Multiprecision multiplication on AVR revisited

Michael Hutter · **Peter Schwabe**

July 31, 2014

Abstract This paper presents new speed records for multiprecision multiplication on the AVR ATmega family of 8-bit microcontrollers. For example, our software takes only 1976 cycles for the multiplication of two 160-bit integers; this is more than 15% faster than previous work. For 256-bit inputs, our software is not only the first to break through the 6000-cycle barrier; with only 4797 cycles it also breaks through the 5000-cycle barrier and is more than 21% faster than previous work. We achieve these speed records by carefully optimizing the Karatsuba multiplication technique for AVR ATmega. One might expect that subquadratic-complexity Karatsuba multiplication is only faster than algorithms with quadratic complexity for large inputs. This paper shows that it is in fact faster than fully unrolled product-scanning multiplication already for surprisingly small inputs, starting at 48 bits. Our results thus make Karatsuba multiplication the method of choice for high-performance implementations of elliptic-curve cryptography on AVR ATmega microcontrollers.

Keywords Karatsuba multiplication · microcontroller · ATmega.

This work was supported by the Austrian Science Fund (FWF) under the grant number TRP251-N23, by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114, and by the European Cooperation in Science and Technology (COST) Action IC1204 (Trustworthy Manufacturing and Utilization of Secure Devices - TRUDEVICE). Part of the work was done while the authors visited Academia Sinica. They wish to thank Bo-Yin Yang for his hospitality. Permanent ID of this document: `102fe77c6d1003e5694ac04543a52410`.

Michael Hutter
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A-8010, Graz, Austria
E-mail: michael.hutter@iaik.tugraz.at

Peter Schwabe
Radboud University Nijmegen
Digital Security Group
PO Box 9010, 6500GL Nijmegen, The Netherlands
E-mail: peter@cryptojedi.org

1 Introduction

How much effort is it to multiply two integers? Over the last 6 decades, many researchers have attempted to answer this question. One main line of research is concerned with the *asymptotic complexity* of integer multiplication. In the 1950s, Kolmogorov conjectured that the complexity for the multiplication of two n-digit integers is $\Theta(n^2)$. This conjecture was proven wrong by one of his students, Karatsuba, in 1960 who presented a multiplication algorithm with asymptotic complexity $\Theta(n^{\log_2 3})$ This ground-breaking result was published in 1962 [13]. See [14, Section 6] for a history
$2^{\log}$

ATmega architecture. We do not claim novelty for any particular technique we used. What is new is the combination of techniques and careful hand-optimization of multiprecision multiplication for AVR ATmega microcontrollers.

Notes on the naming. While working on this paper we observed that the term "schoolbook multiplication" has a different meaning for different people and in different contexts. Sometimes, it only refers to operand-scanning multiplication. Other techniques with quadratic complexity, such as product-scanning multiplication, hybrid multiplication [8], or operand-caching multiplication [11] are not considered to be "schoolbook". For an example of this naming convention, see [11, Section 3]. However, when distinguishing multiplication algorithms with different asymptotic complexities, "schoolbook multiplication" is often used to refer to any quadratic-complexity algorithm. See, for example, [8, Section 3]. To avoid confusion we avoid the term "schoolbook multiplication" throughout this paper.

It is common to refer to the product-scanning technique as "Comba multiplication", and to give credit to a 1990 paper by Comba [3]. See, for example, [5, 11, 21]. However, the technique has earlier been described (without a claim of novelty) by Barrett in [1, Diagram three]. The method has in fact already been described by Leonardo Pisano (Fibonacci) in his work "Liber Abaci" from 1202; see [24, Chapter 2]. Swetz in [25, Chapter 4] states that the "cross method of multiplication" can be traced back to the *Līlāvatī* by Bhāskara from 1150, but we were not able to confirm this in the English translation by Patwardhan, Naimpally, and Singh [19]. We will use the term "product-scanning multiplication".

Related work. Many results exist on fast multiprecision multiplication on embedded processors, often in the context of modular arithmetic and elliptic-curve cryptography. Some papers also consider the Karatsuba technique for multiplication on embedded processors. For example, Großschädl, Avanzi, Savaş, and Tillich use the Karatsuba technique for fast and energy-efficient multiplication of 512-bit and larger integers on StrongARM [7]; Gouvêa and López use Karatsuba for 256-bit multiplication on the MSP430 [5]; Gouvêa, Oliveira, and López use it for fast 256-bit multiplication on the MSP430X [6].

On AVR ATmega microcontrollers the state of the art in multiplication of integers of up to 256 bits has consistently been held by algorithms with quadratic complexity. Until 2004 the fastest known algorithm was product-scanning multiplication. For inputs of size larger than 96 bits this changed with the introduction of *hybrid multiplication* by Gura, Patel, Wander, Eberle and Chang Shantz in [8]. This algorithm was later improved by Scott and Szczechowiak in [21]. The next milestone in optimizing multiprecision multiplication on AVR was the best paper of CHES 2011 by Hutter and Wenger that introduced *operand caching* multiplication [11]. The results of that paper were slightly improved in two followup papers by Seo and Kim, which introduced and optimized *consecutive operand caching* [22, 23]. These papers mark the current state of the art in multiprecision multiplication on AVR ATmega microcontrollers. Note that the use of hybrid multiplication is covered by a patent [9]; a patent for operand-caching multiplication is pending [12].

We are aware of only two papers considering the Karatsuba technique for multiplication of big integers on AVR ATmega. Both papers conclude that Karatsuba multiplication is slower than quadratic-complexity multiplication algorithms for input sizes commonly used in elliptic-curve cryptography. In [17], Liu, Großschädl, and Kizhvatov consider different approaches for implementing Montgomery multiplication including "Hybrid Karatsuba-Comba Multiplication (HKCM)". They conclude in Section III.B that

> *"The HKCM method is faster than HSOS (and also HFIPS) for operands exceeding 512 bits in length, but slower in the case of 256-bit operands.* [...] *For 512-bit operands, the HKCM method achieves essentially the same performance as HSOS, and both are roughly 7% faster than HFIPS."*

Both "Hybrid Separated Operand Scanning" (SOS) and "Hybrid Finely Integrated Product Scanning" (HFIPS) are algorithms with quadratic complexity.

In [10], we used Karatsuba for multiplication of 256-bit integers; however, with 6686 cycles, that approach turned out to be considerably slower than state-of-the-art operand-caching and consecutive-operand-caching multiplication.

Availability of Software. We placed all software described in this paper into the public domain[1]. We will not apply for any patents for the techniques described in this paper.

Organization of the paper. Section 2 briefly reviews the specifics of the AVR ATmega family of microcontrollers. Section 3 first considers efficient approaches for small multiprecision multiplication, then discusses different approaches for implementing Karatsuba multiplication on AVR ATmega, and finally derives a lower bound on the number of cycles purely from arithmetic instructions. Section 4 describes how we minimize the number of loads and stores in Karatsuba multiplication for different input sizes to translate the lower computational complexity to lower cycle counts. In Section 4.2 we present detailed performance results of our software and compare with the best results from the literature. We conclude the paper and give ideas for future work in Section 6.

[1] The source code is available at `http://cryptojedi.org/crypto/#avrmul` and at `http://mhutter.org/research/avr/#karatsuba`.

2 The AVR ATmega architecture

This paper optimizes Karatsuba multiplication for the AVR ATmega family of 8-bit microcontrollers. Many of the techniques we describe apply in a similar way on other architectures, but the concrete application of these techniques and the cost analysis is specific to the 8-bit AVR ATmega architecture. This section briefly reviews the specifics of this architecture that are relevant to the remainder of the paper.

Register set. The AVR has 32 registers labeled `R0`,..., `R31`. The register pair (`R26`,`R27`) is aliased as `X`, the register pair (`R28`,`R29`) is aliased as `Y`, and the the register pair (`R30`,`R31`) is aliased as `Z`. These three register pairs are the only ones that can hold the address argument of a load or store instruction. The register pair (`R0`,`R1`) is special because it holds the output of a multiplication instruction (see below).

Memory access. All load and store instructions on the ATmega take 2 cycles. The `LD` load instruction and the `ST` store instruction access memory at the address specified in their argument (either `X`, `Y`, or `Z`). They can post-increment or predecrement their 2-register argument for free. The `LDD` load instruction takes a constant offset to the address register as second argument; so does the `STD` store instruction.

