A Methodology for High-Speed Software Implementations of Number-Theoretic Cryptosystems

Ç. K. Koç and T. Acar
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Technical Report, May 1997

Abstract

Because of their flexibility and cost effectiveness, software implementations of number-theoretic cryptographic algorithms (e.g., RSA and Diffie-Hellman) are often desired. In order to obtain the required level of performance (speed) on a selected platform, the developers turn to algorithm-level optimizations and assembly language programming. In this paper, we examine these implementation issues and propose a design methodology for obtaining high-speed implementations. We show that between the full assembler implementation and the standard C implementation, there is a design option in which only a small number of code segments (kernel operations) are written in assembler in order to obtain a significant portion of the speed increase gained by the full assembler implementation. We propose a small set of kernel operations which are as simple as $a \cdot b + c$, where the numbers a, b, c are 1-word integers. The results of our experiments on several processors are also summarized.

Key Words: Cryptography, high-speed arithmetic, modular exponentiation, assembler.

1 Introduction

The privacy and authenticity of information (whether it is stored on a single computer or shared on a network of computers) requires implementation of cryptographic functions. The basic functions of information security services are very few, and almost invariant. These include public-key cryptosystems, digital signatures, message digest functions, and secret-key cryptographic algorithms. The design and evaluation of these cryptographic functions is a special topic on its own, requiring advanced knowledge of combinatorial mathematics, number theory, abstract algebra, and theoretical computer science. There is also the subject of *algorithm engineering*, which refers to high-speed and cost effective hardware and software implementation of cryptographic algorithms [2].

Most public-key cryptographic functions require operations with elements of a large finite group, and need to be optimized on the chosen platform for high-speed implementation. As an example, the RSA cryptosystem [15] uses modular arithmetic operations (addition, multiplication, and exponentiation) with large integers, usually in the range of 512 to 1024 bits. Arithmetic with such large integers is time consuming, considering the fact that currently available processor have arithmetic logic units with wordsize up to 32 bits. The current fast implementations of the RSA signature algorithm with a 512-bit key size require on the order of 50 ms on a signal processor using advanced algorithmic techniques and assembly language programming [5]. However, the security

requirements are already pushing the key size to 1024 bits, at which a signature operation takes nearly half a second. This is not an acceptable speed for most networking applications. Other cryptographic algorithms, for example, the Diffie-Hellman key exchange method [4], the ElGamal public-key cryptosystem and digital signature algorithm [6], the Digital Signature Standard [13] also require implementation of modular arithmetic operations with large integers.

Software implementations of these number-theoretic cryptographic algorithms are often desired because of their flexibility and cost effectiveness. Furthermore, certain applications are suitable for software implementations because of their very nature. However, the performance is always an issue, requiring the designer to optimize these cryptographic algorithms on the selected processor. In order to exploit the architectural and arithmetic-logic properties of the processor, the designer needs to analyze and reformulate the underlying algorithms. Almost inevitably, the programming is performed in assembly language in order to take advantage of the specific architectural properties of the processor, and thus, to obtain the desired performance [5, 3, 11, 7, 10].

In this paper, we examine these implementation issues in order to determine the actual contribution of assembly level programming to the speed of the cryptographic algorithms. We show that between the full assembler implementation and the standard C implementation, there is a design option in which only a small number of code segments are written in assembler in order to obtain a significant portion of the speed increase gained by the full assembler implementation. These small code segments are the *kernel* of arithmetic operations for number-theoretic cryptographic algorithms, and have been obtained by analyzing several different implementations of these number-theoretic cryptosystems. We propose a small set (only 8) of such code segments, implementing certain arithmetic operations. Our experimental results on the Pentium PC show that by developing efficient assembly language implementation of these 8 operations as 'in-line' assembly code segments in the RSA cryptosystem, we can obtain a speedup of 2.33 over the standard C implementation. This speedup is about 64 % of the speedup obtained by a full assembler implementation.

