Verification of Optimised 48-bit Multiplications on AVR

Julien Schmaltz
Eindhoven University of Technology
Eindhoven, The Netherlands
j.schmaltz@tue.nl

Peter Schwabe
Radboud University
Nijmegen, The Netherlands
peter@cryptojedi.org

We present the verification of hand-optimised assembly programs computing 48-bit multiplications for the AVR ATmega 8-bit microcontroller. The specification of these algorithms is given by a straight-forward AVR assembly program implementing product-scanning multiplication. We prove that the computation produced by a carefully optimised quadratic-complexity and by an optimised Karatsuba multiplication are equal to the result produced by the specification. We also prove equivalence between two versions of the Karatsuba multiplication. The proofs are based on symbolic simulation in ACL2 using a formal model of the relevant parts of the Instruction Set Architecture of the AVR ATmega. The proofs then use automatic solvers either via the GL interface or applied outside ACL2 using the CNF files produced by GL. The conclusion is that the verification of some of these 48-bit multiplications is systematic.

1 Introduction

Cryptography is one of the relatively few domains of software engineering where speed is so important that experts spend serious effort to optimise software by hand on the assembly level. This is particularly true for cryptographic software running on slow and resource-constrained embedded devices.

The problem with such hand-optimised code is that its correctness is very hard to ascertain. Testing can provide some level of confidence in software, but for security-critical software we would like to have the strong guarantees, that can only be obtained through formal verification. This is particularly the case for highly optimised arithmetic on large integers that has been found to be succeptible to bugs that are hard or close to impossible to catch by testing. See, for example, [4], the comment on https://cryptojedi.org/crypto/#ed25519, and the bug in the squaring routine of OpenSSL described in CVE-2014-3570.

In this paper, we consider the verification of hand-optimised 48-bit multiprecision multiplications proposed by Hutter and Schwabe [9]. The software is targeting the AVR ATmega architecture and consists in two programs: an optimized quadratic-complexity multiplication algorithm and an optimised subquadratic-complexity Karatsuba multiplication. We focus on multiplication of 48-bit integers because it is the smallest size for which the Karatsuba multiplication pays off. Note that 48-bit integers are much smaller than what is typically used in, for example, elliptic-curve cryptography. Verification of correctness of the signficantly more complex multiplication routines that handle 160-bit to 256-bit multiplication from [9] seem to be out of reach with the approach described here.

We present results in using ACL2 in combination with SAT solvers to verify these optimised assembly programs. Following Moore's approach for modelling and symbolically simulating machine code [13, 14], we developed a simple and basic formal model of the AVR ATmega ISA. We used ACL2 to symbolically compute each resulting byte of each multiplication. The specification of all algorithms is a naive implementation of the standard product scanning multiplication. We use GL [20, 21], functionally reduced AIGs, and a SAT solver to prove equivalence between each algorithm and the specification.

Submitted to:
ACL2 2015

© J. Schmaltz & P. Schwabe
This work is licensed under the
Creative Commons Attribution License.

Our experimental results are that the verification of the tweaked product scanning and the proof of equivalence between two versions of the Karatsuba multiplication are feasible. These proofs are systematic but take several hours of computation time. Note that symbolically simulating the programs seems scalable and could be fully automated using techniques used in the x86 machine code [8][1]. The bottleneck is solving the SAT instances. In almost all cases, functionally reducing the problem yields the empty clauses. For the equivalence between a Karatsuba multiplication and the naive specification, we are still waiting for the SAT solver to return UNSAT for the proof of equivalence between two pairs of bytes.

2 Multiprecision multiplication on AVR8

2.1 The AVR8 Architecture

We consider the architecture of the AVR ATmega 8-bit microcontroller. The AVR has 32 registers labelled R0 to R31. Some pairs are aliased and used to access memory. Register pair R26:R27 is aliased as `X`, pair R28:R29 as `Y`, and pair `R30:R31` as `Z`. These registers are the only ones that can hold addresses in load and store instructions.

The most relevant arithmetic instruction is MUL that takes 2 register labels as input. It multiplies the unsigned 8-bit integers in these two registers and writes the 16-bit result in registers R0 and R1, where R0 contains the low bits and R1 contains the high bits. Note that the carry flag is overwritten by the MUL instruction. We use addition without (ADD) or with (ADC) carry, subtraction without (SUB) or with (SBC) carry, and exclusive or (EOR). Other relevant instructions are CLR that clears a register to 0, NEG that negates a register, COM that computes the one's complement, MOV that moves a value from one register to another, MOVW that moves the values in a pair of registers to another pair of registers. The stack is manipulated using instructions PUSH and POP.

2.2 Representing multiprecision integers

The AVR8 architecture can natively work on integers of size up to 8-bits. This is insufficient not only for cryptographic applications but also for C programs that use 16-bit integers (type `int`), 32-bit integers (type `long`), or 64-bit integers (type `long long`). An integers of a size that exceeds the natively supported integers size of the target architecture is called a *multiprecision integer*.

On the AVR, such multiprecision integers are typically represented as an array of unsigned bytes. More specifically, an $8m$-bit integer A is represented as m bytes $(a_0,\ldots,a_{m-1})$ such that

$$A = \sum_{i=0}^{m-1} a_i \cdot 2^{8i},$$

and $a_i \in \{0,\ldots,255\}$ for $i = 0,\ldots,m-1$.