The standard way to use the stack is to use the instructions `PUSH` and `POP`. However, it is also possible to use two `IN` instructions to copy the stack pointer into one of the address register pairs `X`, `Y`, or `Z` and then operate on the stack with `LD`/`LDD` and `ST`/`STD` instructions. Writing back the stack pointer takes two `OUT` instructions

Arithmetic instructions. Our software makes use of only relatively few arithmetic instructions. Most important is the `MUL` instruction, which multiplies the 8-bit unsigned integers in its two register arguments; the 16-bit result is written to the register pair (`R0`,`R1`). The `MUL` instruction takes 2 cycles and it overwrites the carry flag. Addition (`ADD`/`ADC`), subtraction (`SUB`,`SBC`), and exclusive or (`EOR`) are two-operand instructions; one of their inputs is overwritten by the output. For subtraction it is always the minuend that is overwritten. Another helpful instruction is `SBCI` which performs subtraction with borrow of an immediate value from a register. There is no equivalent "`ADCI`" instruction to perform addition with carry of an immediate value. The `CLR` instruction sets a register to zero; the `MOV` instruction copies the value in one register to another register. The `MOVW` instruction copies a register pair to another register pair. Note that two adjacent registers are a register pair only if the lower register is "even" (i.e., `R0`, `R2`, ...). It is worth noting that `MOVW`, like all other arithmetic instructions except `MUL`, takes only one cycle.

Aside from the typical flags (like carry, zero, etc.), the AVR also features a T flag, which can be used to remember a single bit. The `BST` instruction stores one bit of a given register to the T flag, the `BLD` instruction loads from the T flag into one bit of a given register. It is possible to perform conditional branches depending on the value of the T flag.

C function-call ABI. The `avr-gcc` function-call ABI specifies that the first three 16-bit arguments (e.g., pointers) are passed in register pairs (`R24`,`R25`), (`R22`,`R23`), and (`R20`,`R21`). It furthermore specifies that registers `R2`–`R17`, `R28`, and `R29` are caller registers, and that register `R1` has to be set to zero before returning from a function. Our software follows these conventions to make it directly usable from C code, but as in previous papers we do not include function-call-ABI related overhead in our cycle counts. See Section 4.2.

3 Arithmetic considerations

In this section we consider the pure arithmetic cost, i.e., ignoring costs for loads and stores, of Karatsuba multiplication on AVR ATmega. We start with fixing the representation of big integers and reviewing the arithmetic cost of small multiprecision multiplications to establish a baseline.

Representation of big integers. Throughout this paper we will represent big integers in unsigned radix-2^8 representation, i.e., an $8m$-bit integer A is represented in m bytes $(a_0,\ldots,a_{m-1})$ with $A = \sum_{i=0}^{m-1} a_i 2^{8i}$ and $a_i \in \{0,\ldots,255\}$. This big-integer representation is standard for AVR throughout the literature. We do not expect any benefits from using a signed representation or a "carry-safe" representation, which leaves some bits on the top of each limb free to accumulate carries.

Small multiprecision multiplications. Karatsuba multiplication, like hybrid multiplication, constructs full-size multiprecision multiplication from blocks of smaller multiplications. The block sizes that are most relevant for this paper are 24×24 bits, 32×32 bits, 40×40 bits, and 48×48 bits. One obvious way to handle those "small multiprecision" multiplications is to use operand scanning or product scanning. However, this is not optimal as demonstrated in the context of inner-loop optimization for the hybrid multiplication by Lederer, Mader, Koschuch, Großschädl, Szekely, and Tillich in [15], by Liu, Großschädl, and Kizhvatov in [17], and most recently by Liu and Großschädl in [16].

For 32×32-bit multiplication we adapted the technique described in [16, Section 3.1] for multiplication (the original algorithm performs multiply-accumulate). Inspired by this technique we wrote similar routines for 24×24-bit, for 40×40-bit, and for 48×48-bit multiplication. Table 1 lists instruction and cycle counts for those small multiplications; the corresponding code listings are in Appendix B. Note that our routines are slightly different from the one by Liu and Großschädl in the sense that they can be seen as tweaked operand scanning. We assume that inputs are already loaded

Table 1 Instruction and cycle counts for small multiprecision multiplications without loads and stores. Corresponding counts of fully unrolled product-scanning multiplication are listed in parentheses.

Input size	`MUL`	`ADD/ADC`	`MOV (W)`	`CLR`	cycles
24×24 bits	9	17	4	3	42
	(9)	(22)	(1)	(5)	(46)
32×32 bits	16	35	6	3	76
	(16)	(43)	(1)	(7)	(83)
40×40 bits	25	58	8	2	118
	(25)	(70)	(1)	(9)	(130)
48×48 bits	36	83	15	2	172
	(36)	(103)	(1)	(11)	(187)

to registers and that outputs are also kept in registers. The cost for loads and stores depends on the context in which these multiplication blocks are used.

We do not claim speed records for these small multiplications, although we are not aware of any faster results. We would expect that there exist thoroughly optimized routines for these input sizes that are in the range of standard C data types. We were surprised to find the currently fastest approach somewhat "hidden" as an inner-loop optimization of big-integer hybrid multiplication in a paper on Montgomery modular multiplication.

Note that the optimized small multiprecision multiplications need slightly more live registers than fully unrolled product-scanning multiplications. Whether they are better than product scanning or not depends on the context, i.e., the amount of registers that are available without spilling.

Additive vs. subtractive Karatsuba. From now on we are considering $n \times n$-byte multiplication, where n is even and $k = n/2$. The typical way to describe Karatsuba multiplication of an n-byte integer $A \hat{=} (a_0, \ldots, a_{n-1})$ and and n-byte integer $B \hat{=} (b_0, \ldots, b_{n-1})$ is the following:

- Write $A = A_\ell + 2^{8k} A_h$ and $B = B_\ell + 2^{8k} B_h$ for k-byte integers A_ℓ, A_h, B_ℓ, and B_h;
- compute $L = A_\ell \cdot B_\ell \hat{=} (\ell_0, \ldots, \ell_{n-1})$;
- compute $H = A_h \cdot B_h \hat{=} (h_0, \ldots, h_{n-1})$;
- compute $M = (A_\ell + A_h) \cdot (B_\ell + B_h) \hat{=} (m_0, \ldots, m_n)$; and
- obtain the result as $A \cdot B = L + 2^{8k}(M - L - H) + 2^{8n} H$.

We will refer to this approach as *additive Karatsuba*. The problem with this approach is that the additions of two k-byte numbers $A_\ell + A_h$ and $B_\ell + B_h$ produce carry bits. An efficient way to handle multiplications by such a carry bit during the computation of M is to perform a subtraction-with-carry from a zero register to produce a register that is either `0xff` (if the carry is one) or zero and then compute multiplication through an `AND` instruction with this register. Subsequent accumulation of the one-byte result of such a multiplication costs only two addition instructions (one `ADD` and one `ADC`) instead of three instructions for two-byte results.

The problem with this approach is twofold: First the multiplications by carry bits still contribute a significant overhead. Second the tweak to use `AND` instructions only works for a single carry bit. Recursive application of Karatsuba's technique yields multiple carry bits which have to be handled by full multiplication and accumulation. It turns out to be more efficient to use *subtractive Karatsuba*:

- Write $A = A_\ell + 2^{8k} A_h$ and $B = B_\ell + 2^{8k} B_h$ for k-byte integers A_ℓ, A_h, B_ℓ, and B_h;
- compute $L = A_\ell \cdot B_\ell \hat{=} (\ell_0, \ldots, \ell_{n-1})$;
- compute $H = A_h \cdot B_h \hat{=} (h_0, \ldots, h_{n-1})$;
- compute $M = |A_\ell - A_h| \cdot |B_\ell - B_h| \hat{=} (m_0, \ldots, m_{n-1})$;
- set $t = 0$, if $M = (A_\ell - A_h) \cdot (B_\ell - B_h)$; $t = 1$ otherwise;
- compute $\hat{M} = (-1)^t M = (A_\ell - A_h)(B_\ell - B_h)$ $\hat{=} (\hat{m}_0, \ldots, \hat{m}_{n-1})$; and
- obtain the result as $A \cdot B = L + 2^{8k}(L + H - \hat{M}) + 2^{8n} H$.

This variant of Karatsuba avoids the carry bits in the computation of M but instead needs to compute two absolute differences $|A_\ell - A_h|$ and $|B_\ell - B_h|$ and one conditional negation of M. This has to be done in constant time to make the multiplication routine suitable for timing-attack-protected implementations of cryptographic primitives.

