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 susceptible 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 significantly 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.
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]. 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 $A = (a_0, \ldots, a_5)$ and $B = (b_0, \ldots, b_5)$. The basic idea of the multiplication of $A$ and $B$ is to

---

1Note 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:

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

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 the 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 \( A_\ell, A_h, B_\ell, \) 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 properly 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.:

(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:

See books/models/y86 of the ACL2 distribution.
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:

```lisp
(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:

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

```lisp
(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))
         ???))

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

(defun 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)))))

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.

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

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:

```
(defun-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/
Table 2: Verification times for the Karatsuba multiplication.

<table>
<thead>
<tr>
<th>byte</th>
<th>create CNF</th>
<th># variables</th>
<th># clauses</th>
<th># vars fraig</th>
<th># clauses fraig</th>
<th>time fraig</th>
<th>time SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1893</td>
<td>6027</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4483</td>
<td>14858</td>
<td>5618</td>
<td>22182</td>
<td>1800</td>
<td>&gt; 3 days</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>7900</td>
<td>26429</td>
<td>9967</td>
<td>39562</td>
<td>1800</td>
<td>&gt; 3 days</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>12599</td>
<td>42308</td>
<td>4887</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>17370</td>
<td>58406</td>
<td>6669</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>20531</td>
<td>69079</td>
<td>7856</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>420</td>
<td>21962</td>
<td>73919</td>
<td>8400</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2400</td>
<td>22290</td>
<td>75034</td>
<td>8529</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>18000</td>
<td>22475</td>
<td>75665</td>
<td>8596</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>36000</td>
<td>22565</td>
<td>75967</td>
