Verification of Machine Code Implementations of Arithmetic Functions for Cryptography

Magnus O. Myreen, Michael J. C. Gordon

Computer Laboratory, University of Cambridge, Cambridge, UK
{magnus.myreen,mike.gordon}@cl.cam.ac.uk

Abstract. This report presents a methodology and some preliminary results for verification of machine code implementations of cryptographic operations. Modularity and reusability of proofs is emphasised.

1 Introduction

Cryptography algorithms such as RSA, Diffie-Hellman and Elliptic Curve Cryptography require efficient operations over large natural numbers (>300 bits). Implementations of operations over large numbers are supported by processors through special purpose instruction (sometimes even special purpose coprocessors). Parts of cryptographic operations are therefore often written directly in machine code to make use of the special purpose instructions for high performance. The fact that parts of cryptographic systems are implemented directly in machine code easily lead to *ad hoc* correctness proofs.

In this report we present a methodology and provide theorems that aim to make verification of different machine code implementations of cryptographic operations manageable and reusable (even across different architectures). Two case studies are presented; the first one illustrates our methodology on a straightforward implementation of addition, the second one presents the development of an efficient machine code implementation of Montgomery multiplication (multiplication modulo a large prime number). The work presented in this report has been carried out within the HOL4 theorem prover.

Affeldt and Marti's paper [1] on verification of SmartMIPS implementations of arithmetic functions (including Montgomery multiplication) motivated our work. They propose an approach where one augments the machine language with familiar while-loop-like constructs in order to reason about code using a conventional Hoare logic; once the code with while loops is verified the loops are removed through a light-weight compilation. Their approach seems impractical as their proofs seem to be one-off proofs that are specific to SmartMIPS and cluttered with assertion statements. In this work we attempt to improve on their approach and use a Hoare logic developed for reasoning directly at the level of machine code [15, 14].

2 Methodology

The methodology we propose strives towards modularity and reusability of proofs by a three phase strategy: In order to verify or construct a correct implementation of a particular algorithm one will first (i) verify a basic functional version of the algorithm, then (ii) combine and unroll parts of the verified program to build a correct but more realistic functional version of the algorithm; and finally (iii) prove that a piece of machine code implements the functional program produced in step (ii).

The benefit of this approach is that step (i) and (ii) are independent of the target machine architecture and can hence be reused. Also in our experience it is rather easy to prove that the functional programs from step (iii) are implemented by the proposed optimised machine code.

The following table gives some indication that our approach may require less effort than the approach proposed by Affeldt and Marti. The table below gives the number of lines of proof script required for the verification of functional and machine code implementations of the algorithms listed in the left column.

algorithm	TFL	ARM	total	A&M
(script at top of file)	120	205	**325**	—
addition	135	68	**203**	**835**
subtraction	160	68	**228**	**1473**
Montgomery multiplication	945	691	**1636**	**3881**
Montgomery exponentiation	230	487	**717**	—

TFL — number of lines of proof script required in step (i) and (ii)
ARM — number of lines of proof script required in step (iii) for ARM machine code
total — sum of TFL and ARM columns
A&M — total number for Affeldt and Marti's SmartMIPS implementations [1]

A comparison between our numbers and those of Affeldt and Marti is only approximate, since Affeldt and Marti work in Coq rather than HOL4.

3 Case studies

The general methodology is presented first on the simple case of addition and then on the less obvious algorithm for Montgomery multiplication.

3.1 Addition

When constructing a functional implementation of a particular algorithm, we start by defining the elementary operations that we know how to implement in machine code (or can expect to find in the instruction set). For addition we require an operation that performs one step of a large addition. Define `single_add` to take two bit strings of length α and a boolean carry bit as input and have it

return the sum of the inputs as well as a boolean carry-out. Here `w2n` converts a bit string of length α to the natural number it represents[1].

```
single_add (x:α word) (y:α word) c =
   (x + y + if c then 1w else 0w,
    2^α ≤ w2n x + w2n y + if c then 1 else 0)
```