2 Implementation Methods

The usefulness of a C implementation is due to its portability, i.e., the fact that the program can easily be compiled and executed on another machine. However, the C program may not execute as fast as an assembler program accomplishing the same computation since specific architectural properties of the new machine are not taken into account. On the other hand, efficient assembler software development requires full understanding of the sophisticated microprocessor architecture. The assembly language programmer needs to know the properties of the assembler instructions, the operation of multiple functioning units, the rules of instruction issuing, pipeline structure, and alignment rules, and also certain specific information about the cache and the memory structure. The development of assembler programs is a tedious, lengthy, and expensive task. It can be argued that a smart compiler will be aware of the detailed architectural issues, and thus, can produce more efficient code than a straightforward assembler implementation in many instances [1]. However, the cryptographic system developers often have to turn to assembly language programming in order to obtain the required speed. This gives the chance to reformulate the algorithm to be implemented by taking into account the architectural properties of the processor.

In this study, we consider the design options between the standard C and the assembly language for implementing the number-theoretic cryptographic algorithms. The properties of these two extreme design options are:

- Standard C: Portable, inexpensive, short development time, slow execution.
- Assembler: Not portable, expensive, long development time, fast execution.

There are several design options between these two ends. A particular design option involves the use of *non-standard* C data types such as `int64` or `long long`. We name this approach *C with Extended Types*. It turns out some amount of speed increase can be gained using such data types for number-theoretic cryptography. We are however limited to those platforms which support these data types and their particular definitions and uses. We gain a certain amount of speed by renouncing a small amount of portability.

As soon as the use of assembly language programming enters the picture, we loose portability. Once the portability is no longer an issue, the development cost of assembly language programming needs to be taken into account. One approach is the development of the entire code or the most crucial subroutines (e.g., the Montgomery multiplication and squaring) in the assembly language. This involves a great amount of assembly language programming, and we argue that it is not necessary in many instances. We propose a design approach in which only a specific kernel of operations need to be developed in assembler. In this paper, we evaluate and compare the following four approaches in terms of their resulting performance.

- Standard C code
- C with extended types
- Complete assembler
- C with kernel in assembler

2.1 Standard C Code

In the C language, operands of an arithmetic expression are converted to a common type before the computation [8], which is referred to as *converted type*. The value of a variable may be *truncated* to a less significant type, or it can be *promoted* to a more significant type. The high order bits are ignored in case of truncation. The promotion is performed using zero padding or sign extension. The result of an operation is also of the converted type. The truncation inhibits availability of high order bits of certain arithmetic operations in C, enforcing an emulation approach for precise calculations. For an addition of n operands using w-bit scalar type, maximum value of result is $n(2^w - 1)$. Assuming $n \leq 2^w$, the exact result can be stored in two w-bit words. Exact addition of such variables can be accomplished by computing lower and higher bits separately. The code segment given below adds two w-bit words to obtain the $(w + 1)$-bit result. Low w bits are stored in `S`, and high order 1-bit (the carry) is stored in `C`. The multiprecision addition can be carried out similarly.

```
#define WSIZE     (8*sizeof(word))
#define MSBMASK   ((word)1 << (WSIZE-1))
  S=(a & ~MSBMASK) + (b & ~MSBMASK);
  C=(a >> (WSIZE-1)) + (b >> (WSIZE-1)) + (S >> (WSIZE-1));
  S = a + b;
  C >>= 1;
```

A multiplication expression in C stores only the low order word of the two-word product. Let the operation be $c := a \cdot b$ where a and b are `word` type variables. The type of the result is also `word`. In order to obtain the complete product, the w-bit input operands are split into two $w/2$-bit numbers. The following C code segment can be used to obtain the full result of c in the word pair `(C,S)`.

```
#define WSIZE           (8*sizeof(word))
#define LOWBITS(x)      ((x) & (~((word)0) >> (WSIZE/2)))
#define HIGHBITS(x)     ((x) >> (WSIZE/2))
  albl = LOWBITS (a) * LOWBITS (b);
  ahbl = HIGHBITS(a) * LOWBITS (b);
  albh = LOWBITS (a) * HIGHBITS(b);
  ahbh = HIGHBITS(a) * HIGHBITS(b);
  sum = LOWBITS(albh) + LOWBITS(ahbl) + HIGHBITS(albl);
  S = (sum << (WSIZE/2)) + LOWBITS(albl);
  C = ahbh + HIGHBITS(albh) + HIGHBITS(ahbl) + HIGHBITS(sum);
```