<td>8626</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

\[
\text{(unsigned-byte-p 48 m)}
\]

:concl

\[
\text{(implies (not (equal (k\_Z3 m n)}
\[
\text{(c\_Z3 m n)))}
\]

\[
\text{(not (equal (k\_Z5 m n}}
\[
\text{(c\_Z5 m n))))}
\]

:g-bindings `{((n ,(gl::g-int 0 2 49))
\[
\text{(m ,(gl::g-int 1 2 49))))}
\]

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:

\[
\text{(defthm k\_z3=-c\_Z3}
\]

\[
\text{(implies (and (unsigned-byte-p 48 n)}
\[
\text{(unsigned-byte-p 48 m))}
\]

\[
\text{(equal (k\_Z3 m n)}
\[
\text{(c\_Z3 m n))}}
\]

: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.

<table>
<thead>
<tr>
<th>byte</th>
<th>create CNF</th>
<th># variables</th>
<th># clauses</th>
<th># vars fraig</th>
<th># clauses fraig</th>
<th>time fraig</th>
<th>time SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>17370</td>
<td>58406</td>
<td>6669</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>20531</td>
<td>69079</td>
<td>7856</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>42</td>
<td>21962</td>
<td>73919</td>
<td>8400</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>243</td>
<td>22290</td>
<td>75034</td>
<td>8529</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>18021</td>
<td>22475</td>
<td>75665</td>
<td>8596</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>36003</td>
<td>22565</td>
<td>75967</td>
<td>8626</td>
<td>2</td>
<td>3000</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 3: Verification times for a second version of the Karatsuba multiplication.

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

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


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 \( X+0, X+1, \ldots, X+5 \); input \( B \) in \( Y+0, Y+1, \ldots, Y+5 \); result \( R \) in \( Z+0, Z+1, \ldots, Z+11 \).

<table>
<thead>
<tr>
<th>CLR R5</th>
<th>STD Z+3, R3</th>
<th>ADC R2, R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R6, X+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R7, X+</td>
<td>MUL R6, R16</td>
<td>ADC R4, R1</td>
</tr>
<tr>
<td>LD R8, X+</td>
<td>ADD R4, R0</td>
<td>ADC R2, R5</td>
</tr>
<tr>
<td>LD R9, X+</td>
<td>ADC R2, R1</td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>LD R10, X+</td>
<td>ADD R2, R1</td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>LD R11, X+</td>
<td>MUL R7, R15</td>
<td>ADC R4, R1</td>
</tr>
<tr>
<td>LDD R12, Y+0</td>
<td>ADD R4, R0</td>
<td>ADC R2, R5</td>
</tr>
<tr>
<td>LDD R13, Y+1</td>
<td>ADC R2, R1</td>
<td>STD Z+6, R3</td>
</tr>
<tr>
<td>LDD R14, Y+2</td>
<td>ADC R3, R5</td>
<td></td>
</tr>
<tr>
<td>LDD R15, Y+3</td>
<td>MUL R8, R14</td>
<td></td>
</tr>
<tr>
<td>LDD R16, Y+4</td>
<td>ADD R4, R0</td>
<td>MUL R8, R17</td>
</tr>
<tr>
<td>LDD R17, Y+5</td>
<td>ADC R2, R1</td>
<td>ADD R4, R0</td>
</tr>
<tr>