2.3 General multiprecision multiplication

In the following we will focus on the case $m = 6$, that is, we will consider the multiplication of two 48-bit integers $A \mathrel{\hat{=}} (a_0,\ldots,a_5)$ and $B \mathrel{\hat{=}} (b_0,\ldots,b_5)$. The basic idea of the multiplication of A and B is to

[1]Note that the techniques proposed in submissions 9 and 13 to this workshop would also be very useful to automate the symbolic simulations.

compute coefficients $r'_0, \ldots, r'_{10}$ such that $R = A \cdot B = \sum_{i=0}^{10} r'_i \cdot 2^{8i}$ as follows:

$$\begin{aligned}
r'_0 &= a_0 b_0 \\
r'_1 &= a_0 b_1 + a_1 b_0 \\
r'_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
r'_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \\
&\vdots \\
r'_9 &= a_4 b_5 + a_5 b_4 \\
r'_{10} &= a_5 b_5
\end{aligned}$$

The coefficients r'_i and the corresponding representation of R look very similar to the representation of A and B; however, the r_i do not fit into 8 bits. Each partial product $a_i b_j$ produces a result of up to 16 bits; accumulating those partial products generates carry bits. What we really want to obtain as a result is a byte array $(r_0, \ldots, r_{11})$ such that $R = \sum_{i=0}^{11} r_i \cdot 2^{8i}$ and $r_i \in \{0, \ldots, 255\}$. It is still helpful to keep the above computation of $r'_0, \ldots, r'_{10}$ in mind: most multiprecision-multiplication algorithms perform exactly this computation and handle the "overlaps" of the r'_i on the fly to produce the result as a byte array.

2.4 Product-scanning multiplication

The idea of product scanning is to scan through the bytes of the product from least significant to most significant. Let us assume that the input bytes $a_0, \ldots, a_5$ and $b_0, \ldots, b_5$ are already loaded into registers. Product scanning first computes $a_0 b_0$, stores the low byte of the result into r_0 and keeps the high byte of the result in an accumulator register t_1. We then clear registers t_2 and t_3, compute $a_0 b_1 + a_1 b_0$ and accumulate into registers t_1, t_2, t_3. Then we store the least significant accumulator register t_1 to r_1 and clear a register t_4. The next step computes $a_0 b_2 + a_1 b_1 + a_2 b_0$, accumulates into t_2, t_3, t_4, stores t_2 to t_2, clears a new most significant accumulator register t_5 and so on. The full AVR assembly code for 48-bit fully unrolled product-scanning multiplication is listed in Appendix A in Listing 1.

2.5 Optimised 48-bit multiplication

The fully unrolled product-scanning multiplication described in the previous subsection was long time believed to be the fastest approach to multiply "small" multiprecision integers on AVR8. As integers become larger the algorithm at some point needs more registers than are available on th AVR8 and register allocation becomes critical.

Liu and Großschädl in [11] use an inner-loop optimization that demonstrates that for 32-bit multiplication product-scanning is in fact *not* optimal. The optimisation is quite specific to the AVR8 and makes use of the fact that the `movw` instruction is capable of changing two bytes in one cycle. This can be used to reorder computation and accumulation of partial products $a_i b_j$, simplify carry handling, and save several cycles. Hutter and Schwabe[9] generalize the idea to other input sizes including 48-bit multiplication. The resulting optimised multiplication takes 172 cycles (excluding the cost of loading inputs and storing outputs) instead of 187 cycles for fully unrolled product scanning. The full AVR assembly code for the 48-bit optimised multiplication is listed in Appendix A, Listing 2.

2.6 Karatsuba multiplication

Reordering the computation of partial products does not change the fact that $n \times n$-byte multiprecision multiplication following the general approach described in Subsection 2.3 needs n^2 multiplication instructions. There exist various multiplication algorithms with lower asymptotic complexities. The first such algorithm was proposed by Karatsuba in 1963 [10]. The idea is to split a multiplication of two integers into 3 multiplications of half the size and some additions as follows:

1. Write $A = A_\ell + 2^k A_h$ and $B = B_\ell 2^k B_h$, where A_ℓ, A_h, B_ℓ, and B_h have half the size of A and B;
2. compute $R_\ell = A_\ell B_\ell$;
3. compute $R_h = A_h B_h$;
4. compute $R_m = |A_\ell - A_h| \cdot |B_\ell - B_h|$;
5. set $t = 0$, if $R_m = (A_\ell - A_h) \cdot (B_\ell - B_h)$, $t = 1$ otherwise;
6. compute $R = R_\ell + 2^k(R_\ell + R_h - (-1)^t R_m) + 2^{2k} R_h$.

Recursive application of this idea yields an asymptotic running time of $\Theta(\log_2 3)$. Hutter and Schwabe [9] optimized this algorithm in AVR assembly and showed that the asymptotic advantage starts to pay off already for 48-bit input; much earlier than previously believed.