Constant-time absolute differences. We compute $|A_\ell - A_h|$ as follows: First perform a subtraction of $A_\ell - A_h$ which costs k subtraction instructions. Then we use a subtract-with-carry of a register from itself to obtain a register with the value $t_A =$ `0xff` if $A_\ell < A_h$ or $t_A = 0$ otherwise. We then xor t_A to all k result registers of the subtraction $A_\ell - A_h$. If $t_A = 0$, this does not change anything; if $t_A =$ `0xff`, this produces the ones' complement. We then negate t_A (obtaining either $t_A = 1$ or $t_A = 0$), add it to the lowest of the k registers and ripple the carry through to obtain the two's complement. The whole computation costs $k+1$ `SUB/SBC` instructions, k `EOR` instructions, one `NEG` instruction, and k `ADD/ADC` instructions adding up to a total of $3k+2$ instructions accounting for $3k+2$ cycles. The computation of $|B_\ell - B_h|$ is done in the same way. We obtain the value of t required for the conditional negation of M as $t = t_A \oplus t_B$.

Constant-time conditional negation. The most obvious way to compute $L + H - \hat{M}$, given M, is to use a conditional branch that either adds or subtracts M, depending on the value of t. Note that the `EOR` instruction which we use to compute t sets the zero flag, which we can then use for the branch condition. On many platforms such a conditional branch would inevitably create a timing leak. The AVR does not have any branch-prediction mechanisms and we can balance the time taken in each of the two branches through `NOP` instructions to eliminate timing leaks. We implemented this approach and refer to it as the "branched" approach in the following.

There are multiple reasons to avoid branches in cryptographic software. In our port of NaCl to the AVR architec-

ture described in [10], we avoid all secret-data-dependent branches primarily because of the fact that reviewing NOP-balanced branches for timing leaks is tedious work and argued that avoiding such branches incurs only small penalties. Furthermore, secret-data-dependent branches are often an easy target for *safe-error attacks*. See, for example, Yen and Joye who described these attacks in [27]. A careful analysis of different multiplication methods from a side-channel point of view is outside the scope of this paper, but we believe that eliminating secret-data-dependent branches is generally a good practice.

An alternative, branch-free way to perform conditional negation is to use the same technique that we used for constant-time absolute differences above (without the initial subtraction). The additions required to convert from the ones' complement to the two's complement can be merged with the additions that are required to combine the partial results; we simply move the bit to the carry flag and replace one ADD instruction by an ADC instruction.

We recommend the branch-free approach for applications that handle secret data and the slightly faster branched approach for applications that do not handle secret data, e.g., signature verification.

Refined Karatsuba multiplication. Combining the partial results in the last step as $L+2^{8k}(L+H-\hat{M})+2^{8n}H$ looks like two n-byte additions and one n-byte subtraction plus rippling a carry bit to the end. However, observe that the byte at position k of the result is obtained as $r_k = \ell_k - \hat{m}_0 + \ell_0 + h_0$; the byte at position n is obtained as $r_n = h_0 - \hat{m}_k + \ell_k + h_k$. What looks like 4 additions and 2 subtractions can be reduced to 3 additions and 2 subtractions by precomputing $s = h_0 + \ell_k$ and then obtaining $r_k = \ell_0 + s - \hat{m}_0$ and $r_n = h_k + s - \hat{m}_k$. The same trick applies to r_{k+1} and r_{n+1} and so on and saves a total of k additions. We learned this trick from a Crypto 2009 paper by Bernstein [2, Section 2].

An additional advantage of refined Karatsuba is that we can merge the additions of $h_0+\ell_k$, $h_1+\ell_{k+1}$ etc. into the multiplication $H = A_h \cdot B_h$. This is not an advantage from the point of view of purely arithmetic cost, but it simplifies register allocation as explained in Section 4. Note that $\bar{H} = H + (\ell_k, \ldots, \ell_{n-1})$ cannot overflow, the result fits into n bytes.

However, there is also a slight disadvantage of merging this accumulation of $(\ell_k, \ldots, \ell_{n-1})$. The carry bit that may result from the accumulation is immediately rippled into $h_k, \ldots, h_{n-1}$. Later we add $(\ell_0, \ldots, \ell_{k-1}, h_k, \ldots, h_{n-1})$ into the result with an offset of k bytes and subtract $(\hat{m}_0, \ldots, \hat{m}_{n-1})$ with the same offset. The addition may produce a carry bit c which needs to be rippled to the end; the subtraction may produce a borrow bit b which needs to be rippled to the end. One can also think of this as a carry bit $d = b + c$ which is either 0, 1, or -1; The fact that this carry bit can be negative is a direct consequence of merging the addition of $(\ell_k, \ldots, \ell_{n-1})$ into the multiplication $H = A_h \cdot B_h$ and rippling the resulting carry. The non-merged computation of $(\ell_0, \ldots, \ell_{n-1}) + (h_0, \ldots, h_{n-1}) - (\hat{m}_0, \ldots, \hat{m}_{n-1})$ would always produce a non-negative carry, which can simply be rippled to the end.

Merging carries and borrows. If we independently rippled a carry bit $c \in \{0,1\}$ and a borrow bit $b \in \{-1,0\}$ to the end of the result, we would essentially lose the arithmetic benefit of refined Karatsuba. What we do instead is to first compute c, then, after subtraction of $(\hat{m}_0, \ldots, \hat{m}_{n-1})$, use an SBCI of zero from c to obtain $d \in \{-1,0,1\}$ and set the borrow bit if and only if $d = -1$. We then clear another register f and perform an SBC from the same register to clear the content of f and to obtain $f \in \{-1,0\}$ depending on the value of d. Now the register pair (d, f) contains $(-1,-1)$, $(0,0)$ or $(1,0)$. In the case of the branched-free approach, we first merge c and b into d and then perform a MOV operation of d into f and apply an ASR instruction afterwards, which logically shifts d to the right, resulting in $f \in \{-1,0\}$. After that, we can ripple the carry to the end of the result through one addition of d and then subsequent additions-with-carry (ADC instructions) of f.

Putting it together. The overall arithmetic cost of (branched) Karatsuba multiplication on AVR is thus composed of the following parts:

- One CLR instruction to produce a zero register
- The cost of computing L (multiplication of two k-byte integers);
- the cost of computing M (multiplication of two k-byte integers);
- the cost of computing $\bar{H} = H + (\ell_k, \ldots, \ell_{n-1})$ (essentially the cost a k-byte integer multiplication and k addition instructions);
- $2k+2$ SUB/SBC instructions, $2k$ EOR instructions, 2 NEG instructions, and $2k$ ADD/ADC instructions to compute two absolute differences $|A_\ell - A_h|$ and $|B_\ell - B_h|$;
- $n+1$ ADD/ADC instructions in order to add $(\ell_0, \ldots, \ell_{k-1}, h_k \ldots, h_{n-1})$ to the result and to remember the carry bit;
- one EOR instruction to compute t and to set the zero flag if $t = 0$;
- one BRNE instruction;
- if the branch is not taken (1 cycle for BRNE):
  - $n+2$ SUB/SBC instructions to subtract M and to produce the carry register pair (d, f);
  - one RJMP instruction (2 cycles);
- if the branch is taken (2 cycles for BRNE):
  - $n+1$ ADD/ADC instructions and one CLR instruction to add M and to produce the carry register pair (d, f);
  - one NOP instruction;
- k ADD/ADC instructions to ripple the carry in (d, f) to the end.

In the example of multiplying two 48-bit integers (i.e., $k = 3$, see also Appendix A) the computation of L and M costs 40 cycles each (cf. Table 1, the cost is slightly lower because we can replace some `CLR` instructions by `MOVW` from a zero register pair; this becomes more efficient for multiple multiplications). The computation of $\bar{H} = H + (\ell_3, \ldots, \ell_5)$ costs 44 cycles. Overall we obtain a cost of 169 cycles from arithmetic (and branch) instructions. This is 18 cycles faster than fully unrolled product-scanning multiplication and 3 cycles faster than our optimized 48-bit multiplication. Note that the overhead from loads and stores in this case is the same for all three approaches: 12 loads of input bytes and 12 stores of outputs; 48-bit Karatsuba multiplication does not need any spills as detailed in Section 4. The 3-cycle gain is small and probably of merely theoretic interest (in particular because Karatsuba multiplication requires more registers), but the gain becomes larger for bigger inputs.