2.2 C with Extended Types

This approach relies on a non-standard type of the C programming language. The code is still portable, maintainable, and testable, however, it is restricted to the platform on which the the non-standard language extensions are supported. This method depends on the fact that a variable of twice the size of a general purpose register contains all result bits for the operation. Let the name of this extended type be `dword`. We can then implement the addition and multiplication of two words as follows:

```
#define WSIZE     (8*sizeof(word))
  CS = (dword)a + (dword)b;
  S = (word)CS;
  C = (word)(CS >> WSIZE);
```

```
#define WSIZE     (8*sizeof(word))
  CS = (dword)a * (dword)b;
  S = (word)CS;
  C = (word)(CS >> WSIZE);
```

Here `C` and `S` are `word` type variables, and `CS` is a `dword` type variable. The extended type can be used for other operations, e.g., shift and division. The C compiler must convert the right shifting by *wordsize* bits to a single register access to achieve better performance.

This approach does not require the assembler level implementation. However, the compiler must support the extended type, and also it should be capable of generating efficient code for C blocks involving the extended type. Currently, most C compilers support double register-size variables. For example, Microsoft Visual C++ has "`__int64`" while most of UNIX C compilers have "`long long`" type for variables of twice the register-size of the platform processor.

2.3 Complete Assembler

An efficient assembler implementation of a cryptographic algorithm requires a detailed study of the architecture of the underlying processor. Issues related to the instruction set architecture, register space, multiple functional units, and memory hierarchy need to be well understood. An assembler implementation produces smaller and faster code by sacrificing portability. However, assembler implementations are preferred if development costs are relatively less than final benefits.

The Appendix gives assembler programming example codes in Intel Pentium and Sparc V9 assembly languages for performing several different operations with 32-bit numbers. For example, the operations `ADD(C,S,a,b)` and `MUL(C,S,a,b)` respectively denote $(C, S) := a + b$ and $(C, S) := a \cdot b$, where C, S, a, and b are 32-bit unsigned integers.

2.4 C with Kernel in Assembler

The speedup obtained with the extended types is not as high as possible due to inefficient utilization of the processor architecture. The performance is limited by the optimization capabilities of the C compiler. We propose an alternative hybrid approach which benefits from flexibility of C and high performance of assembly languages. We minimize the development cost of assembly language programming by proposing a small set of arithmetic operations which need to be coded in the assembler. The remainder of the code is produced in the standard C. The proposed set of kernel operations is given below:

Table 1. The proposed kernel operations.

Operation	Description
`ADD(C,S,a,b)`	$(C, S) := a + b$
`ADD2(C,S,a,b,c)`	$(C, S) := a + b + c$
`MUL(C,S,a,b)`	$(C, S) := a \cdot b$
`MULADD(C,S,a,b,c)`	$(C, S) := a \cdot b + c$
`MULADD2(C,S,a,b,c,d)`	$(C, S) := a \cdot b + c + d$
`MUL2ADD2(CC,C,S,a,b,c,d)`	$(CC, C, S) := 2 \cdot a \cdot b + c$
`SQU(C,S,a)`	$(C, S) := a^2$
`SQUADD(C,S,a,b)`	$(C, S) := a^2 + b$

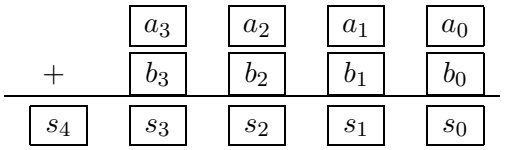
These operations need to be coded in the assembly language as macros or in-line assembly code segments. They can be written as functions, but this creates considerable overhead. The best situation will be the one in which these operations are supported by the hardware either as instructions or macro instructions.

The amount of assembly language is indeed minimal: each one of these operations can be coded using about 4 to 8 lines of assembler instructions. Therefore, the entire set requires about 60 lines of assembly code. The resulting standard C plus assembler code, if carefully constructed, can be ported to another machine quite easily: only the assembly code segments need to be developed for the new machine, replacing the existing segments.