The cycles required by the Karatsuba multiplication software described in [9] are independent of the inputs; a property which is important in the context of cryptographic software because otherwise the timing might "leak" information about secret data. Hutter and Schwabe actually propose two variants of their 48-bit Karatsuba software. Both versions apply the Karatsuba recursion only for one level and compute the three 24-bit multiplications by an optimised quadratic-complexity approach. However their faster software uses a branch instruction to decide whether R_m is subtracted or added; the (slightly slower) branchfree method uses an arithmetic approach to conditionally negate R_m depending on the value of t. The same arithmetic approach is used in the computation of the absolute values in Step 4: The idea is to first expand the condition bit to an all-zero or all-one bitmask; then xor all bytes by this mask and then subtract this bit mask with borrow from all bytes. The idea behind this approach is the following: If the bitmask is all-zero then neither the xor nor the subtraction changes anything; the value is thus not negated. If the bitmask is all-one, then the xor produces the ones' complement; the subsequent subtraction of the mask is a subtraction of -1, which is the same as addition of 1. This step thus computes the two's complement. An earlier version of the Karatsuba software actually used addition of 1 instead of subtraction of -1 at this step. This means that, compared to the code listed in Listing 3, the lines

```
SUB R2, R26
SBC R3, R26
SBC R4, R26
SUB R5, R27
SBC R6, R27
SBC R7, R27
```

were

```
NEG R26
NEG R27
ADD R2, R26
ADC R3, R23
ADC R4, R23
ADD R5, R27
ADC R6, R23
ADC R7, R23
```

Also the lines

```
EOR R26, R27
COM R26
```

are different in that earlier version and read as follows:

```
EOR R26, R27
DEC R26
```

Finally, those two lines were later in the code, namely right before

```
ADC R23, R26
```

The reason that the two lines had to move to an earlier position in the code is subtle: the `DEC` instruction does not modify the carry flag, which is picked up by the `ADC R23, R26` instruction; the `COM` instruction always sets the carry flag to 1.

For more details on the 48-bit Karatsuba multiplication in AVR assembly see [9, Section 3]. The verification effort in this paper only considers the branchfree version of 48-bit Karatsuba multiplication. The full AVR assembly code for this multiplication is listed in Appendix A, Listing 3.

3 Formal model of the AVR8 ISA

The state of our machine includes a program counter, registers, flags, a stack, the program, and a memory.

```
(defun make-state (pc regs flags stack program memory)
  (list pc regs flags stack program memory))
```

The PC is modelled as an integer. All other components are modelled by lists. The memory is modelled by an ACL2 'alist' where keys are addresses pointing to memory content. There are eight flags. The only relevant one for our applications is the carry flag. Our model only includes the carry flag when necessary. This abstraction has a strong impact on the performance of symbolic simulations.

All state elements are unbounded lists. To proper represent the semantics of the AVR8 instructions, we need to explicitly make some of these elements finite. In particular, we need to restrict registers to 8 bits. To do so, we use the following macro, taken from the Y86 project by Hunt et al.[2]:

```
(defmacro n08 (x) `(logand ,x ,255))
```

We often need to select a particular byte of a large number. The following macro returns byte number i (starting from 0) of a bit sequence represented by integer x:

```
(defun byte_i (i x)
  (n08 (logtail (* i 8) x)))
```

Instructions have one or two parameters, which are accessed using functions `avr-arg1` and `avr-arg2`. Instruction MUL takes two registers and write in R0 and R1 the result of multiplying their content. A carry is produced if the 15th bit of the result is 1. The semantics of the MUL instruction is modelled as follows:

[2]See books/models/y86 of the ACL2 distribution.

```
(defun execute-MUL (inst s)
  (let* ((res (* (nth (avr-arg1 inst) (regs s))
                 (nth (avr-arg2 inst) (regs s))))
         (low (byte_i 0 res))
         (high (byte_i 1 res)))
    (make-state (+ 1 (apc s))
                (update-nth *R1* high (update-nth *R0* low (regs s)))
                (update-nth *flag-C* (logtail 7 high) (flags s))
                (stack s)
                (prg s)
                (memory s))))
```

First, the content of the registers are multiplied using the ACL2 '*' function. The result is restricted to two bytes which are then written in R0 and R1. The carry flag is set if the 8th bit of the high byte of the result is 1. Otherwise, the carry flag is cleared.

We proceed similarly to model the following instructions: ADD, ADC, SUB, SBC, DEC, EOR, NEG, ASR, MOV, MOVW, PUSH, POP, COM, and CLR. We also modelled some of the instructions to load directly or indirectly from memory.

Instructions are decoded as usual:

```
(defun do-inst (inst s)
  (cond ((equal (opcode inst) 'ADC)
         (execute-ADC   inst s))
        ((equal (opcode inst) 'ADD)
         (execute-ADD   inst s))
        ((equal (opcode inst) 'ASR)
         (execute-ASR inst s))
          ...
        ((equal (opcode inst) 'EOR)
         (execute-INC   inst s))
          ...
        ((equal (opcode inst) 'MUL)
         (execute-MUL   inst s))
        ((equal (opcode inst) 'NEG)
         (execute-NEG   inst s))
          ...
        (t s)))
```

We finally define the step and run functions:

```
(defun avr-step (s)
     (do-inst (next-inst s) s))

(defun run (sched s)
     (if (endp sched)
         s
       (run (cdr sched) (avr-step s))))
```