<td>MUL R6, R12</td>
<td>ADC R3, R5</td>
<td>ADC R2, R1</td>
</tr>
<tr>
<td>MOV R4, R1</td>
<td></td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>STD Z+0, R0</td>
<td>MUL R10, R12</td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>CLR R2</td>
<td>ADD R4, R0</td>
<td>MUL R10, R15</td>
</tr>
<tr>
<td>CLR R3</td>
<td>ADC R2, R1</td>
<td>ADC R4, R0</td>
</tr>
<tr>
<td>STD Z+1, R4</td>
<td>ADC R3, R5</td>
<td>ADC R2, R1</td>
</tr>
<tr>
<td>MUL R6, R13</td>
<td>STD Z+4, R4</td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>ADD R4, R0</td>
<td>CLR R4</td>
<td>MUL R11, R14</td>
</tr>
<tr>
<td>ADC R2, R1</td>
<td></td>
<td>ADC R4, R0</td>
</tr>
<tr>
<td>ADC R3, R1</td>
<td>MUL R6, R17</td>
<td>ADC R2, R1</td>
</tr>
<tr>
<td>ADC R4, R0</td>
<td>ADD R2, R0</td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>ADC R5, R5</td>
<td>ADC R3, R1</td>
<td>STD Z+7, R4</td>
</tr>
<tr>
<td>ADC R6, R5</td>
<td>ADC R4, R5</td>
<td>CLR R4</td>
</tr>
<tr>
<td>STD Z+2, R4</td>
<td>MUL R7, R16</td>
<td>MUL R9, R17</td>
</tr>
<tr>
<td>CLR R4</td>
<td>ADD R2, R0</td>
<td>ADD R2, R0</td>
</tr>
<tr>
<td>ADC R3, R1</td>
<td>ADC R3, R1</td>
<td>ADC R5, R1</td>
</tr>
<tr>
<td>ADC R4, R5</td>
<td>ADC R4, R0</td>
<td>ADC R4, R0</td>
</tr>
<tr>
<td>ADC R6, R15</td>
<td>ADC R3, R1</td>
<td>ADC R3, R5</td>
</tr>
<tr>
<td>ADD R3, R0</td>
<td>MUL R8, R14</td>
<td>ADC R4, R5</td>
</tr>
<tr>
<td>ADC R4, R1</td>
<td>ADD R3, R0</td>
<td>ADD R3, R0</td>
</tr>
<tr>
<td>ADC R5, R5</td>
<td>ADD R3, R0</td>
<td>STD Z+9, R3</td>
</tr>
<tr>
<td>ADC R6, R13</td>
<td>ADC R2, R1</td>
<td>STD Z+9, R3</td>
</tr>
<tr>
<td>ADD R3, R0</td>
<td>MUL R8, R16</td>
<td>ADD R4, R0</td>
</tr>
<tr>
<td>ADC R4, R1</td>
<td>ADD R3, R0</td>
<td>ADC R2, R1</td>
</tr>
<tr>
<td>ADC R5, R5</td>
<td>ADC R4, R1</td>
<td>STD Z+10, R4</td>
</tr>
<tr>
<td>ADC R6, R12</td>
<td>ADC R2, R5</td>
<td>STD Z+11, R2</td>
</tr>
<tr>
<td>ADD R3, R0</td>
<td>MUL R9, R15</td>
<td></td>
</tr>
<tr>
<td>ADC R4, R1</td>
<td>ADD R3, R0</td>
<td></td>
</tr>
<tr>
<td>ADC R5, R5</td>
<td>ADC R4, R1</td>
<td></td>
</tr>
</tbody>
</table>

Verification of Optimised 48-bit Multiplications on AVR
Listing 2 Optimised multiplication of two 48-bit integers. Input $A$ in $X+0, X+1, \ldots, X+5$; input $B$ in $Y+0, Y+1, \ldots, Y+5$; result $R$ in $Z+0, Z+1, \ldots, Z+11$.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R2, X+</td>
<td>ADC R20,R0</td>
</tr>
<tr>
<td>LD R3, X+</td>
<td>ADC R21,R1</td>
</tr>
<tr>
<td>LD R4, X+</td>
<td>ADC R22,R2</td>
</tr>
<tr>
<td>LD R5, X+</td>
<td>ADC R23,R3</td>
</tr>
<tr>
<td>LD R6, X+</td>
<td>ADC R24,R4</td>
</tr>
<tr>
<td>LD R7, X+</td>
<td>ADC R25,R5</td>
</tr>
<tr>
<td>LD R8, Y+</td>
<td>MUL R4,R10</td>
</tr>
<tr>
<td>LD R9, Y+</td>
<td>ADD R16,R0</td>
</tr>
<tr>
<td>LD R10, Y+</td>