4 Efficient scheduling for Karatsuba multiplication

As shown in the previous section, Karatsuba multiplication needs fewer *arithmetic* instructions than, e.g., fully unrolled product scanning already for very small input sizes. However, it is yet unclear how this arithmetic cost translates to an overall cost including the cost for loads and stores. This section explains our strategies to make efficient use of the available registers and the specifics of the AVR instruction set to keep the overhead from load and store instructions low.

These strategies consist of two levels of optimizations. First we use carefully tuned instruction scheduling that minimizes the number of live registers throughout the whole Karatsuba multiplication. Second we use various techniques to avoid costly loads and stores for the cases where not sufficient registers are available despite smart scheduling. Some of these techniques slightly increase the number of arithmetic instructions; the total number of cycles required for Karatsuba multiplication can thus not be obtained by adding the *lower bound* on arithmetic instructions derived in Section 3 to the memory-access overhead explained here. The complete cycle counts for multiprecision multiplication on AVR are reported in Section 4.2. All instruction counts in this section refer to the branched variant of our software.

4.1 One level of Karatsuba

For multiplications with input sizes of 48, 64, 80, and 96 bits we use 1 level of Karatsuba. Our approach to scheduling the computations for 1-level Karatsuba multiplication with effects on register usage is detailed in Algorithm 1. Note that the number of register stated in this algorithm is ignoring some registers, specifically,

- a zero register required to accumulate carries,
- registers to hold the borrows from the subtractions in Step 5,
- registers `R0` and `R1` which hold the result of multiplication instructions,
- accumulation registers in the multiplications in Steps 2, 6, and 7,
- two registers required to ripple the carry or borrow to the end in Step 11.

Even with these additional registers taken into account, refined Karatsuba multiplication of 48-bit inputs and 64-bit inputs does not require any load and store instructions beyond loading inputs once and storing the result once. What is crucial to make this possible for the 64-bit input case is the computation of $\bar{H}$, i.e., that we accumulate $(\ell_k, \ldots, \ell_{n-1})$ on the fly during the multiplication $A_h \cdot B_h$. This is possible because we use refined Karatsuba; without this approach the $6k$ registers would increase to $7k$ registers and all input sizes starting from 64 bits would need significantly more load and store instructions.

For 80-bit and 96-bit multiplications we cannot entirely avoid memory access beyond loading inputs and storing the result. In the following we describe the techniques we use to keep the overhead from these additional loads and stores as small as possible.

Reload $\ell_0, \ldots, \ell_{k-1}$. Spilling register contents to stack costs 4 cycles per register, 2 cycles for the store (`PUSH`) instruction and 2 cycles for the load (`POP`) instruction. We avoid such spills as much as possible by re-loading values that had to be stored anyway as part of the result stores. Specifically, after storing $\ell_0, \ldots, \ell_{k-1}$ in Step 3, we can "forget" the values in the corresponding registers and reload these values again, when they are needed in Step 8. This only costs k load instructions and no additional stores, and reduces the maximal amount of required registers from $6k+2$ to $5k+2$.

Minimize accumulation registers. The multiplications in Steps 2, 6, and 7 need registers to accumulate the result coefficients. For the multiplication $A_\ell \cdot B_\ell$ in Step 2 this is no problem, because the result does not overwrite any of the inputs and simply occupies n "fresh" registers. The optimized versions of small multiprecision multiplications described in Section 3 need two additional registers, but this is also not a problem in Step 2. The situation is different in Steps 6 and 7. When using unrolled product scanning, the result coefficients of the multiplication in Step 6 can overwrite $\ell_k, \ldots, \ell_{n-1}$ with the low half of result coefficients and one of the inputs with the high half of the result coefficients. The multiplication in Step 7 cannot overwrite any registers for the low half of the result (this is why it temporarily needs additional k accumulation registers), but can overwrite input coefficients with the high half of the result.

Algorithm 1 Scheduling and register use for $n \times n$ 1-level Karatsuba multiplication (notation: $k = n/2$).

Input: $A \triangleq (a_0, \ldots, a_{n-1})$ and $B \triangleq (b_0, \ldots, b_{n-1})$, pointers to inputs in register pairs X,Y, pointer to output in register pair Z
Output: $R \leftarrow A \cdot B \triangleq (r_0, \ldots, r_{2n-1})$

1:	Load A_ℓ and $B_\ell \in (0, \ldots, 2^{k-1})$	▷ $2k+6$ live registers:	$a_0, \ldots, a_{k-1}, b_0, \ldots, b_{k-1}$,X,Y,Z
2:	Compute $L \leftarrow A_\ell \cdot B_\ell \triangleq (\ell_0, \ldots, \ell_{n-1})$	▷ $4k+6$ live registers:	$a_0, \ldots, a_{k-1}, b_0, \ldots, b_{k-1}, \ell_0, \ldots, \ell_{n-1}$,X,Y,Z
3:	Store $\ell_0, \ldots, \ell_{k-1}$ to $r_0, \ldots, r_{k-1}$	▷ $4k+6$ live registers:	$a_0, \ldots, a_{k-1}, b_0, \ldots, b_{k-1}, \ell_0, \ldots, \ell_{n-1}$,X,Y,Z
4:	Load A_h and B_h	▷ $6k+2$ live registers:	$a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, \ell_0, \ldots, \ell_{n-1}$,Z
5:	Compute $\lvert A_\ell - A_h \rvert$ and $\lvert B_\ell - B_h \rvert$	▷ $6k+2$ live registers:	$2k$ registers for $\lvert A_\ell - A_h \rvert$ and $\lvert B_\ell - B_h \rvert$, $a_k, \ldots, a_{n-1}, b_k, \ldots, b_{n-1}, \ell_0, \ldots, \ell_{n-1}$, Z
6:	Compute $\bar{H} \leftarrow A_h \cdot B_h + (\ell_k, \ldots, \ell_{n-1})$	▷ $5k+2$ live registers:	$2k$ registers for $\lvert A_\ell - A_h \rvert$ and $\lvert B_\ell - B_h \rvert$, $2k$ registers for $\bar{H}$, $\ell_0, \ldots, \ell_{k-1}$, Z
7:	Compute $M \leftarrow \lvert A_\ell - A_h \rvert \cdot \lvert B_\ell - B_h \rvert$	▷ $5k+2$ live registers:	$2k$ registers for M, $2k$ registers for $\bar{H}$, $\ell_0, \ldots, \ell_{k-1}$, Z (temporarily need $6k+2$ registers during multiplication)
8:	Compute $U \leftarrow (\ell_0, \ldots, \ell_{k-1}, h_k, \ldots, h_{n-1}) + \bar{H}$	▷ $5k+2$ live registers:	$2k$ registers for U, $2k$ registers for M, $h_k, \ldots, h_{n-1}$, Z
9:	Compute $U \leftarrow U + M$ or $U \leftarrow U - M$	▷ $3k+2$ live registers:	$2k$ registers for U, $h_k, \ldots, h_{n-1}$, Z
10:	Store U to $r_k, \ldots, r_{n+k-1}$	▷ $k+2$ live registers:	$h_k, \ldots, h_{n-1}$, Z
11:	Ripple carry/borrow from Steps 8+9 through $h_k, \ldots, h_{n-1}$	▷ $k+2$ live registers:	$h_k, \ldots, h_{n-1}$, Z
12:	Store $h_k, \ldots, h_{n-1}$ to $r_{n+k}, \ldots, r_{2n-1}$		

Overwriting registers that are no longer needed with result coefficients step-by-step is not possible to the same extent with the optimized small multiprecision multiplications. We therefore often use fully unrolled product scanning instead of the optimized multiplication variants in Step 6 and 7 to reduce the number of live register variables.

Using the T flag. An AVR-specific optimization is to make use of the T flag. Specifically, the bit $t = t_A \oplus t_B$, which decides whether we need to add or subtract M, does not need to occupy a register. Instead we can use a BST instruction to store this bit in the T flag and later use a BRTS instruction to branch depending on the value of this bit. The branch-free variant of our software needs to use a BLD instruction to copy this bit back to a register. This is still cheaper than a PUSH and a POP, because writing and reading the T flag costs only 1 cycle each.