We prove usual 'opener' lemmas about these functions and disable them:

```
(defthm step-opener
  (implies (consp (next-inst s))
           (equal (avr-step s)
                  (do-inst (next-inst s) s))))


(defthm run-opener
  (and (equal (run nil s) s)
       (equal (run (cons th sched) s)
              (run sched (avr-step s)))))
```

4 Verification of the algorithms

4.1 Proof methodology

Symbolic simulation For each algorithm, we compute symbolic expressions for all bytes of the result. This means that we compute twelve terms representing byte 0 to byte 11 of each resulting product. This step is mainly accomplished following the traditional method for symbolic simulation in ACL2 [13]. The only issue is that terms are getting large and it becomes intractable to simulate an entire program in one step. We then decompose the simulation into intermediate steps. For 48-bit multiplications, this is not enough. Terms produced at intermediate steps need to be "hidden" using disabled functions. Simulation thus proceeds as follows. We first submit the following theorem to obtain the state produced after a given number of steps:

```
(thm
 (implies (AND (UNSIGNED-BYTE-P 16 A)
       (UNSIGNED-BYTE-P 16 B)
       (UNSIGNED-BYTE-P 16 R)
       (< (+ A 5) B)
       (< B (- (EXPT2 16) 5))
       (< R (- (EXPT2 16) 11))
       (UNSIGNED-BYTE-P 48 N)
       (UNSIGNED-BYTE-P 48 M))
  (equal (run (repeat 30 'TICK) (k_s0 a b r m n))
         ???))
          ... ) ; hints etc
```

This event produces a call to `make-state` from which we extract expressions representing registers. We typically define functions like the following:

```
(defund k_R9-0 (m n)
  (LOGAND (+ (BYTE_I 0 (* (BYTE_I 0 M) (BYTE_I 1 N)))
             (LOGAND (+ (BYTE_I 0 (* (BYTE_I 0 N) (BYTE_I 1 M)))
                        (BYTE_I 1 (* (BYTE_I 0 M) (BYTE_I 0 N))))
                     255))
          255))
```

We then re-write the call to `make-state` using these functions instead of the large expressions. The following function defines the state reached in the simulation of the Karatsuba multiplication after 64 steps:

```
(defun k_s3 (a b r m n)
  (MAKE-STATE
   64
   (LIST
    ... ;; some small terms
    (k_R8-0 M N)
    (k_R9-0 M N)
    (k_R10-0 M N)
    (k_R11-0 M N)
    (k_R12-0 M N)
    (k_R13-0 M N)
    (BYTE_I 0 (* (BYTE_I 2 M) (BYTE_I 2 N)))
    ... ;; some small terms
    )
   (LIST
    0 0 0 0 0 0 1 ... ) ; large carry expression not shown
   '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
   *bigint_karatsuba_branchfree_mul48*
   (LIST* ;; updated expression of the memory
    (LIST (+ 2 R) (k_Z2 M N))
    (LIST (+ 1 R) (k_Z1 M N))
    (LIST R (k_Z0 M N))
    (INIT-MEMORY-*BIGINT_PRODSCAN_MUL48* A B R N M)))
```

We then continue the symbolic simulation from this state expression keeping functions `k_RX-i` and `k_ZX` disabled.

Bit blasting and SAT solving We submit theorems stating the equality between each byte using the GL interface (`def-gl-thm`) in SAT mode. The SAT solver used is `glucose` [18] that is itself built on top of Minisat [6]. If the proof does not succeed within 15 minutes, we save the CNF file produced by GL, convert the CNF to the AIGER format. We then use ABC [22] and the 'fraig' command to reduce the AIG and the command 'write_cnf' to write the result back as a CNF file. In many cases, functionally reducing the AIG produces two clauses, one of them being the empty clause. For the remaining cases, we convert the reduced AIG to CNF and submit the result to the SAT solver. A typical call to GL looks like the following:

```
(def-gl-thm k_z9-=-c_Z9
  :hyp
  (and (unsigned-byte-p 48 n)
       (unsigned-byte-p 48 m))
  :concl
  (equal (k_Z9 m n) ;; byte 9 of Karatsuba
         (c_Z9 m n)) ;; byte 9 of product scanning
  ::g-bindings `((n ,(gl::g-int 0 2 49))
                 (m ,(gl::g-int 1 2 49))))
```

4.2 Experimental results

Experiments[3] have been conducted on an Intel i7 running at 1.7GHz under Mac OS X 10.10.3. The machine has 8 GB of main memory. We use ACL2 version 7.1 and CCL version `1.10-dev-r16130M-trunk`. We use Glucose 4.0 as our SAT solver. To functionally reduce AIG, we use the 'fraig' command from ABC version 1.1. The conversion from CNF to AIGER is done using the tool cnf2aig[4] from Biere.