The functional program for addition is then a simple loop:

```
mw_add [] ys c = ([],c)
mw_add (x::xs) (y::ys) c =
  let (z,c1) = single_add x y c in
  let (zs,c2) = mw_add xs ys c1 in
    (z::zs,c2))
```

In order to state the specification for the functional implementations, we define `n2mw` i n to be a list (of length i) of bits strings (of length α) such that the concatenation of these bit strings represent the $i \times \alpha$ least-significant bits of the binary representation of the natural number n.

```
n2mw 0 n     = []
n2mw (i+1) n = n2w n :: n2mw i (n DIV dimword (:α))
```

The specification for `mw_add` is the following. Let `b2n c = if c then 1 else 0` and `add_carry_out i m n c = 2^(i×α) ≤ m MOD 2^(i×α) + n MOD 2^(i×α) + b2n c`.

```
∀i p m n c.
  mw_add (n2mw i m) (n2mw (i + p) n) c =
    (n2mw i (m + n + b2n c), add_carry_out i m n c)
```

The functional program `mw_add` can be implemented in ARM machine code using a single loop that tests for the end of the sequence using the `teq` instruction (an instruction that leaves the carry status bit untouched).

```
     add  t, a, t            ; initilse t
L:   ldr  x, [a], #4         ; load value at a into x, then increment a
     ldr  y, [b], #4         ; load value at b into y, then increment b
     adcs x, x, y            ; add x and y using carry from status bits
     str  x, [c], #4         ; store x into location c, then increment c
     teq  a, t               ; test a = t
     bne  L                  ; jump to L, if a ≠ t
```

One can prove that the above code implements `mw_add` by induction on the length of the first argument to `mw_add` together with the rules of the Hoare logic presented in Myreen and Gordon [15]. The specification which states that the ARM code implements `mw_add` is shown in Appendix A.

The specification is made legible if we introduce the following definition. Let `bignum a i n` state that register `a` holds an aligned address which points at

[1] Anthony Fox's HOL4 theory of bit strings, called `wordsTheory`, is used to represent bit strings of a length given as part of the type. The theory is based on a formalisation of finite Cartesian products, which is originally due to John Harrison [9].

some location where the sequence `n2mw i n` is stored. Let `bignum' a i n` be the same except that the address points at the location immediately following the sequence `n2mw i n`.

```
bignum a i n = ∃x. R30 a x * ms x (n2mw i n)
bignum' a i n = ∃x. R30 a (x + n2w i) * ms x (n2mw i n)
```

Using `bignum` the above ARM code has the following Hoare-triple specification.

```
{ bignum a i m * bignum b i n * bignum c i _ *
  R x _ * R y _ * R t (n2w (4 × i)) * carry_status c * (i ≠ 0) }
   --- code as above ---
{ bignum' a i m * bignum' b i n * bignum' c i (m + n + b2n c) *
  R x _ * R y _ * R t _ * carry_status (add_carry_out i m n c)) }
```

3.2 Montgomery multiplication

Montgomery multiplication is an algorithm that is commonly used in implementations that require multiplication modulo a large prime. Given n, r and r' such that $n < r$, $gcd(n, r) = 1$ and $(r \times r')$ mod $n = 1$, Montgomery multiplication *monprod* calculates the product of a, b and r' modulo n:

$$monprod(a, b, n) = (a \times b \times r') \bmod n, \qquad \text{for } a < n \text{ and } b < n.$$

Let $\bar{a}$ denote $(a \times r)$ mod n. Montgomery multiplication calculates the product of values represented as $\bar{a}$ and $\bar{b}$.

$$\begin{aligned} monprod(\bar{a}, \bar{b}, n) &= (\bar{a} \times \bar{b} \times r') \bmod n \\ &= (a \times r \times b \times r \times r') \bmod n \\ &= (a \times r \times b \times 1) \bmod n \\ &= (a \times b \times r) \bmod n \\ &= \overline{(a \times b)} \end{aligned}$$

The conversion from $\bar{a}$ into a mod n can be done using *monprod*.

$$\begin{aligned} monprod(\bar{a}, 1, n) &= (\bar{a} \times 1 \times r') \bmod n \\ &= (a \times r \times r') \bmod n \\ &= a \bmod n \end{aligned}$$

The conversion from a into $\bar{a}$ requires an implementation of modulus.