Memory access in 1-level Karatsuba on AVR. In total, the 48-bit refined Karatsuba multiplication needs 12 LD/LDD instructions, and 12 ST/STD instructions. The 64-bit multiplication needs 16 LD/LDD instructions, and 16 ST/STD instructions. These instructions are precisely what is needed to load the inputs from memory and to store the result. The 80-bit multiplication needs 25 LD/LDD instructions, and 20 ST/STD instructions. The 96-bit multiplication needs 42 LD/LDD instructions, 24 ST/STD instructions, 4 PUSH instructions, and 4 POP instructions.

4.2 Two levels of Karatsuba

For input sizes of 128, 160, and 192 bits we use two levels of Karatsuba recursion. That means that we use the 1-level Karatsuba multiplication routines described above as building blocks. The general strategy to perform 2-level Karatsuba multiplication is similar to the 1-level Karatsuba multiplications but requires additional spills (PUSH and POP) to the stack. For details of the scheduling and register use, see Algorithm 2. The register usage in this algorithm describes the usage *after* each step and ignores constant overhead; for details inside the respective steps, see Algorithm 1.

We applied the following techniques to improve the performance of 2-level Karatsuba.

Address-pointer handling. For 160 and 192-bit Karatsuba, the input-address pointers have to be spilled to the stack in each 1-level Karatsuba multiplication and they have to be restored from the stack afterwards. This spilling of X and Y is only required for the computation of L and H; after the computation of M the input taddresses are not needed anymore. Spilling would typically require a total of 8 PUSH and 8 POP instructions (i.e., 32 cycles); these are 4 PUSH and 4 POP instructions for each of the two 1-level Karatsuba multiplications. To improve the efficiency, we initially store the address pointers on the stack and load them twice afterwards using two IN, four LDD instructions, and one MOVW instruction. The LDD instructions load the pointer to A from stack into X and the pointer to B into two temporary registers. The MOVW instruction finally copies the pointer from the temporary registers to Y. This saves 5 cycles in total (needing 4 PUSH instructions, 2 IN instructions, 4 LDD instructions, 1 MOVW instruction, and 4 POP instructions).

We further decided to push X and Y right after the loading of A_h in Step 4 in Algorithm 1. There are two reasons for pushing the addresses at this point. First, the X registers already point to the next input address needed in the compu-

Algorithm 2 Scheduling and register use for $n \times n$ 2-level Karatsuba multiplication (notation: $k = n/2$).

Input: $A \hat{=} (a_0, \ldots, a_{n-1})$ and $B \hat{=} (b_0, \ldots, b_{n-1})$, pointers to inputs in register pairs `X`,`Y`, pointer to output in register pair `Z`
Output: $R \leftarrow A \cdot B \hat{=} (r_0, \ldots, r_{2n-1})$

1:	Compute $L \leftarrow A_\ell \cdot B_\ell$ using Algorithm 1	▷ $k+2$ live registers:	$\ell_k, \ldots, \ell_{n-1}$, `Z` (`X` and `Y` are pushed on the stack in Step 4 of Algorithm 1)
2:	Load `X` and `Y` from stack (restore address pointers)	▷ $k+6$ live registers:	$\ell_k, \ldots, \ell_{n-1}$, `X`, `Y`, `Z`
3:	Compute $\bar{H} \leftarrow A_h \cdot B_h + (\ell_k, \ldots, \ell_{n-1})$ using Algorithm 1	▷ 2 live registers:	`Z` (for on-the-fly accumulation of $\ell_k, \ldots, \ell_{n-1}$ into H see correponding paragraph in Section 4.2)
4:	Load `X` and `Y` from stack (restore address pointers)	▷ 6 live registers:	`X`, `Y`, `Z`
5:	Load A and B and compute $\lvert A_\ell - A_h \rvert$ and $\lvert B_\ell - B_h \rvert$	▷ $2k+2$ live registers:	$a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}$, `Z`
6:	Compute $M \leftarrow \lvert A_\ell - A_h \rvert \cdot \lvert B_\ell - B_h \rvert$	▷ $2k+2$ live registers:	$2k$ registers for M, `Z`
7:	Load $\ell_k, \ldots, \ell_{n-1}$	▷ $3k+2$ live registers:	$2k$ registers for M, $\ell_k, \ldots, \ell_{n-1}$, `Z`
8:	Compute $U_\ell \leftarrow (\ell_k, \ldots, \ell_{n-1}) \pm M_\ell$	▷ $3k+2$ live registers:	$2k$ registers for M, k registers for U_ℓ, `Z`
9:	Load $h_0, \ldots, h_{k-1}$	▷ $3k+2$ live registers:	k registers for M_h, k registers for U_ℓ, $h_0, \ldots, h_{k-1}$, `Z`
10:	Compute $U_\ell \leftarrow (h_0, \ldots, h_{k-1}) + U_\ell$	▷ $3k+2$ live registers:	k registers for M_h, k registers for U_ℓ, $h_0, \ldots, h_{k-1}$, `Z`
11:	Store U_ℓ to $r_k, \ldots, r_{n-1}$	▷ $2k+2$ live registers:	k registers for M_h, $h_0, \ldots, h_{k-1}$, `Z`
12:	Compute $U_h \leftarrow (h_0, \ldots, h_{k-1}) \pm M_h$	▷ $2k+2$ live registers:	k registers for M_h, k registers for U_h, `Z`
13:	Load $h_k, \ldots, h_{n-1}$	▷ $2k+2$ live registers:	k registers for U_h, $h_k, \ldots, h_{n-1}$, `Z`
14:	Compute $U_h \leftarrow (h_k, \ldots, h_{n-1}) + U_h$	▷ $2k+2$ live registers:	k registers for U_h, $h_k, \ldots, h_{n-1}$, `Z`
15:	Store U_h to $r_n, \ldots, r_{n+k-1}$	▷ $k+2$ live registers:	$h_k, \ldots, h_{n-1}$, `Z`
16:	Ripple carry/borrow through $h_k, \ldots, h_{n-1}$	▷ $k+2$ live registers:	$h_k, \ldots, h_{n-1}$, `Z`
17:	Store $h_k, \ldots, h_{n-1}$ to $r_{n+k}, \ldots, r_{2n-1}$		

tation of H, so no additional update of the pointer is needed, e.g., using the `ADIW` instruction, which would be needed if we pushed the pointer right before the computation of L. Second, after pushing the address on the stack, the register `X` can be efficiently re-used for storing the input operands of B_h. This avoids additional spilling of registers.

When mixing `LDD` with `PUSH` and `POP`, the stack pointer needs to be corrected again at the end of the computation. This can be done by one `ADIW` instruction and two `OUT` instructions.

On-the-fly accumulation. As in 1-level Karatsuba, and essentially thanks to refined Karatsuba, we perform an on-the-fly accumulation of $(\ell_k, \ldots, \ell_{n-1})$ during the multiplication of $H = A_h \cdot B_h$ in 2-level Karatsuba. Applying this optimization, however, is not as straight forward as in 1-level Karatsuba, because H itself is computed using 1-level Karatsuba. This makes the accumulation and especially the handling of carry bits more complex.

The main idea to avoid the propagation of carry bits from the accumulation of $(\ell_k, \ldots, \ell_{n-1})$ into the multiplication of $A_h \cdot B_h$ in 2-level Karatsuba, is to split the accumulation into two parts of size $k/2$ each. The first part $(\ell_k, \ldots, \ell_{1.5k-1})$ is accumulated into $(h_0, \ldots, h_{k/2-1})$ and the result is stored in memory. After that, one could accumulate the second part $(\ell_{1.5k}, \ldots, \ell_{n-1})$ into $(h_{k/2}, \ldots, h_{n-1})$. However, this would also add an unintended carry into h_k that would also ripple through $h_{k+1}, \ldots, h_{n-1}$. In order to avoid this unintended carry propagation, we accumulate $(\ell_{1.5k}, \ldots, \ell_{n-1})$ again into $(h_0, \ldots, h_{k/2-1})$, which still resides in registers. The advantage of this technique is that this accumulated result is then added correctly only once in Step 8 of Algorithm 1 and the carry is added only once into $h_{k/2}$ and ripples correctly through $h_{k/2+1}, \ldots, h_{n-1}$.

Memory access in 2-level Karatsuba on AVR. In total, the 128-bit refined Karatsuba multiplication needs 92 `LD`/`LDD` instructions, 50 `ST`/`STD` instructions, 2 `PUSH` instructions, and 2 `POP` instructions. The 160-bit multiplication needs 140 `LD`/`LDD` instructions, 80 `ST`/`STD` instructions, 15 `PUSH` instructions, 17 `POP` instructions, and 4 `IN` instructions to copy the stack pointer to `Y`. The 192-bit multiplication needs 241 `LD`/`LDD` instructions, 108 `ST`/`STD` instructions, 46 `PUSH` instructions, 21 `POP` instructions, 8 `IN` instructions, and 2 `OUT` instructions.

