OP2
ADD    &RESLO,r5
ADDC   &RESHI,r6
ADC    r14
MOV    r9,&OP2
ADD    &RESLO,r6
ADDC   &RESHI,r7
ADC    r15
```

Fig. 5. The processing of 4 partial products using hybrid method on MSP430F1611

At a first glance it might appear that the cost of both methods (hybrid and standard) is exactly the same on the MSP430. Calculation of 4 partial products using the simple Comba method requires 8 `MOV`, 4 `ADD`, 4 `ADDC` and 4 `ADC` instructions. The hybrid method uses also 20 instructions with exactly the same mnemonics. However on closer inspection we will see that the advantage of the hybrid method lies in the total number of clock cycles. The 8 `MOV` instructions in the simple algorithm takes 48 clock cycles in total. The hybrid method can take advantage of register to memory operations and saves 14 cycles using only 34 cycles for the same work carried out. As we can see the benefit in clock cycles of our improved hybrid method is very similar on both the MSP430 and the ATmega128-L in the case where $m = 2$. Note that, as for the Atmel chip, the savings are due to the overall reduction in memory read accesses to the digits of the multiplier and multiplicand.

Table 2. Comparison of instruction counts for MSP430F1611

Instruction		CPI	This Work		Classic Comba	
			Instructions	Cycles	Instructions	Cycles
add	&label,reg	3	100	300	100	300
addc	&label,reg	3	100	300	100	300
adc	reg	1	109	109	100	100
mov	x(reg),&label	6	45	270	180	1080
mov	reg,x(reg)	4	20	80	20	80
mov	reg,reg	1	27	27	38	38
mov	reg,&label	4	100	400		
mov	x(reg),reg	3	45	135		
other				125		167
Totals				**1746**		**2065**

We cannot achieve further savings on the MSP430, as we have used all 12 available registers in our hybrid method with $m = 2$. Using this algorithm 160×160 bit multiplication was performed in 1746 clock cycles, which is a nice improvement compared to the 2065 cycles for the standard Comba method. See Table 2 for details.

Assuming that our MSP430 device is clocked at the standard frequency of 8MHz we can calculate the result of the multiplication of two 160-bit numbers in 0.22ms. Our result compares favourably with the 0.95ms reported for the same operation on a MSP430F1611 in [13]. In [3] Guajardo et al. tested multiprecision multiplication on the TI MSP430x33x family of processors, which differs slightly in design from our platform. The common features are the same instruction set and the presence of the hardware multiplier. In order to compare results in [3] with our achievements we had to run our multiplication algorithm with 128-bit numbers. Once again our improved hybrid method proved to be more efficient using only 1154 clock cycles against the 1425 reported in [3].

5 The ARM processor

Finally we consider the popular 32-bit ARM processor. This has a standard RISC load-store architecture, with several innovative features, including effectively free shifting of operands and conditional execution. It has 16 registers, although significantly three of these are reserved for special purposes, r13 as a stack pointer, r14 holds a function return address, and r15 is the program counter. The ARM processor has recently made an appearance in the WSN segment, but is more well known as the processor of choice for PDA and mobile embedded applications.

Figure 6 shows the calculation of a typical partial product on this architecture using the standard Comba method. Registers r5 and r6 store the memory addresses of the first 32-bit digits of the multiplicand and the multiplier. In-

dexed addressing mode is used to find the address of a particular 32-bit digit. The `UMULL` instruction calculates the 64-bit product in registers r8 and r9. These are then added-with-carry to the triple register r2:3:4.

```
LDR r0,[r5,#4]
LDR r1,[r6,#8]
UMULL r8,r9,r0,r1
ADDS r2,r2,r8
ADCS r3,r3,r9
ADC r4,r4,#0
```

Fig. 6. The processing of a single partial product on the ARM

However a naive attempt to implement our hybrid method with $m = 2$ will fail, due to a lack of registers. For the multiplication algorithm 3 registers are required to hold the addresses of the operands, 4 more are required as column registers, 3 as carry-catchers, and 2 more for the row elements, plus another for the multiplier. Furthermore the multiplication instruction requires 2 registers to hold the 64-bit product of a pair of 32-bit registers, for a total of 15 registers.

We solve this problem by exploiting the novel features of the instruction set. The basic idea is to use just one carry-catcher register instead of three. As the carry-catcher registers are used to catch and accumulate the sum of individual carry bits, they do not require the 32-bit precision of a full register. In fact for any reasonable application with big number operands of a useful size, a byte would be sufficient. We exploit this fact to compress the carry-catcher requirement into a single register, without incurring any extra cost.