We will refrain from describing the details of Montgomery multiplication here for that is described well elsewhere: Montgomery describes the basic algorithm elegantly in [13], Dussé *et al.* [7] and Certin Kaya Koc *et al.* [10] describe optimisations; Instead we will just note that the following implementation of *monprod* called `mw_monprod`,

```
mw_mul [] y c = ([],c)
mw_mul (x::xs) y c =
   let (z,c1) = single_mul x y c in
   let (zs,c2) = mw_mul xs y c1 in
     (z::zs,c2)

mw_add_mul x ys zs =
   FST (mw_add zs (FST (mw_mul ys x 0w)) F)

mw_monmult [] ys ns m zs      = zs
mw_monmult (x::xs) ys ns m zs =
   let u = (x * HD ys + HD zs) * m in
   let zs = mw_add_mul x ys (zs ++ [0w]) in
   let zs = mw_add_mul u ns zs in
     mw_monmult xs ys ns m (TL zs)

mw_monprod xs ys ns m zs =
   let zs = mw_monmult xs ys ns m zs in
   let (zs',c) = mw_sub zs ns T in
     (if c then zs' else zs)
```

satisfies the specification given below. Let `montgomery_vars` n n' r r' sum up that r is even, $n < r$ and $gcd(n, r) = 1$ with witnesses n' and r'.

```
montgomery_vars n n' r r' = (r × r' - n × n' = 1) ∧ n < r ∧ EVEN r

∀a b n n' r' i.
    montgomery_vars n n' (2^(i×α)) r' ∧ a ≤ n ∧ b ≤ n ⇒
    (mw_monprod (n2mw i a) (n2mw (i+2) b) (n2mw (i+2) n) (n2w n')
       (n2mw (i+1) 0) =
     n2mw (i+1) ((a × b × r') MOD n))
```

Optimisation 1. The first version of Montgomery multiplication `mw_monprod` was reasonably easy to prove (≈400 lines of proof script), but the implementation is unsatisfactory in many ways. In what follows we will successively improve the functional implementation towards functional implementations that calculate the value in a less wasteful manner.

The first and obvious improvement is to combine the two occurrences of `mw_add_mul` into one function. A function `mw_add_mul_mul` was constructed which calculates the same value as two applications of `mw_add_mul`. An unrolling of the new function is now the body for function `mw_monmult2`.

```
mw_monmult2 [] ys ns m zs = zs
mw_monmult2 (x::xs) (y::ys) (n::ns) m (z::zs) =
   let u = (x * y + z) * m in
   let (w1,c1,b1) = double_mul_add y n x u 0w 0w z in
   let (ws,c2,b2) = mw_add_mul_mul ys ns zs x u c1 b1 in
   let (w3,c3,b3) = double_mul_add 0w 0w x u c2 b2 (LAST zs) in
   let (w4,c4,b4) = double_mul_add 0w 0w x u c3 b3 0w in
```

```
   let zs = ws ++ [w3; w4] in
      mw_monmult2 xs (y::ys) (n::ns) m zs
```

The new definition of `mw_monprod` satisfies the same specification as the original except that now both occurrences of `+2` are removed.

Optimisation 2. For the second optimisation we note that the result is one word too long, the result is returned as a list of `i+1` words while the result fits into `i` words. By looking at the implementation one observes that the last element of the list `zs` is always handled separately from the rest of `zs`. This suggests that it may be beneficial to keep the last element separate throughout the computation (in the machine code implementation we would like to keep the last element in a register). The new implementation simply splits the last element off `zs`:

```
mw_monmult3 [] ys ns m (zs,z') = (zs,z')
mw_monmult3 (x::xs) (y::ys) (n::ns) m (z::zs,z') =
   let u = (x * y + z) * m in
   let (w1,c1,b1) = double_mul_add y n x u 0w 0w z in
   let (ws,c2,b2) = mw_add_mul_mult ys ns zs x u c1 b1 in
   let (w3,c3,b3) = double_mul_add 0w 0w x u c2 b2 z' in
   let (w4,c4,b4) = double_mul_add 0w 0w x u c3 b3 0w in
   let (zs,z') = (ws ++ [w3],w4) in
     mw_monmult3 xs (y::ys) (n::ns) m (zs,z')

mw_monprod3 xs ys ns m (zs,z) =
   let (zs,z) = mw_monmult3 xs ys ns m (zs,z) in
   let (zs',c) = mw_sub zs ns in
   let c = SND (single_sub z 0w c) in
     (if c then zs' else zs)
```