4.3 Three levels of Karatsuba

We implemented the 256-bit multiplication using 3-levels of Karatsuba. Due to the high register usage of the 1-level and 2-level Karatsuba blocks there is almost no room to hold and re-use registers. Thus, we store all results obtained from the 2-level Karatsuba multiplications in the memory and load the values again at the end of calculation M. Also all absolute differences are pushed to the stack and are popped again during the final 2-level Karatsuba multiplication. The obtained results for M are also pushed to the stack and are popped again at the end of the multiplication.

Table 2 Speed and size comparison of multiprecision multiplication on AVR ATmega. All counts exclude function-call overhead.

		Input size (bits)							
Approach		48	64	80	96	128	160	192	256
Unrolled product scanning:	cycles	235	395	595	836	—	—	—	—
	bytes	350	598	910	1288	—	—	—	—
Operand caching [11]:	cycles	—	—	—	—	—	2393[a]	3467[a]	6121[a]
	bytes	—	—	—	—	—	3696[a]	5354[a]	9476[a]
Consecutive operand caching [22]:	cycles	—	—	—	—	1532	2356	3464	6180
	bytes	—	—	—	—	N/A	3652	N/A	N/A
(Consecutive) operand caching [23]:	cycles	—	—	—	—	1523[b]	2341[b]	3437	6115
	bytes	—	—	—	—	2346[b]	3622[b]	N/A	N/A
This paper (branched):	cycles	217[c]	360[c]	522[c]	780[c]	1325[d]	1976[d]	2923[d]	4797[e]
	bytes	348[c]	580[c]	828[c]	1228[c]	2228[d]	3222[d]	4602[d]	8022[e]
	stack	0	0	0	4	1	19	36	58
This paper (branch-free):	cycles	222[c]	368[c]	533[c]	800[c]	1369[d]	2030[d]	2987[d]	4961[e]
	bytes	342[c]	576[c]	826[c]	1226[c]	2156[d]	3106[d]	4492[d]	7616[e]
	stack	0	0	0	4	1	19	36	58

[a] Counts obtained using the online code generator
[b] Counts from the software we received from the authors
[c] One level of Karatsuba
[d] Two levels of Karatsuba
[e] Three levels of Karatsuba

In total, the 256-bit multiplication needs the following memory instructions: 352 `LD`/`LDD` instructions, 156 `ST`/`STD` instructions, 82 `PUSH` instructions, 130 `POP` instructions, 8 `IN` instructions, and 32 `OUT` instructions.

5 Results

This section reports cycle counts, code size, and stack usage for the software presented in this paper. All cycle counts are obtained through simulation in the Atmel AVR Studio version 5.0.1223. All multiplication routines passed tests on 1000 random inputs and passed a test with all input bytes set to 255. These tests were performed on an ATmega2560 (Arduino MEGA development board).

Like previous papers we report cycle counts, code sizes, and stack usage excluding the function-call cost, i.e., the cost for `CALL`, `RET`, initial `PUSH` and final `POP` of caller registers, 3 `MOVW` instructions required to copy the function arguments to the `X`, `Y`, `Z` registers, and the cost to clear register `R1` before returning from the function.

It is important to note that for small input sizes, product scanning does not use all available registers and can avoid some of the `PUSH`s and `POP`s of caller registers. A function that only multiplies, e.g., two 48-bit integers and follows the C function-call ABI for AVR will thus be faster when using product scanning than our Karatsuba multiplication. However, the 48-bit Karatsuba multiplication will be faster if it is used in a larger (inlined) context. See also Section 4.

A summary of our results, together with the best previous results from the literature, is presented in Table 2. All implementations listed in this table focus on speed and are fully unrolled. For input sizes from 48 to 96 bits we are not aware of any results from the literature achieving better speeds than fully unrolled product-scanning multiplication. For those input sizes we include a comparison with fully unrolled product scanning. For 48-bit inputs this is not optimal as demonstrated by our optimized multiplication routine (see Section 3 and Appendix B). We believe that also for 64-bit, 80-bit, and 96-bit inputs, careful optimization of quadratic-complexity multiplication can gain a few cycles compared to fully unrolled product scanning. However, we do not expect those gains to be larger than what we gain by using subquadratic-complexity Karatsuba multiplication.

The software described in [11] is available through an online code generator at `http://mhutter.org/research/avr#mulopcache`. The cycle counts of the software generated by this online tool are slightly lower than the ones reported in the paper. We compare to the improved cycle counts. From the authors of [23], we received 128-bit and 160-bit consecutive-operand-caching multiplication routines, which are slightly faster than the numbers listed in their paper. We also compare to the improved cycle counts.

6 Conclusion and future work

In this paper we presented new speed records for multiplication of integers from 48 bits up to 256 bits on AVR ATmega.

We showed that carefully optimized Karatsuba multiplication technique is more efficient than quadratic-complexity multiplication already for much smaller input sizes than previously believed.

The most obvious future work is to apply the multiplication routines described in this paper to elliptic-curve cryptography. For example, in [18], Liu, Seo, Großschädl, and Kim use consecutive operand-caching multiplication to push the performance boundaries for arithmetic on the NIST-P192 curve. It will be straight-forward to push the boundaries even further by replacing consecutive operand-caching multiplication by our Karatsuba multiplication routines.

Furthermore, this paper focuses on *speed* of multiplication routines without considering the size of the implementation. It will be interesting to investigate tradeoffs between speed and size for Karatsuba multiplication on AVR, for example by implementing the small multiprecision multiplications at the bottom of the recursion only once and use jumps or calls to this routine. Another direction of future research is to examine whether the Karatsuba technique can also speed up squaring on AVR. Finally, we hope that the techniques described in this paper will serve as an inspiration to re-examine possible performance gains from Karatsuba multiplication for relatively small inputs on other embedded platforms.

References

1. Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag Berlin Heidelberg, 1987. 2
2. Daniel J. Bernstein. Batch binary Edwards. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer-Verlag Berlin Heidelberg, 2009. http://cr.yp.to/papers.html#bbe. 5
3. Paul G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4), 1990. http://lyle.smu.edu/~seidel/courses/cse8351/papers/CombaCRYPTO.pdf. 2
4. Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. 1
5. Conrado P. L. Gouvêa and Julio López. Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In Nicolas Sendrier Bimal Roy, editor, *Progress in Cryptology – INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 248–262. Springer-Verlag Berlin Heidelberg, 2009. http://conradoplg.cryptoland.net/files/2010/12/indocrypt09.pdf. 2
6. Conrado P. L. Gouvêa, Leonardo B. Oliveira, and Julio López. Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *Journal of Cryptographic Engineering*, 2(1), 2012. http://conradoplg.cryptoland.net/files/2010/12/jcen12.pdf. 2
7. Johann Großschädl, Roberto M. Avanzi, Erkay Savaş, and Stefan Tillich. Energy-efficient software implementation of long integer modular arithmetic. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer-Verlag Berlin Heidelberg, 2005. www.iacr.org/archive/ches2005/006.pdf. 2
8. Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In Marc Joye, editor, *Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag Berlin Heidelberg, 2004. www.iacr.org/archive/ches2004/31560117/31560117.pdf. 2
9. Nils Gura and Lawrence A. Spracklen. Hybrid multi-precision multiplication. United States Patent 7650374, 2010. Application filed Nov. 23, 2004, http://www.freepatentsonline.com/7650374.html. 2
10. Michael Hutter and Peter Schwabe. NaCl on 8-bit AVR microcontrollers. In Amr Youssef and Abderrahmane Nitaj, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer-Verlag Berlin Heidelberg, 2013. http://cryptojedi.org/papers/#avrnacl. 2, 5
11. Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer-Verlag Berlin Heidelberg, 2011. https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=58138. 2, 9
12. Michael Hutter and Erich Wenger. Multiplication of large operands. WIPO Patent Application WO/2013/044276, 2013. Application filed Sep.27, 2011, http://www.freepatentsonline.com/WO2013044276A1.html. 2
13. Anatolii Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962. Scanned version on http://cr.yp.to/bib/1963/karatsuba.html. 1
14. Anatolii A. Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics*, 211:169–183, 1995. http://www.ccas.ru/personal/karatsuba/divcen.pdf. 1
15. Christian Lederer, Roland Mader, Manuel Koschuch, Johann Großschädl, Alexander Szekely, and Stefan Tillich. Energy-efficient implementation of ECDH key exchange for wireless sensor networks. In Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practice*, volume 5746 of *Lecture Notes in Computer Science*, pages 112–127. Springer-Verlag Berlin Heidelberg, 2009. http://www.cs.bris.ac.uk/~tillich/pdf/Lederer2009Energy-EfficientImplementation.pdf. 3
16. Zhe Liu and Johann Großschädl. New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers. Cryptology ePrint Archive, Report 2013/882, 2013. https://eprint.iacr.org/2013/882/. 3
17. Zhe Liu, Johann Großschädl, and Ilya Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things – SECIoT'10*, 2010. https://www.nics.uma.es/seciot10/files/pdf/liu_seciot10_paper.pdf. 2, 3
18. Zhe Liu, Hwajeong Seo, Johann Großschädl, and Howon Kim. Efficient implementation of NIST-compliant elliptic curve cryptography for sensor nodes. In Sihan Qing, Jianying Zhou, and Dongmei Liu, editors, *Information and Communications Security*, volume 8233 of *Lecture Notes in Computer Science*, pages 302–317. Springer-Verlag Berlin Heidelberg, 2013. http://orbilu.uni.lu/bitstream/10993/12934/1/ICICS2013.pdf. 10