Table 1 shows the verification times for the verification of the optimised product scanning multiplication. For proving correctness of bytes 0 to 4, GL succeeds within 15 minutes. For bytes 5 and 6, the time needed for bit blasting is about a minute. This time then grows to more than 10 hours for byte 11. For all these bytes, functionally reducing the corresponding AIG solves each problem within 50 minutes. The only difficult proof is byte 5. The SAT solver needs about 9 hours to prove the problem UNSAT. Note that even if the reduced CNF file actually has more variables and clauses than the initial one, it takes less time to solve it. Solving the initial problem was killed when the reduced file was proven UNSAT.

byte #	create CNF	# variables	# clauses	# vars fraig	# clauses fraig	time fraig	time SAT
0	0	2	0	–	–	–	0
1	0	2	0	–	–	–	0
2	0	1893	6207	–	–	–	0
3	0	3639	12052	–	–	–	30
4	0	5965	19862	–	–	–	850
5	60	8856	29613	11109	44070	1800	33540
6	60	11898	39900	4575	2	3000	0
7	120	14464	48574	5549	2	3000	0
8	780	16460	55318	6303	2	3000	0
9	5400	17880	60111	6834	2	3000	0
10	18000	18737	62993	7150	2	3000	0
11	39600	19023	63949	7246	2	3000	0

Table 1: Verification times for the optimised product scanning multiplication.

4.3 Verification of the Karatsuba multiplication

Table 2 shows the computation times to prove the Karatsuba multiplication. We consider the earlier version of the Karatsuba multiplication using addition of 1 and the `DEC` instruction. Byte 0 to 2 are proven instantaneously by GL. Bytes 5 to 11 are proven within 50 minutes by the 'fraig' command of ABC. For this case, bit blasting costs also up to 10 hours for byte 11. The real bottleneck is the time needed to the SAT solver for bytes 3 and 4. Solving these instances was killed after 3 days.

We found a workaround using contrapositive reasoning and the fact that byte 5 is computed correctly. Consider the verification of byte 3. We first submit the following to ACL2 to generate a CNF file:

```
(def-gl-thm k_z3-=-c_Z3-contra
  :hyp
  (and (unsigned-byte-p 48 n)
```

[3] All the ACL2 files and generated CNF files are available on-line at http://win.tue.nl/∼jschmalt/publications/acl215/acl215.html
[4] http://fmv.jku.at/cnf2aig/

byte #	create CNF	# variables	# clauses	# vars fraig	# clauses fraig	time fraig	time SAT
0	0	2	0	–	–	–	0
1	0	2	0	–	–	–	0
2	0	1893	6207	–	–	–	0
3	0	4483	14858	5618	22182	1800	$>$ 3 days
4	0	7900	26429	9967	39562	1800	$>$ 3 days
5	5	12599	42308	4887	2	3000	0
6	10	17370	58406	6669	2	3000	0
7	30	20531	69079	7856	2	3000	0
8	420	21962	73919	8400	2	3000	0
9	2400	22290	75034	8529	2	3000	0
10	18000	22475	75665	8596	2	3000	0
11	36000	22565	75967	8626	2	3000	0

Table 2: Verification times for the Karatsuba multiplication.

```
      (unsigned-byte-p 48 m)
      )
  :concl
  (implies (not (equal (k_Z3 m n)
                       (c_Z3 m n)))
           (not (equal (k_Z5 m n)
                       (c_Z5 m n))))
  ::g-bindings `((n ,(gl::g-int 0 2 49))
                 (m ,(gl::g-int 1 2 49))))
```

The CNF file – generated within 1 second – is converted to AIGER and processed by the 'fraig' command of ABC. This produces the empty clause within 50 minutes. We prove the correctness of byte 3:

```
 (defthm k_z3-=-c_Z3
  (implies (and (unsigned-byte-p 48 n)
                (unsigned-byte-p 48 m))
           (equal (k_Z3 m n)
                  (c_Z3 m n)))
  :hints (("GOAL"
           :use k_z3-=-c_Z3-contra)))
```

We proceed similarly to prove byte 4 and obtained similar results. This completes the proof of the correctness of the Karatsuba multiplication.

4.4 Verification of another version of the Karatsuba multiplication

To further experiment, we consider the proof of equivalence between the two versions of the Karatsuba multiplication, the earlier one with addition and the more recent one using subtraction. Table 3 shows the results of proving equivalence between these two programs. Because the programs are close to each other, GL performs bit blasting very quickly. The high bytes (6 to 11) requires 50 minutes to be solved by the 'fraig' command of ABC. Proving the equivalence for byte 5 requires slightly more than 10 minutes

byte #	create CNF	# variables	# clauses	# vars fraig	# clauses fraig	time fraig	time SAT
0	0	2	0	–	–	–	0
1	0	2	0	–	–	–	0
2	0	2	0	–	–	–	0
3	0	2	0	–	–	–	0
4	0	6355	21203	–	–	–	5
5	30	10059	33751	12757	50685	1800	700
6	20	13657	45898	5298	2	3000	0
7	20	15995	53796	6180	2	3000	0
8	20	16953	57035	6543	2	3000	0
9	20	17075	57447	6587	2	3000	0
10	20	17173	57777	6587	2	3000	0
11	20	17271	58107	6655	2	3000	0

Table 3: Verification times for a second version of the Karatsuba multiplication.

of SAT solving. Even if the two programs are very similar, proving their equivalence takes about an hour (assuming all bytes are proven in parallel).