The specification of `mw_monprod3` is the same as the one for `mw_monprod2` except that the result is now of length `i` and `(n2mw (i+1) 0)` is replaced by `(n2mw i 0,0w)`.

Optimisation 3. It would be unfortunate if the final implementation requires the array implementing `zs` to be initialised to zero, since that is likely to require a separate loop, which writes zero into each element of an array before calling the algorithm. The requirement of a zeroed input can be removed by unrolling the main loop once in order to break out the part of the program which can assume that zs is zero, and hence can be implemented more efficiently (fewer load instructions). The new implementation unrolls `mw_monmult3` once.

```
mw_moninit ys ns x m =
  mw_monmult3_step ys ns (MAP (λx.0w) ys, 0w) x m

mw_monprod4 (x::xs) ys ns m =
   let (zs,z) = mw_moninit ys ns x m in
   let (zs,z) = mw_monmult3 xs ys ns m (zs,z) in
```

```
let (zs',c) = mw_sub zs ns T in
let c' = SND (single_sub z 0w c) in
  (if c' then zs' else zs)
```

The new implementation satisfies the following specification.

```
∀a b n n' r' i.
   montgomery_vars n n' (2^(i×α)) r' ∧ a ≤ n ∧ b ≤ n ⇒
   (mw_monprod4 (n2mw i a) (n2mw i b) (n2mw i n) (n2w n') =
    n2mw i ((a × b × r') MOD n))
```

ARM implementation. An ARM implementation of `mw_monprod4` has been verified using the technique used for addition. The only difference is that the proof is slightly longer since the function for Montgomery multiplication is longer. The parts of the proof that correspond to the unrolled parts of loops are often repetitions of the verification of the loop body with slight modifications. These proofs are easily constructed: one copies the proof of the code corresponding to the body of the function and makes minor alterations, where the code differs (the proof becomes long but the effort is minimal).

The resulting specification for Montgomery multiplication is given below. It uses `bignum` as defined earlier and states the following: given that register `a` holds the address of an array of length `i` storing the natural number `p` (modulo the capacity of the array), register `b` holds an address of the stored number `q`, register `c` holds an address of number `n`, and `d` points at some array of length `i`; registers `v1`-`v10` have some values and the stack holds `n2w n'` and `n2w (4 * i)`; if all of the above holds, then the program will execute so as to store `(p × q × r') MOD n` in the array pointed to by `b`, without using or altering any resources out side of the scope of the specification. The verified code is given in Appendix B. The difference between pre- and postcondition is highlighted using `boxes`.

```
montgomery_vars n n' (2^(i×32)) r' ∧ p < n ∧ q < n ∧ 1 < i ⇒

{ bignum a i p * bignum b i [q] *
  bignum c i n * bignum d i _ *
  R v1 _ * R v2 _ * R v3 _ * R v4 _ * R v5 _ * R v6 _ * R v7 _ *
  R v8 _ * R v9 _ * R v10 _ * stack sp [n2w n'; n2w (4 * i)] 1 * S _ }
   --- code as given in appendix B ---
{ bignum a i p * bignum b i [((p × q × r') MOD n)] *
  bignum c i n * bignum d i _ *
  R v1 _ * R v2 _ * R v3 _ * R v4 _ * R v5 _ * R v6 _ *  R v7 _ *
  R v8 _ * R v9 _ * R v10 _ * stack sp [n2w n'; n2w (4 * i)] 1 * S _ }
```