19. Krishnaji S. Patwardhan, Somashekara A. Naimpally, and Shyam L. Singh. *Līlāvtī of Bhāskarācārya*. Motilal Banarsidass Publishers, 2001. http://books.google.com/books?id=AoX5q7JjM2kC. 2
20. Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, 1971. 1
21. Michael Scott and Piotr Szczechowiak. Optimizing multiprecision multiplication for public key cryptography. Cryptology ePrint Archive, Report 2007/299, 2007. https://eprint.iacr.org/2007/299/. 2
22. Hwajeong Seo and Howon Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In Dong Hoon Lee MotiYung, editor, *Information Security Applications*, volume 7690 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag Berlin Heidelberg, 2012. http://isaa.sch.ac.kr/wisa2012/%EB%85%BC%EB%AC%B8/Session%202/1-130_Multi-precision%20Multiplication%20for%20Public-Key%20Cryptography%20on%20Embedded%20Microprocessors.pdf. 2, 9
23. Hwajeong Seo and Howon Kim. Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors. *International Journal of Computer and Communication Engineering*, 2(3), 2013. http://www.ijcce.org/papers/183-J034.pdf. 2, 9
24. Laurence E. Sigler. *Fibonacci's Liber Abaci – Leonardo Pisano's Book of Calculation*. Springer-Verlag New York, 2003. http://books.google.com/books?id=PilhoGJeKBUC. 2
25. Frank J. Swetz. *Capitalism and Arithmetic: The New Math of the 15th Century*. Open Court, 1987. 2
26. Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. www.de.ufpe.br/~toom/my-articles/engmat/MULT-E.PDF. 1
27. Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49:967–970, 2000. http://pdf.aminer.org/001/082/827/checking_before_output_may_not_be_enough_against_fault_based.pdf. 5

A Karatsuba multiplication of two 48-bit numbers

Listing 1 Karatsuba multiplication of 48-bit integer at address `X` and 48-bit integer at address `Y`; 96-bit result is written to address `Z`

```
CLR R22
CLR R23
MOVW R12,R22
MOVW R20,R22

;--- load a0..a2 and b0..b2 ---
LD R2, X+
LD R3, X+
LD R4, X+
LDD R5, Y+0
LDD R6, Y+1
LDD R7, Y+2

;--- Compute L ---
MUL R2, R7 ;a0*b2
MOVW R10, R0
MUL R2, R5 ;a0*b0
MOVW R8, R0
MUL R2, R6 ;a0*b1
ADD R9, R0
ADC R10, R1
ADC R11, R23
MUL R3, R7 ;a1*b2
MOVW R14, R0
MUL R3, R5 ;a1*b0
ADD R9, R0
ADC R10, R1
ADC R11, R14
ADC R15, R23
MUL R3, R6 ;a1*b1
ADD R10, R0
ADC R11, R1
ADC R12, R15
MUL R4, R7 ;a2*b2
MOVW R14, R0
MUL R4, R5 ;a2*b0
ADD R10, R0
ADC R11, R1
ADC R12, R14
ADC R15, R23
MUL R4, R6 ;a2*b1
ADD R11, R0
ADC R12, R1
ADC R13, R15
STD Z+0, R8
STD Z+1, R9
STD Z+2, R10

;--- load a3..a5 and b3..b5 ---
LD R14, X+
LD R15, X+
LD R16, X+
LDD R17, Y+3
LDD R18, Y+4
LDD R19, Y+5

;--- subtract a0-a3 etc. ---
SUB R2, R14
SBC R3, R15
SBC R4, R16
; 0xff if carry, 0x00 if no carry
SBC R26, R26

;--- subtract b0-b3 etc. ---
SUB R5, R17
SBC R6, R18
SBC R7, R19
; 0xff if carry, 0x00 if no carry
SBC R27, R27

;--- absolute values ---
EOR R2, R26
EOR R3, R26
EOR R4, R26
EOR R5, R27
EOR R6, R27
EOR R7, R27
NEG R26
NEG R27
ADD R2, R26
ADC R3, R23
ADC R4, R23
ADD R5, R27
ADC R6, R23
ADC R7, R23

;--- Compute H + (l3,l4,l5) ---
MUL R14, R19 ;a0*b2
MOVW R24, R0
MUL R14, R17 ;a0*b0
ADD R11, R0
ADC R12, R1
ADC R13, R24
ADC R25, R23
MUL R14, R18 ;a0*b1
ADD R12, R0
ADC R13, R1
ADC R20, R25
MUL R15, R19 ;a1*b2
MOVW R24, R0
MUL R15, R17 ;a1*b0
ADD R12, R0
ADC R13, R1
ADC R20, R24
ADC R25, R23
MUL R15, R18 ;a1*b1
ADD R13, R0
ADC R20, R1
ADC R21, R25
MUL R16, R19 ;a2*b2
MOVW R24, R0
MUL R16, R17 ;a2*b0
ADD R13, R0
ADC R20, R1
ADC R21, R24
ADC R25, R23
MUL R16, R18 ;a2*b1
MOVW R18,R22
ADD R20, R0
ADC R21, R1
ADC R22, R25

;--- Compute M ---
MUL R2, R7 ;a0*b2
MOVW R16, R0
MUL R2, R5 ;a0*b0
MOVW R14, R0
MUL R2, R6 ;a0*b1
ADD R15, R0
ADC R16, R1
ADC R17, R23
MUL R3, R7 ;a1*b2
MOVW R24, R0
MUL R3, R5 ;a1*b0
ADD R15, R0
ADC R16, R1
ADC R17, R24
ADC R25, R23
MUL R3, R6 ;a1*b1
ADD R16, R0
ADC R17, R1
ADC R18, R25
MUL R4, R7 ;a2*b2
MOVW R24, R0
MUL R4, R5 ;a2*b0
ADD R16, R0
ADC R17, R1
ADC R18, R24
ADC R25, R23
MUL R4, R6 ;a2*b1
ADD R17, R0
ADC R18, R1
ADC R19, R25

;--- add l3+h0 to h3 etc. ---
ADD R8, R11
ADC R9, R12
ADC R10, R13
ADC R11, R20
ADC R12, R21
ADC R13, R22
;store carry in R23
ADC R23, R23

;--- load sign bit ---
EOR R26, R27
BRNE add_M

;subtract M
SUB R8, R14
SBC R9, R15
SBC R10, R16
SBC R11, R17
SBC R12, R18
SBC R13, R19
SBCI R23, 0
SBC R24, R24
;R23:R24 is -1, 0, or 1
RJMP final

add_M:
  ADD R8, R14
  ADC R9, R15
  ADC R10, R16
  ADC R11, R17
  ADC R12, R18
  ADC R13, R19
  CLR R24
  ADC R23, R24
  NOP ; constant time

final:
  STD Z+3, R8
  STD Z+4, R9
  STD Z+5, R10
  STD Z+6, R11
  STD Z+7, R12
  STD Z+8, R13

  ;--- ripple carry ---
  ADD R20, R23
  ADC R21, R24
  ADC R22, R24

  STD Z+9, R20
  STD Z+10, R21
  STD Z+11, R22
```