<td>ADC R17,R1</td>
</tr>
<tr>
<td>LD R11, Y+</td>
<td>ADC R18,R2</td>
</tr>
<tr>
<td>LD R12, Y+</td>
<td>ADC R19,R3</td>
</tr>
<tr>
<td>LD R13, Y+</td>
<td>ADC R20,R4</td>
</tr>
<tr>
<td>CLR R20</td>
<td>MUL R4,R8</td>
</tr>
<tr>
<td>CLR R21</td>
<td>ADD R17,R0</td>
</tr>
<tr>
<td>MOVW R22, R20</td>
<td></td>
</tr>
<tr>
<td>MOVW R24, R20</td>
<td></td>
</tr>
<tr>
<td>MUL R2,R10</td>
<td>ADC R21,R1</td>
</tr>
<tr>
<td>MOVW R16, R0</td>
<td>ADC R22,R2</td>
</tr>
<tr>
<td>MUL R2,R8</td>
<td>MUL R4,R11</td>
</tr>
<tr>
<td>MOVW R14, R0</td>
<td>MOVW R26,R0</td>
</tr>
<tr>
<td>MUL R2,R9</td>
<td>MUL R4,R13</td>
</tr>
<tr>
<td>ADD R15,R0</td>
<td>ADD R19,R6</td>
</tr>
<tr>
<td>ADC R16,R1</td>
<td>ADC R20,R7</td>
</tr>
<tr>
<td>ADD R17,R1</td>
<td>ADD R19,R0</td>
</tr>
<tr>
<td>ADD R18,R0</td>
<td>ADD R20,R1</td>
</tr>
<tr>
<td>MUL R2,R12</td>
<td>ADC R21,R2</td>
</tr>
<tr>
<td>ADD R19,R0</td>
<td>ADC R20,R2</td>
</tr>
<tr>
<td>ADC R22,R1</td>
<td>ADC R21,R3</td>
</tr>
<tr>
<td>MUL R2,R8</td>
<td>ADC R22,R4</td>
</tr>
<tr>
<td>ADD R17,R0</td>
<td>ADC R23,R5</td>
</tr>
<tr>
<td>ADC R21,R1</td>
<td>MUL R5,R9</td>
</tr>
<tr>
<td>MUL R3,R10</td>
<td>ADD R18,R0</td>
</tr>
<tr>
<td>MOVW R26, R0</td>
<td>ADD R20,R0</td>
</tr>
<tr>
<td>MUL R3,R8</td>
<td>MUL R5,R10</td>
</tr>
<tr>
<td>ADD R15,R0</td>
<td>ADD R20,R12</td>
</tr>
<tr>
<td>ADC R16,R1</td>
<td>ADC R20,R27</td>
</tr>
<tr>
<td>ADD R17,R0</td>
<td>ADC R21,R0</td>
</tr>
<tr>
<td>MUL R2,R25</td>
<td>ADC R22,R1</td>
</tr>
<tr>
<td>MUL R3,R9</td>
<td>ADC R23,R5</td>
</tr>
<tr>
<td>ADD R16,R0</td>
<td>MUL R5,R11</td>
</tr>
<tr>
<td>ADC R17,R1</td>
<td>MUL R5,R13</td>
</tr>
<tr>
<td>ADD R18,R0</td>
<td>MUL R6,R8</td>
</tr>
<tr>
<td>ADC R19,R0</td>
<td>ADD R19,R0</td>
</tr>
<tr>
<td>ADC R20,R1</td>
<td>ADC R23,R1</td>
</tr>
<tr>
<td>ADC R21,R25</td>
<td>ADD R20,R0</td>
</tr>
<tr>
<td>MUL R3,R11</td>
<td>ADD R18,R0</td>
</tr>
<tr>
<td>MOVW R26,R0</td>
<td>MUL R6,R10</td>
</tr>
<tr>
<td>MUL R3,R13</td>
<td>MOVW R26,R0</td>
</tr>
<tr>
<td>ADD R18,R26</td>
<td>MUL R6,R8</td>
</tr>
<tr>
<td>ADC R19,R27</td>
<td>ADD R18,R0</td>
</tr>
</tbody>
</table>
Listing 3 Branchfree Karatsuba multiplication of two 48-bit integers. Input $A$ in $X+0, X+1, \ldots, X+5$; input $B$ in $Y+0, Y+1, \ldots, Y+5$; result $R$ in $Z+0, Z+1, \ldots, Z+11$.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR R22</td>
<td>EOR R2, R26</td>
</tr>
<tr>
<td>CLR R23</td>
<td>ADC R15, R0</td>
</tr>
<tr>
<td>MOVW R12, R22</td>
<td>EOR R3, R26</td>
</tr>
<tr>
<td>MOVW R20, R22</td>
<td>ADC R16, R1</td>
</tr>
<tr>
<td>LD R2, X+</td>
<td>EOR R4, R26</td>
</tr>
<tr>
<td>LD R3, X+</td>
<td>ADC R17, R24</td>
</tr>
<tr>
<td>LD R4, X+</td>
<td>EOR R5, R27</td>
</tr>
<tr>
<td>LD R5, Y+0</td>
<td>ADC R25, R23</td>
</tr>
<tr>
<td>LDD R6, Y+1</td>