The specification reveals that the implementation does indeed perform multiplication modulo `n`, if we instantiate `p` and `q` to `(p × r) MOD n` and `(q × r) MOD n`, respectively:

```
montgomery_vars n n' r r' ∧ (r = 2^(i×32)) ∧ 1 < i ⇒
```

```
{ bignum a i ((p × r) MOD n) * bignum b i ((q × r) MOD n) *
  bignum c i n * bignum d i _ *
  R v1 _ * R v2 _ * R v3 _ * R v4 _ * R v5 _ * R v6 _ * R v7 _ *
  R v8 _ * R v9 _ * R v10 _ * stack sp [n2w n'; n2w (4 * i)] 1 * S _ }
   --- code as given in appendix B ---
{ bignum a i ((p × r) MOD n) * bignum b i ((p × q) × r) MOD n) *
  bignum c i n * bignum d i _ *
  R v1 _ * R v2 _ * R v3 _ * R v4 _ * R v5 _ * R v6 _ *  R v7 _ *
  R v8 _ * R v9 _ * R v10 _ * stack sp [n2w n'; n2w (4 × i)] 1 * S _ }
```

4 Related Work

Many have worked on verification of machine code programs. Some early work was done by Maurer [12], Clutterbuck and Carré [6] and Bevier [3]. Boyer and Yu [4] did impressive pioneering work on verifying machine code written for a commercial processor: they verified programs using the bare operational semantics of a model of the Motorola MC68020. Projects on proof-carrying code (PCC) [16] and particularly foundation PCC [2] have ignited new interest in verification of low-level code. Of work on PCC, Tan and Appel's work [18] is relevant to this paper: they use a Hoare logic to reason about a detailed model of the Sparc machine language. As for most work on PCC, their aim is to address safety properties that can be proved automatically (e.g. type safety). Ni *et al* [17], Feng *et al* [8] and Cai *et al* [5] work towards verifying parts of operating systems and runtime environments.

The the best of our knowledge Affeldt and Marti [1] are the only ones who attempt to verify machine code implementations of cryptographic operations. Li *et al* [11] have developed a proof-producing compiler that they intend to use to generate efficient machine code implementations of cryptographic algorithms. In the future we hope to collaborate with Li *et al*.

5 Further Work

Our aim is to develop machine code implementations of all the operations required for an implementation of elliptic curve cryptography. Another goal is to investigate how these ideas can be applied to architectures other than ARM. We believe that the Hoare logic, which was used to reason about ARM machine code, is readily instantiated to other models of instruction set architectures.

References

1. Reynald Affeldt and Nicolas Marti. An approach to formal verification of arithmetic functions in assembly. In *11th Annual Asian Computing Science Conference (ASIAN 2006)*, Dec 2006. LNCS, Springer-Verlag.
2. Andrew W. Appel. Foundational proof-carrying code. In *LICS*, 2001.

3. William R. Bevier. *A verified operating system kernel*. PhD thesis, University of Texas at Austin, 1987.
4. Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.
5. Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, 2007.
6. D. L. Clutterbuck and B. A. Carré. The verification of low-level code. *Software Engineering Journal*, 3:97–111, 1988.
7. Stephen R. Dussé and Jr. Burton S. Kaliski. A cryptographic library for the Motorola DSP56000. In *EUROCRYPT '90: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 230–244. Springer-Verlag, 1991.
8. Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, 2006.
9. John Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129, Oxford, UK, 2005. Springer-Verlag.
10. Cetin Kaya Koc, Tolga Acar, and Jr. Burton S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.
11. Guodong Li, Scott Owens, and Konrad Slind. A proof-producing software compiler for a subset of higher order logic. In *European Symposium on Programming (ESOP 2007)*, LNCS, pages 205–219. Springer-Verlag, 2007.
12. W. D. Maurer. Proving the correctness of a flight-director program for an airborne minicomputer. In *SIGMINI '76: Proceedings of the ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*, pages 103–108, New York, NY, USA, 1976. ACM Press.
13. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
14. Magnus O. Myreen, Anthony C.J. Fox, and Michael J.C. Gordon. A Hoare logic for ARM machine code. In *To appear in IPM International Symposium on Fundamentals of Software Engineering (FSEN 2007)*, LNCS. Springer-Verlag, 2007.
15. Magnus O. Myreen and Michael J.C. Gordon. A Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, LNCS, pages 568–582. Springer-Verlag, 2007.
16. George C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.
17. Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic system code: Machine context management. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To Appear.
18. Gang Tan and Andrew W. Appel. A compositional logic for control flow. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.