2.5 Determination and Utilization of Kernel Operations

The arithmetic operations in the kernel are obtained by analyzing the algorithms and implementations of the number-theoretic cryptosystems. The proposed kernel is quite minimal in the sense adding other similar or more complicated operations does not provide any considerable speedup gain. Since our objective is to write as little assembly code as possible, we carefully selected these operations among several candidates. These experiments were run on the Intel 486 DX4, Intel Pentium, and Sun UltraSparc-II V8+ machines by examining the algorithms and source codes for the RSA, Diffie-Hellman, and DSA algorithms. These algorithms require modular arithmetic with large integers. A typical time-consuming operation is the multiprecision modular exponentiation which is computed using the Montgomery multiplication and squaring operations [9, 12, 10]. Furthermore,

the RSA private key operation uses the Chinese remainder in order to speedup the computation [14]. Therefore, typical multiprecision arithmetic operations used in these number-theoretic cryptosystems are multiprecision addition, multiplication, modular and Montgomery multiplication and squaring operations. As an example, the addition and multiplication of two 4-word integers are illustrated below:



$$\begin{aligned}(c_0, s_0) &= a_0 + b_0\\ (c_1, s_1) &= a_1 + b_1 + c_0\\ (c_2, s_2) &= a_2 + b_2 + c_1\\ (s_4, s_3) &= a_3 + b_3 + c_2\end{aligned}$$

A word of the 5-word sum $(s_4, s_3, s_2, s_1, s_0)$ can be computed using the `ADD2` kernel operation, as illustrated in the pseudocode given below. Here, the variables a, b and c are arrays of type `word`, k is the number of words, and `C` and `S` are `word` variables.

```
C = 0
for i=0 to k-1 do
  ADD2 (C, S, a[i], b[i], C)
  s[i] = S
s[k] = C
```

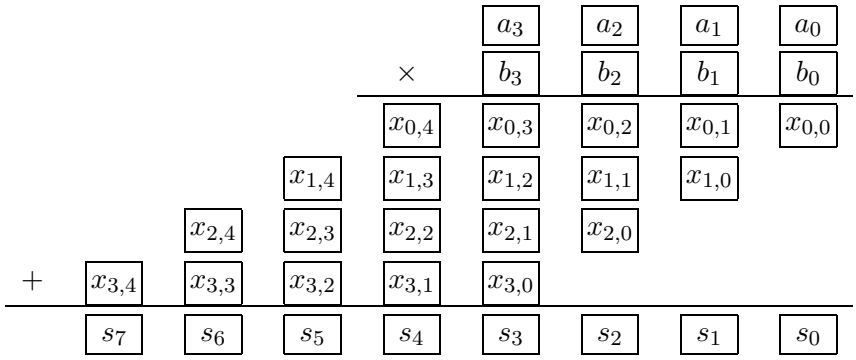
In order to multiply two 4-word unsigned integers, we accumulate the partial products in the 8-word array s as depicted below. The accumulation requires the addition of the previous product word s_i to the partial product word $x_{i,j}$. The words of these these partial products are computed by

$$(c_{i,j}, x_{i,j}) = a_j \cdot b_i + c_{i,j-1} \ ,$$

where $c_{i,-1} = 0$ and $i, j = 0, 1, \ldots, k-1$. This operation can be accomplished using the `MULADD` primitive. In order to add the computed product, the previous carry, and the accumulated result, we need to perform the operation

$$(c_{i,j}, s_{i+j}) = a_j \cdot b_i + c_{i,j-1} + s_{i+j} \ ,$$

which can be accomplished using `MULADD2` primitive.



The pseudocode given below computes the product $a \cdot b$ in the k-word array s employing the `MULADD2` kernel operation. Other primitive operations, e.g., `MUL2ADD2` and `SQUADD`, are used in squaring when $a = b$.

```
for i=0 to 2*k-1 do   s[i] = 0
for i=0 to k-1 do
  C = 0
  for j=0 to k-1 do
    MULADD2 (C, S, a[i], b[j], s[i+j], C)
    s[i+j] = S
  s[i+k] = C
```

The plain C versions of the given code fragments can be obtained by replacing the addition and multiplication statement sequences given in the previous sections. The use of these kernel operations drastically reduce the code size. Moreover, the performance gained by the kernel operations exceeds the benefits of the extended types in both speed and code size.