<td>EOR R6, R27</td>
</tr>
<tr>
<td>LDD R7, Y+2</td>
<td>ADD R16, R0</td>
</tr>
<tr>
<td>MUL R2, R7</td>
<td>ADC R17, R1</td>
</tr>
<tr>
<td>MUL R2, R1</td>
<td>ADC R18, R24</td>
</tr>
<tr>
<td>MUL R2, R6</td>
<td>MUL R4, R5</td>
</tr>
<tr>
<td>ADD R9, R0</td>
<td>MUL R4, R6</td>
</tr>
<tr>
<td>ADC R10, R1</td>
<td>ADD R16, R0</td>
</tr>
<tr>
<td>ADC R11, R23</td>
<td>ADC R17, R1</td>
</tr>
<tr>
<td>MUL R3, R7</td>
<td>ADC R18, R24</td>
</tr>
<tr>
<td>MUL R2, R6</td>
<td>MOVW R24, R0</td>
</tr>
<tr>
<td>ADD R10, R0</td>
<td>ADC R19, R23</td>
</tr>
<tr>
<td>ADC R11, R1</td>
<td>ADD R11, R0</td>
</tr>
<tr>
<td>ADC R12, R15</td>
<td>ADC R20, R24</td>
</tr>
<tr>
<td>MUL R3, R7</td>
<td>ADC R25, R23</td>
</tr>
<tr>
<td>MUL R3, R5</td>
<td>MUL R4, R1</td>
</tr>
<tr>
<td>ADD R9, R0</td>
<td>ADD R8, R11</td>
</tr>
<tr>
<td>ADC R10, R1</td>
<td>ADC R9, R12</td>
</tr>
<tr>
<td>ADC R11, R14</td>
<td>MUL R14, R19</td>
</tr>
<tr>
<td>MUL R3, R6</td>
<td>ADC R10, R13</td>
</tr>
<tr>
<td>ADD R10, R0</td>
<td>MOVW R24, R0</td>
</tr>
<tr>
<td>ADC R11, R1</td>
<td>ADC R11, R20</td>
</tr>
<tr>
<td>ADC R12, R15</td>
<td>MUL R15, R17</td>
</tr>
<tr>
<td>MUL R4, R7</td>
<td>ADC R12, R21</td>
</tr>
<tr>
<td>MUL R4, R5</td>
<td>ADC R13, R22</td>
</tr>
<tr>
<td>ADD R9, R0</td>
<td>MUL R4, R6</td>
</tr>
<tr>
<td>ADC R10, R0</td>
<td>ADD R15, R0</td>
</tr>
<tr>
<td>ADC R11, R1</td>
<td>EOR R14, R26</td>
</tr>
<tr>
<td>ADC R12, R14</td>
<td>EOR R16, R26</td>
</tr>
<tr>
<td>ADC R15, R23</td>
<td>EOR R17, R26</td>
</tr>
<tr>
<td>MUL R4, R6</td>
<td>MOVW R24, R0</td>
</tr>
<tr>
<td>ADD R11, R0</td>
<td>EOR R19, R26</td>
</tr>
<tr>
<td>ADC R12, R1</td>
<td>ADD R26, R26</td>
</tr>
<tr>
<td>ADC R13, R15</td>
<td>ADC R8, R14</td>
</tr>
<tr>
<td>STD Z+0, R8</td>
<td>ADC R9, R15</td>
</tr>
<tr>
<td>STD Z+1, R9</td>
<td>MOVW R18, R22</td>
</tr>
<tr>
<td>STD Z+2, R10</td>
<td>ADD R20, R0</td>
</tr>
<tr>
<td>LD R14, X+</td>
<td>ADD R21, R1</td>
</tr>
<tr>
<td>LD R15, X+</td>
<td>ADC R13, R19</td>
</tr>
<tr>
<td>LD R16, X+</td>
<td>ADC R20, R23</td>
</tr>
<tr>
<td>LDD R17, Y+3</td>
<td>ADC R21, R0</td>
</tr>
<tr>
<td>LDD R18, Y+4</td>
<td>MUL R2, R7</td>
</tr>
<tr>
<td>LDD R19, Y+5</td>
<td>ADC R22, R0</td>
</tr>
<tr>
<td>SUB R2, R14</td>
<td>MUL R2, R5</td>
</tr>
<tr>
<td>SBC R3, R16</td>
<td>ADD R15, R0</td>
</tr>
<tr>
<td>SBC R4, R16</td>
<td>STD Z+3, R8</td>
</tr>
<tr>
<td>SBC R26, R26</td>
<td>STD Z+4, R9</td>
</tr>
<tr>
<td>SUB R5, R17</td>
<td>STD Z+5, R10</td>
</tr>
<tr>
<td>SBC R6, R18</td>
<td>STD Z+6, R11</td>
</tr>
<tr>
<td>SBC R7, R19</td>
<td>STD Z+7, R12</td>
</tr>
<tr>
<td>SBC R27, R27</td>
<td>STD Z+8, R13</td>
</tr>
<tr>
<td></td>
<td>STD Z+9, R20</td>
</tr>
<tr>
<td></td>
<td>STD Z+10, R21</td>
</tr>
<tr>
<td></td>
<td>STD Z+11, R22</td>
</tr>
</tbody>
</table>