A Specification of Addition

```
{ R30 i ix * ms ix xs *
```

```
  R30 j jx * ms jx ys *
  R30 k kx * ms kx zs *
  R a _ * R b _ * R30 t (ix + wLENGTH xs) *
  carry_statuc c * cond (xs ≠ []) *
  cond (LENGTH xs = LENGTH zs) * cond (LENGTH zs ≤ LENGTH ys) }
[    ldr  a, [i], #4                          ;
     ldr  b, [j], #4                          ;
     adcs a, a, b                             ;
     str  a, [k], #4                          ;
     teq  i, t                                ;
     bne  -20                                 ]
{ R30 i (ix + wLENGTH xs) * ms ix xs *
  R30 j (jx + wLENGTH xs) * ms jx ys *
  R30 k (kx + wLENGTH xs) * ms kx (FST (mw_add xs ys c)) *
  R a _ * R b _ * R30 t (ix + wLENGTH xs) *
  carry_status (SND (mw_add xs ys c)) }
```

B ARM Implementation of Montgomery Multiplication

```
ldr v4,[sp,#4]      adcs v7,v4,#0       ldr v8,[d],#4       teq a,v1
ldr v2,[a]          adds v10,v6,v9      ldr v6,[b],#4       bne -152
add v5,b,v4         adcs v9,v7,#0       ldr v7,[c],#4       sub v5,v5,b
add v1,a,v4         teq b,v5            umlal v4,v8,v6,v2   ldr v4,[sp,#-4]
add a,a,#4          bne -52             mov v6,#0           add v1,d,v5
ldr v8,[sp]         ldr v2,[sp,#4]      umlal v6,v8,v7,v3   ldr v8,[d],#4
ldr v6,[b],#4       str v10,[d],#4      adds v8,v8,v10      ldr v7,[c],#4
ldr v7,[c],#4       str v9,[sp,#-4]     str v8,[d,#-8]      sub a,a,v5
umull v4,v9,v6,v2   sub b,b,v2          adcs v9,v4,v9       subs v8,v8,v7
mov v10,#0          sub c,c,v2          mov v4,#0           str v8,[b],#4
mul v3,v9,v8        sub d,d,v2          adcs v7,v4,#0       ldr v6,[d],#4
umlal v10,v9,v7,v3  ldr v2,[a],#4       adds v10,v6,v9      ldr v7,[c],#4
adds v10,v10,v4     ldr v8,[sp]         adcs v9,v7,#0       sbcs v6,v6,v7
mov v4,#0           ldr v9,[d],#4       teq b,v5            str v6,[b],#4
adcs v9,v4,#0       ldr v6,[b],#4       bne -56             teq d,v1
ldr v6,[b],#4       ldr v7,[c],#4       ldr v8,[sp,#-4]     bne -20
ldr v7,[c],#4       mov v4,#0           ldr v2,[sp,#4]      sub b,b,v5
umull v4,v8,v6,v2   umlal v4,v9,v6,v2   adds v8,v8,v10      sub c,c,v5
mov v6,#0           mov v10,#0          str v8,[d,#-4]      sub d,d,v5
umlal v6,v8,v7,v3   mul v3,v9,v8        adc v10,v9,#0       sbcs v4,v4,#0
adds v8,v8,v10      umlal v10,v9,v7,v3  sub b,b,v2          movcc v1,d
str v8,[d],#4       adds v10,v10,v4     sub c,c,v2          movcc d,b
adcs v9,v4,v9       mov v4,#0           sub d,d,v2          movcc b,v1
mov v4,#0           adcs v9,v4,#0       str v10,[sp,#-4]
```