Our solution is illustrated in figure 7.

The trick is in the use of the `ADDCS` instruction (ADD if Carry Set) which adds a carry bit to a specific byte in the shared carry-catcher register r11. Here r2:3:4:10 are the column registers. In this case the savings are again revealed by simple inspection – we have saved four load instructions.

The foot-of-column processing is a little complex, as the individual carry-catcher components must now be masked and shifted before being added to the column registers. However a register argument can be shifted at no extra cost using the ARM's barrel shifter, so the overall cost is small.

The method was implemented and tested in the context of a 192×192 bit multiplication. Using the ARM RealView simulator [7] in the context of the ARM7 variant of the architecture, we found that the basic Comba method required 580 clock cycles, whereas our modified hybrid method required only 487 cycles.

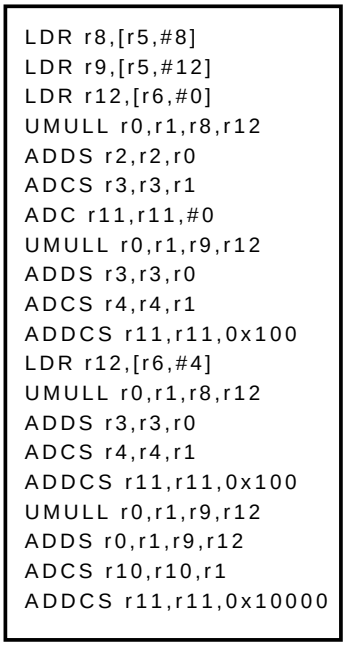


Fig. 7. The processing of 4 partial products using hybrid method on the ARM

6 Conclusion

We have described a simple and relatively generic method of implementing the hybrid method for multiprecision multiplication. The general applicability of the idea has been demonstrated on three quite different 8-bit, 16-bit and 32-bit architectures. In all of these cases significant performance improvements have been achieved.

If there should be insufficient registers, one idea would be to drop intermediate carry-catchers, and to allow the carry to propagate past the missing carry-catcher to the next available one, at the cost of some extra add-with-carries in the calculation of each partial product.

As a negative result we would like to report that the proposed method with $m = 2$ when applied to the x86-64 architecture, and attempting to exploit the extra registers r8-r15 available with this architecture, did not produce any improvement in our experiments. This is explained in part by the fact that the unsigned multiply instruction always expects one of the 64-bit multiplicands to be in the rax register, and the extra register moves required to get it there offset any advantage.

References

1. P. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.
2. Johann Großschädl and Erkay Savas. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In *Cryptographic Hardware and Embedded Systems - CHES'2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 2004.
3. J. Guajardo, R. Bluemel, U. Krieger, and C. Paar. Efficient implementation of elliptic curve cryptosystems on the TI MSP430x33x family of microcontrollers. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC 2001)*, 2001.
4. N. Gura, A. Patel, A. Wander, W. Eberle, and S. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems - CHES'2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag, 2004.
5. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.
6. C. Koç, T. Acar, and B. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, pages 26–33, June 1996.
7. ARM Ltd. Arm realview tools. `http://www.arm.com/products/DevTools/`.
8. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, Boca Raton, Florida, 1996. URL: `http://cacr.math.uwaterloo.ca/hac`.
9. P. Montgomery. Modular multiplication without division. *Mathematics of Computation*, 44(170):519–521, 1985.
10. M. Scott. Implementing cryptographic pairings. In *Pairing-Based Cryptography – Pairing '2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 177–196. Springer-Verlag, 2007.
11. MIPS technologies. `http://www.mips.com/products/architectures/smartmips-ase/`.
12. Leif Uhsadel, Axel Poschmann, and Christof Paar. Enabling Full-Size Public-Key Algorithms on 8-bit Sensor Nodes. In *Proceedings of ESAS 2007*, volume 4572 of *LNCS*. Springer, 2007. `http://www.ist-ubisecsens.org/publications/ecc_esas2007.pdf`.
13. Haodong Wang, Bo Sheng, and Qun Li. Elliptic curve cryptography based access control in sensor networks. *International Journal of Security and Networks (IJSN). Special Issue on Security Issues on Sensor Networks*, 1(3/4):127–137, 2006.