5 Related work

Recently, Lin et al. [23] used a combination of interactive theorem proving and the Boolector SMT solver [5] to verify a Montgomery ladder step used for elliptic curve computations. Programs are written in an architecture independent assembly language. They consider radix-2^{51} and radix-2^{64} representations and verify a 256-bit modular multiplication. Their approach is largely automated but requires a significant amount of annotations to the source code as well as manual interaction with the theorem proving system. In contrast, our approach aims at being fully automatic but it is unclear how it evolves when applied to modular multiplications and larger integers.

The verification of big number libraries is an active research field for decades. Recently, Myreen and Curelo [15] used HOL4 [19] and a proof producing compiler decompiler to verify 64-bit x86 arbitrary-precision arithmetic functions. They verified both multiplication and divisions. Their verification traverses more layers. They prove correctness of the assembly programs and correctness of the generated machine code. Affeldt and Marti [2, 1] developed SmartMIPS assembly programs for arithmetic operations excluding division and modulus but including a Montgomery multiplication. Fisher [7] used the Isabelle/HOL [17] theorem prover to verify a C-like implementation of arbitrary-precision arithmetic functions. Moore [12] seems to have been the first to prove a big number routine in assembly. He used a big number addition to illustrate the verification of Piton code.

6 Conclusion

We presented the application of ACL2 in combination with external SAT solvers to verify hand optimised assembly programs for 48-bit multiplication on AVR. The formal model of the AVR ISA was developed following the approach proposed by Moore for the M1 machine. The symbolic simulation requires some hiding to keep the size of symbolic terms manageable. Using a combination of functionally reduced

AIGs and SAT solving, we proved equivalence between two product scanning implementations and two Karatsuba implementations. We also proved equivalence between one of the Karatsuba multiplication and the standard product scanning one. To conclude, combining ACL2 together with automatic provers like Glucose or ABC, is capable of verifying hand-optimised 48-bit multiplications on AVR. Our current future work is to look at automating our systematic approach using e.g. Codewalker for symbolic simulation. Our model of the AVR ATmega is very basic and need to be further developed and validated.

References

[1] Reynald Affeldt (2013): *On construction of a library of formally verified low-level arithmetic functions*. *Innovations in Systems and Software Engineering* 9(2), pp. 59–77.

[2] Reynald Affeldt & Nicolas Marti (2007): *An approach to formal verification of arithmetic functions in assembly*. In: *Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues*, Springer, pp. 346–360.

[3] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe & Bo-Yin Yang (2012): *High-speed high-security signatures*. *Journal of Cryptographic Engineering* 2(2), pp. 77–89.

[4] Billy B. Brumley, Manuel Barbosa, Dan Page & Frederik Vercauteren (2012): *Practical Realisation and Elimination of an ECC-Related Software Bug Attack*. In Orr Dunkelman, editor: *Topics in Cryptology – CT-RSA 2012*, *Lecture Notes in Computer Science* 7178, Springer-Verlag Berlin Heidelberg, pp. 171–186. `http://eprint.iacr.org/2011/633`.

[5] Robert Brummayer & Armin Biere (2009): *Boolector: An efficient SMT solver for bit-vectors and arrays*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 174–177.

[6] Niklas Eén & Niklas Sörensson (2004): *An extensible SAT-solver*. In: *Theory and applications of satisfiability testing*, Springer, pp. 502–518.

[7] Sabine Fischer (2008): *Formal verification of a big integer library*. In: *DATE08 Workshop on Dependable Software Systems. Available from http://busserver. cs. uni-sb. de/publikationen/Fi08DATE. pdf.*

[8] Shilpi Goel, Warren A. Hunt, Matt Kaufmann & Soumava Ghosh (2014): *Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls*. In: *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, FMCAD Inc, Austin, TX, pp. 18:91–18:98. Available at `http://dl.acm.org/citation.cfm?id=2682923.2682944`.

[9] Michael Hutter & Peter Schwabe (2015): *Multiprecision multiplication on AVR revisited*. *Journal of Cryptographic Engineering*.

[10] Anatolii Karatsuba & Yuri Ofman (1963): *Multiplication of multidigit numbers on automata*. *Soviet Physics Doklady* 7, pp. 595–596. 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`.

[11] Zhe Liu & Johann Großschädl (2014): *New Speed Records for Montgomery Modular Multiplication on 8-bit AVR Microcontrollers*. `https://eprint.iacr.org/2013/882/`.

[12] J Strother Moore (1989): *A mechanically verified language implementation*. *Journal of Automated Reasoning* 5(4), pp. 461–492.

[13] J. Strother Moore (1998): *Symbolic Simulation: an ACL2 Approach*. In: *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), volume LNCS 1522*, Springer-Verlag, pp. 334–350.

[14] J Strother Moore (2008): *Mechanized Operational Semantics: The M1 Story*.

[15] Magnus O Myreen & Gregorio Curello (2013): *Proof pearl: A verified bignum implementation in x86-64 machine code*. In: *Certified Programs and Proofs*, Springer, pp. 66–81.

[16] Magnus O Myreen & Michael JC Gordon (2007): *Hoare logic for realistically modelled machine code*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 568–582.

[17] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer Science & Business Media.

[18] Laurent Simon & Gilles Audemard (2009): *Predicting Learnt Clauses Quality in Modern SAT Solver*. In: *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*.