3 Implementation Results

The proposed kernel of operations were implemented in the assembly languages of the Intel 486, Intel Pentium, and Sparc V9 machines. The Intel 486 DX4 processor has the speed of 100 MHz, and runs the NextStep operating system v3.0. We used the C compiler of the NextStep. On the other hand, the Microsoft Visual C++ v4.2 and and Intel VTune v2.0 are used in the development and analysis for the Intel Pentium processor on a Windows NT 3.51 system. The SPARCompiler SC4.0 is used for the Sparc V9 processor on a UltraSparc-II V8+ system. The C compiler was configured to obtain the speed-optimized code.

We implemented the 512-bit and 1024-bit bit modular exponentiation operations which are common in the RSA and Diffie-Hellman algorithms. The exponentiation algorithm is the binary method [9] using the Montgomery multiplication [12, 10]. The size of the modulus is 512 bits and 1024 bits. The exponent was selected as 1-word (32-bit) and full-size (the size of the modulus). Table 2 tabulates the timings in milliseconds for these operations.

Table 2. Modular exponentiation timings in milliseconds.

		32-bit exponent				full exponent			
Processor and OS	Modulus Size	C	C with E. Types	C with Kernel	Asm.	C	C with E. Types	C with Kernel	Asm.
i486DX4 100 MHz	512	29	28	26	10	488	405	363	205
NextStep v3.0	1024	110	103	95	39	3,775	3,195	2,800	1,559
UltraSparc-II V8+	512	8	5	6	3	150	103	106	55
Solaris 5.5.1	1024	31	21	23	12	1,144	790	795	414
Pentium 120 MHz	512	15	11	8	5	206	151	91	59
NT v3.51	1024	57	43	28	18	1,618	1,166	694	446

The fastest implementation is obtained using assembly language programming. For example, the assembler implementation of full-size modular exponentiation with 1024 bits is about 3.63 times faster than the standard C implementation on the Pentium machine. The standard C coupled with kernel operations produces a code which is 2.33 times faster than the standard C code, which is about 64 % of the speed increase gained by the assembler implementation. Table 3 illustrates the speedup of the other three approaches to the standard C implementation for performing modular exponentiation where the exponent is the full size (i.e., it is equal to the modulus size).

Table 3. Speedup with respect to the standard C implementation.

Processor and OS	Modulus Size	C with E. Types	C with Kernel	Asm	Kernel vs Asm
i486DX4	512	1.20	1.34	2.38	56 %
NextStep v3.0	1024	1.18	1.34	2.42	55 %
UltraSparc-II V8+	512	1.46	1.42	2.73	52 %
Solaris v5.5.1	1024	1.45	1.45	2.76	53 %
Pentium	512	1.36	2.26	3.49	65 %
NT v3.51	1024	1.39	2.33	3.63	64 %

On the UNIX machines (NextStep and Solaris), we implemented the kernel operations using *functions* since in-line assembly coding is not flexible due to inability to access the C variables within the inline assembly code. In this case, the speed increase gained by the use of kernel operations is given away due to the overhead of function calling. For example, the C with kernel operations case is only slightly slower (or faster) than the C with extended types case for the Sparc machine running Solaris (or the Next machine).

4 Conclusions

We have proposed a design methodology and a small set of kernel operations for obtaining high-speed implementations of the number-theoretic cryptographic algorithms. It is shown that about 64 % of speed increase gained by the use of full assembler implementation can be obtained by coding only the proposed set of kernel operations in the assembly language of the underlying processor. It is preferred that the development system provide in-line assembly coding in order to avoid the overhead of function calling in implementing the kernel operations.

This approach allows the programmer to drastically reduce assembly language programming while gaining a significant speedup. Since the assembler portion is quite minimal (a total of 60 lines at most), the maintainability and testability of the code are retained. The code can easily be ported to a different platform (processor) by implementing only the suggested set of kernel operations. Furthermore, the kernel operations proposed in this paper are easy to implement in hardware. If they are available as instructions (or macros) on microprocessors or signal processors, high-speed implementations of number-theoretic cryptographic algorithms can easily be obtained.

References

[1] M. Atkins and R. Subramaniam. PC software performance tuning. *IEEE Computer Magazine*, 29(8):47–54, August 1996.

[2] T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, May 1989.

[3] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[5] S. R. Dussé and B. S. Kaliski, Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. New York, NY: Springer-Verlag, 1990.