B Small multiprecision multiplications

Listing 2 Optimized multiplication of two 24-bit integers, input A in registers R2,R3,R4; input B in registers R7,R8,R9; result in registers R12,R13,R14,R15,R16,R17.

```
CLR R23             MUL R3,R9           LD R2, X+
                    MOVW R24,R0         MUL R4,R9
MUL R2,R9                               MOVW R24,R0
MOVW R14, R0        MUL R3,R7
                    ADD R13,R0          MUL R4,R7
MUL R2,R7           ADC R14,R1          ADD R14,R0
MOVW R12, R0        ADC R15,R24         ADC R15,R1
                    ADC R25,R23         ADC R16,R24
MUL R2,R8                               ADC R25,R23
ADD R13,R0          MUL R3,R8
ADC R14,R1          ADD R14,R0          MUL R4,R8
ADC R15,R23         ADC R15,R1          ADD R15,R0
                    ADC R16,R25         ADC R16,R1
                                        ADC R17,R25
CLR R16
LD R2, X+           CLR R17
```

Listing 3 Optimized multiplication of two 32-bit integers, input A in registers R2,R3,R4,R5; input B in registers R7,R8,R9,R10; result in registers R12,R13,R14,R15,R16,R17,R18,R19.

```
CLR R23             MUL R3,R8           ADD R17,R25
                    ADD R14,R0          ADC R18,R0
MUL R2,R9           ADC R15,R1          ADC R19,R1
MOVW R14, R0        ADC R25, R23

MUL R2,R7           MUL R4,R10          MUL R5,R8
MOVW R12, R0        ADD R16,R25         MOVW R24,R0
                    ADC R17,R0
MUL R2,R8           ADC R18,R1          MUL R4,R8
ADD R13,R0                              ADD R15, R0
ADC R14,R1                              ADC R24, R1
ADC R15,R23         CLR R19             ADC R25, R23
                    MUL R4,R9
MUL R3,R10          MOVW R24,R0         MUL R5,R7
MOVW R16,R0                             ADD R15, R0
                    MUL R4,R7           ADC R24, R1
                    ADD R14, R0         ADC R25, R23
CLR R18             ADC R15, R1
MUL R2,R10          ADC R16, R24        MUL R5,R9
MOVW R24,R0         ADC R25, R23        ADD R16, R24
                                        ADC R0, R25
MUL R3,R7           MUL R3,R9           ADC R1, R23
ADD R13, R0         ADD R15,R0          ADD R17, R0
ADC R14, R1         ADC R16,R1          ADC R18, R1
ADC R15, R24        ADC R25, R23        ADC R19, R23
ADC R25, R23
                    MUL R5,R10
```

Listing 4 Optimized multiplication of two 40-bit integers, input A in registers R2,R3,R4,R5,R6; input B in registers R7,R8,R9,R10,R11; result in registers R12,R13,R14,R15,R16,R17,R18,R19,R20,R21.

```
CLR R18             ADC R18, R1
CLR R19                                 MUL R5, R8
MOVW R20, R18       MUL R3, R10         ADD R16, R0
                    ADD R16, R0         ADC R17, R1
MUL R2, R9          ADC R17, R1         ADC R25, R21
MOVW R14, R0        ADC R18, R21
                                        MUL R5, R11
MUL R2, R7                              ADD R18, R25
MOVW R12, R0        MUL R4, R9          ADC R19, R0
                    MOVW R24, R0        ADC R20, R1
MUL R2, R8
ADD R13, R0         MUL R4, R7          MUL R5, R10
ADC R14, R1         ADD R14, R0         ADD R18, R0
ADC R15, R21        ADC R15, R1         ADC R19, R1
                    ADC R16, R24        ADC R20, R21
MUL R2, R11         ADC R25, R21
MOVW R16, R0                            MUL R6, R9
                    MUL R4, R8          MOVW R24, R0
MUL R2, R10         ADD R15, R0
ADD R15, R0         ADC R16, R1         MUL R6, R7
ADC R16, R1         ADC R25, R21        ADD R16, R0
ADC R17, R21                            ADC R17, R1
                    MUL R4, R11         ADC R18, R24
                    ADD R17, R25        ADC R25, R21
MUL R3, R9          ADC R18, R0
MOVW R24, R0        ADC R19, R1         MUL R6, R8
                                        ADD R17, R0
MUL R3, R7          MUL R4, R10         ADC R18, R1
ADD R13, R0         ADD R17, R0         ADC R25, R21
ADC R14, R1         ADC R18, R1
ADC R15, R24        ADC R19, R21        MUL R6, R10
ADC R25, R21                            ADD R19, R0
                                        ADC R20, R1
MUL R3, R8          MUL R5, R9          ADC R21, R21
ADD R14, R0         MOVW R24, R0
ADC R15, R1                             MUL R6, R11
ADC R25, R21        MUL R5, R7          ADD R19, R25
                    ADD R15, R0         ADC R20, R0
MUL R3, R11         ADC R16, R1         ADC R21, R1
ADD R16, R25        ADC R17, R24
ADC R17, R0         ADC R25, R21
```

Listing 5 Optimized multiplication of two 48-bit integers, input A in registers R2,R3,R4,R5,R6,R7; input B in registers R8,R9,R10,R11,R12,R13; result in registers R14,R15,R16,R17,R18,R19,R20,R21,R22,R23,R24,R25.

```
CLR R20             MUL R4,R10          MUL R6,R10
CLR R21             MOVW R26, R0        MOVW R26, R0
MOVW R22, R20
MOVW R24, R20       MUL R4,R8           MUL R6,R8
                    ADD R16,R0          ADD R18,R0
MUL R2,R10          ADC R17,R1          ADC R19,R1
MOVW R16, R0        ADC R18,R26         ADC R20,R26
                    ADC R27,R25         ADC R27,R25
MUL R2,R8
MOVW R14, R0        MUL R4,R9           MUL R6,R9
                    ADD R17,R0          ADD R19,R0
MUL R2,R9           ADC R18,R1          ADC R20,R1
ADD R15,R0          ADC R27,R25         ADC R27,R25
ADC R16,R1
ADC R17,R25         MUL R4,R12          MUL R6,R12
                    ADD R19,R27         ADD R21,R27
MUL R2,R12          ADC R20,R0          ADC R22,R0
MOVW R18,R0         ADC R21,R1          ADC R23,R1
                    ADC R22,R25         ADC R24,R25
MUL R2,R11
ADD R17,R0          MUL R4,R11          MUL R6,R11
ADC R18,R1          MOVW R26,R0         MOVW R26,R0
ADC R19,R25
                    MUL R4,R13          MUL R6,R13
MUL R2,R13          ADD R19,R26         ADD R21,R26
ADD R19, R0         ADC R20,R27         ADC R22,R27
ADC R20, R1         ADC R21,R0          ADC R23,R0
                    ADC R22,R1          ADC R24,R1

MUL R3,R10
MOVW R26, R0        MUL R5,R10          MUL R7,R10
                    MOVW R26, R0        MOVW R26, R0
MUL R3,R8
ADD R15,R0          MUL R5,R8           MUL R7,R8
ADC R16,R1          ADD R17,R0          ADD R19,R0
ADC R17,R26         ADC R18,R1          ADC R20,R1
ADC R27,R25         ADC R19,R26         ADC R21,R26
                    ADC R27,R25         ADC R27,R25
MUL R3,R9
ADD R16,R0          MUL R5,R9           MUL R7,R9
ADC R17,R1          ADD R18,R0          ADD R20,R0
ADC R27,R25         ADC R19,R1          ADC R21,R1
                    ADC R27,R25         ADC R27,R25
MUL R3,R12
ADD R18,R27         MUL R5,R12          MUL R7,R12
ADC R19,R0          ADD R20,R27         ADD R22,R27
ADC R20,R1          ADC R21,R0          ADC R23,R0
ADC R21,R25         ADC R22,R1          ADC R24,R1
                    ADC R23,R25         ADC R25,R25
MUL R3,R11
MOVW R26,R0         MUL R5,R11          MUL R7,R11
                    MOVW R26,R0         MOVW R26,R0
MUL R3,R13
ADD R18,R26         MUL R5,R13          MUL R7,R13
ADC R19,R27         ADD R20,R26         ADD R22,R26
ADC R20,R0          ADC R21,R27         ADC R23,R27
ADC R21,R1          ADC R22,R0          ADC R24,R0
                    ADC R23,R1          ADC R25,R1
```