[19] Konrad Slind & Michael Norrish (2008): *A brief overview of HOL4*. In: *Theorem Proving in Higher Order Logics*, Springer, pp. 28–32.

[20] Sol Swords (2010): *A verified framework for symbolic execution in the ACL2 theorem prover*. Ph.D. thesis, University of Texas at Austin.

[21] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011.*, pp. 84–102, doi:10.4204/EPTCS.70.7. Available at http://dx.doi.org/10.4204/EPTCS.70.7.

[22] Berkeley Logic Synthesis & Verification Group: *ABC: A System for Sequential Synthesis and Verification*. Available at http://www.eecs.berkeley.edu/~alanmi/abc/.

[23] Hsin-Hung Lin Peter Schwabe Ming-Hsien Tsai Bow-Yaw Wang Bo-Yin Yang Yu-Fang Chen, Chang-Hong Hsu & Shang-Yi Yang (2014): *Verifying Curve25519 Software*. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, *CCS'14*, ACM, pp. 299–309. http://cryptojedi.org/papers/#verify25519.

A AVR code of multiprecision multiplication

This section lists the AVR assembly code considered in this paper. Listing 2 is essentially the same as [9, Listing 5], except that we include loads of inputs and stores of results. Listing 3 is slightly different than [9, Listing 1], because we consider the branch-free variant of Karatsuba multiplication; for details see [9, Section 3].

Listing 1 Product-scanning multiplication of two 48-bit integers. Input *A* in X+0,X+1,...,X+5; input *B* in Y+0,Y+1,...,Y+5; result *R* in Z+0,Z+1,...,Z+11.

```
CLR R5

LD R6, X+
LD R7, X+
LD R8, X+
LD R9, X+
LD R10, X+
LD R11, X+

LDD R12, Y+0
LDD R13, Y+1
LDD R14, Y+2
LDD R15, Y+3
LDD R16, Y+4
LDD R17, Y+5


MUL R6, R12
MOV R4, R1
STD Z+0, R0
CLR R2
CLR R3

MUL R6, R13
ADD R4, R0
ADC R2, R1
MUL R7, R12
ADD R4, R0
ADC R2, R1
ADC R3, R5
STD Z+1, R4
CLR R4

MUL R6, R14
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R7, R13
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R8, R12
ADD R2, R0
ADC R3, R1
ADC R4, R5
STD Z+2, R2
CLR R2

MUL R6, R15
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R7, R14
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R8, R13
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R9, R12
ADD R3, R0
ADC R4, R1
ADC R2, R5
STD Z+3, R3
CLR R3

MUL R6, R16
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R7, R15
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R8, R14
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R9, R13
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R10, R12
ADD R4, R0
ADC R2, R1
ADC R3, R5
STD Z+4, R4
CLR R4

MUL R6, R17
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R7, R16
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R8, R15
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R9, R14
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R10, R13
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R11, R12
ADD R2, R0
ADC R3, R1
ADC R4, R5
STD Z+5, R2
CLR R2

MUL R7, R17
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R8, R16
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R9, R15
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R10, R14
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R11, R13
ADD R3, R0
ADC R4, R1
ADC R2, R5
STD Z+6, R3
CLR R3

MUL R8, R17
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R9, R16
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R10, R15
ADD R4, R0
ADC R2, R1
ADC R3, R5
MUL R11, R14
ADD R4, R0
ADC R2, R1
ADC R3, R5
STD Z+7, R4
CLR R4

MUL R9, R17
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R10, R16
ADD R2, R0
ADC R3, R1
ADC R4, R5
MUL R11, R15
ADD R2, R0
ADC R3, R1
ADC R4, R5
STD Z+8, R2
CLR R2

MUL R10, R17
ADD R3, R0
ADC R4, R1
ADC R2, R5
MUL R11, R16
ADD R3, R0
ADC R4, R1
ADC R2, R5
STD Z+9, R3

MUL R11, R17
ADD R4, R0
ADC R2, R1
STD Z+10, R4
STD Z+11, R2
```

Listing 2 Optimised multiplication of two 48-bit integers. Input A in `X+0`,`X+1`,...,`X+5`; input B in `Y+0`,`Y+1`,...,`Y+5`; result R in `Z+0`,`Z+1`,...,`Z+11`.

```
LD R2, X+
LD R3, X+
LD R4, X+
LD R5, X+
LD R6, X+
LD R7, X+

LD R8, Y+
LD R9, Y+
LD R10, Y+
LD R11, Y+
LD R12, Y+
LD R13, Y+

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


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


MUL R4,R10
MOVW R26, R0
MUL R4,R8
ADD R16,R0
ADC R17,R1
ADC R18,R26
ADC R27,R25
MUL R4,R9
ADD R17,R0
ADC R18,R1
ADC R27,R25
MUL R4,R12
ADD R19,R27
ADC R20,R0
ADC R21,R1
ADC R22,R25
MUL R4,R11
MOVW R26,R0
MUL R4,R13
ADD R19,R26
ADC R20,R27
ADC R21,R0
ADC R22,R1


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


MUL R6,R10
MOVW R26, R0
MUL R6,R8
ADD R18,R0
ADC R19,R1
ADC R20,R26
ADC R27,R25
MUL R6,R9
ADD R19,R0
ADC R20,R1
ADC R27,R25
MUL R6,R12
ADD R21,R27
ADC R22,R0
ADC R23,R1
ADC R24,R25
MUL R6,R11
MOVW R26,R0
MUL R6,R13
ADD R21,R26
ADC R22,R27
ADC R23,R0
ADC R24,R1


MUL R7,R10
MOVW R26, R0
MUL R7,R8
ADD R19,R0
ADC R20,R1
ADC R21,R26
ADC R27,R25
MUL R7,R9
ADD R20,R0
ADC R21,R1
ADC R27,R25
MUL R7,R12
ADD R22,R27
ADC R23,R0
ADC R24,R1
ADC R25,R25
MUL R7,R11
MOVW R26,R0
MUL R7,R13
ADD R22,R26
ADC R23,R27
ADC R24,R0
ADC R25,R1

ST Z+, R14
ST Z+, R15
ST Z+, R16
ST Z+, R17
ST Z+, R18
ST Z+, R19
ST Z+, R20
ST Z+, R21
ST Z+, R22
ST Z+, R23
ST Z+, R24
ST Z+, R25
```