[6] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[7] B. S. Kaliski, Jr. The Z80180 and big-number arithmetic. *Dr. Dobb's Journal*, pages 50–58, September 1993.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[9] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[10] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[11] D. Laurichesse and L. Blain. Optimized implementation of RSA cryptosystem. *Computers & Security*, 10(3):263–267, May 1991.

[12] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[13] National Institute for Standards and Technology. Digital signature standard (DSS). Federal Register, 56:169, August 1991.

[14] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

[15] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

A Implementation of Kernel Operations

A.1 Pentium

ADD(C,S,a,b)

```
mov    eax,dword ptr [a]
mov    ebx,dword ptr [b]
mov    dword ptr [C],0
add    eax,ebx
mov    dword ptr [S],eax
setc   byte ptr [C]
```

ADD2(C,S,a,b,c)

```
mov    eax,dword ptr [a]
mov    ebx,dword ptr [b]
xor    edx,edx
mov    ecx,dword ptr [c]
mov    dword ptr [C],edx
add    eax,ebx
setc   byte ptr [C]
add    eax,ecx
mov    dword ptr [S],eax
adc    dword ptr [C],0
```

MUL(C,S,a,b)

```
mov    eax,dword ptr [a]
mul    dword ptr [b]
mov    dword ptr [C],edx
mov    dword ptr [S],eax
```

MULADD(C,S,a,b,c)

```
mov    eax,dword ptr [a]
mov    ebx,dword ptr [c]
mul    dword ptr [b]
add    eax,ebx
adc    edx,0
mov    dword ptr [S],eax
mov    dword ptr [C],edx
```

MULADD2(C,S,a,b,c,d)

```
mov    eax,dword ptr [a]
mov    ebx,dword ptr [c]
mul    dword ptr [b]
add    eax,ebx
mov    ebx,dword ptr [d]
adc    edx,0
add    eax,ebx
adc    edx,0
mov    dword ptr [S],eax
mov    dword ptr [C],edx
```

MUL2ADD2(CC,C,S,a,b,c)

```
mov    eax,dword ptr [a]
mul    dword ptr [b]
add    eax,eax
mov    dword ptr [CC],0
adc    edx,edx
mov    ebx,dword ptr [c]
adc    byte ptr [CC],0
add    eax,ebx
adc    edx,0
mov    dword ptr [S],eax
mov    dword ptr [C],edx
adc    byte ptr [CC],0
```

SQU(C,S,a)

```
mov    eax,dword ptr [a]
mul    eax
mov    dword ptr [S],eax
mov    dword ptr [C],edx
```

SQUADD(C,S,a,b)

```
mov    eax,dword ptr [a]
mov    ebx,dword ptr [b]
mul    eax
add    eax,ebx
adc    edx,0
mov    dword ptr [S],eax
mov    dword ptr [C],edx
```

A.2 Sparc V9

ADD(C,S,a,b)

```
clruw   %o2
clruw   %o3
add     %o2,%o3,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

ADD2(C,S,a,b,c)

```
clruw   %o2
clruw   %o3
clruw   %o4
add     %o2,%o3,%g1
add     %g1,%o4,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

MUL(C,S,a,b)

```
clruw   %o2
clruw   %o3
mulx    %o2,%o3,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

MULADD(C,S,a,b,c)

```
clruw   %o2
clruw   %o3
clruw   %o4
mulx    %o3,%o2,%g1
add     %g1,%o4,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

MULADD2(C,S,a,b,c,d)

```
clruw   %o2
clruw   %o3
clruw   %o4
clruw   %o5
mulx    %o2,%o3,%g1
add     %o4,%o5,%g2
add     %g1,%g2,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

MUL2ADD2(CC,C,S,a,b,c)

```
clruw   %o3
clruw   %o4
clruw   %o5
mulx    %o4,%o3,%g1
mov     %g0,%g2
addcc   %g1,%g1,%g1
movcs   %xcc,1,%g2
addcc   %g1,%o5,%g1
movcs   %xcc,1,%g2
stuw    %g1,[%o2]
srlx    %g1,32,%g1
stuw    %g2,[%o0]
retl
stuw    %g1,[%o1]
```

SQU(C,S,a)

```
clruw   %o2
mulx    %o2,%o2,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

SQUADD(C,S,a,b)

```
clruw   %o2
clruw   %o3
mulx    %o2,%o2,%g1
add     %g1,%o3,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```