Listing 3 Branchfree Karatsuba multiplication of two 48-bit integers. Input A in `X+0`,`X+1`,...,`X+5`; input B in `Y+0`,`Y+1`,...,`Y+5`; result R in `Z+0`,`Z+1`,...,`Z+11`.

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

LD R2, X+
LD R3, X+
LD R4, X+
LDD R5, Y+0
LDD R6, Y+1
LDD R7, Y+2

MUL R2, R7
MOVW R10, R0
MUL R2, R5
MOVW R8, R0
MUL R2, R6
ADD R9, R0
ADC R10, R1
ADC R11, R23

MUL R3, R7
MOVW R14, R0
MUL R3, R5
ADD R9, R0
ADC R10, R1
ADC R11, R14
ADC R15, R23
MUL R3, R6
ADD R10, R0
ADC R11, R1
ADC R12, R15

MUL R4, R7
MOVW R14, R0
MUL R4, R5
ADD R10, R0
ADC R11, R1
ADC R12, R14
ADC R15, R23
MUL R4, R6
ADD R11, R0
ADC R12, R1
ADC R13, R15
STD Z+0, R8
STD Z+1, R9
STD Z+2, R10


LD R14, X+
LD R15, X+
LD R16, X+
LDD R17, Y+3
LDD R18, Y+4
LDD R19, Y+5

SUB R2, R14
SBC R3, R15
SBC R4, R16
SBC R26, R26

SUB R5, R17
SBC R6, R18
SBC R7, R19
SBC R27, R27
```

```

EOR R2, R26
EOR R3, R26
EOR R4, R26
EOR R5, R27
EOR R6, R27
EOR R7, R27

SUB R2, R26
SBC R3, R26
SBC R4, R26
SUB R5, R27
SBC R6, R27
SBC R7, R27

MUL R14, R19
MOVW R24, R0
MUL R14, R17
ADD R11, R0
ADC R12, R1
ADC R13, R24
ADC R25, R23
MUL R14, R18
ADD R12, R0
ADC R13, R1
ADC R20, R25

MUL R15, R19
MOVW R24, R0
MUL R15, R17
ADD R12, R0
ADC R13, R1
ADC R20, R24
ADC R25, R23
MUL R15, R18
ADD R13, R0
ADC R20, R1
ADC R21, R25

MUL R16, R19
MOVW R24, R0
MUL R16, R17
ADD R13, R0
ADC R20, R1
ADC R21, R24
ADC R25, R23
MUL R16, R18
MOVW R18,R22
ADD R20, R0
ADC R21, R1
ADC R22, R25


MUL R2, R7
MOVW R16, R0
MUL R2, R5
MOVW R14, R0
MUL R2, R6
ADD R15, R0
ADC R16, R1
ADC R17, R23

MUL R3, R7
MOVW R24, R0
MUL R3, R5
```

```
ADD R15, R0
ADC R16, R1
ADC R17, R24
ADC R25, R23
MUL R3, R6
ADD R16, R0
ADC R17, R1
ADC R18, R25

MUL R4, R7
MOVW R24, R0
MUL R4, R5
ADD R16, R0
ADC R17, R1
ADC R18, R24
ADC R25, R23
MUL R4, R6
ADD R17, R0
ADC R18, R1
ADC R19, R25


EOR R26, R27
COM R26

ADD R8, R11
ADC R9, R12
ADC R10, R13
ADC R11, R20
ADC R12, R21
ADC R13, R22

ADC R23, R26
MOV R0, R23
ASR R0

EOR R14, R26
EOR R15, R26
EOR R16, R26
EOR R17, R26
EOR R18, R26
EOR R19, R26
ADD R26, R26

ADC  R8, R14
ADC  R9, R15
ADC R10, R16
ADC R11, R17
ADC R12, R18
ADC R13, R19

ADC R20, R23
ADC R21, R0
ADC R22, R0

STD Z+3, R8
STD Z+4, R9
STD Z+5, R10
STD Z+6, R11
STD Z+7, R12
STD Z+8, R13
STD Z+9, R20
STD Z+10, R21
STD Z+11, R22
```