Bachelor thesis
Computer Science

Radboud University

Parallel SHA-256 in NEON for use in hash-based signatures

Author:
Wouter van der Linde
`wlinde@science.ru.nl`

First supervisor/assessor:
dr., Peter Schwabe
`peter@cryptojedi.org`

Second assessor:
dr., Lejla Batina
`lejla@cs.ru.nl`

July 13, 2016

Abstract

In this thesis we present an SIMD implementation of the SHA-256 algorithm, optimised for the ARM Cortex-A8 processor, using the NEON vector instruction set. This implementation is specifically designed for use in hash-based signatures. By implementing SHA-256 directly in assembly, we achieve a performance gain of 33.5% for 256-bit input and 27.5% for 512-bit input messages.

Contents

Chapter 1

Introduction

In 1994, Shor showed that both the discrete logarithm problem and integer factorization can be solved in polynomial time by a quantum computer [20]. As most (if not all) of the widely used asymmetric cryptographic systems today depend on the hardness of those problems, these systems will be broken when we manage to actually build a quantum computer. We therefore need alternatives to classical systems like RSA, DSA and ECDSA. In the case of digital signatures, one such alternative is hash-based signatures. These have been introduced by Lamport in 1979 [13], as a one-time signature scheme. In 1989 Merkle proposed his Merkle signature scheme [16], which extended Lamport's scheme with a tree-hash algorithm enabling the use of multiple signatures with one public key, and which has been the basis for modern schemes like GMSS [6] and XMSS [5]. Recent developments show that hash-based signature schemes are becoming more and more practical [12] [7], and schemes based on MSS have been proven to be secure in a post-quantum world [22]. However, more work needs to be done before they are ready to become the new standard for digital signatures.

This thesis aims to provide a stepping stone to efficient implementations of these hash-based signature schemes on more restricted devices by providing an efficient implementation of the hash function SHA-256 on the ARMv7-A architecture, specifically the ARM Cortex A8. This implementation will exploit the NEON vector instruction set available on most processors of this family. ARMv7-A processors, and their successors from the ARMv8-A architecture, are widely used in mobile devices such as smartphones and tablets by companies like Apple, HTC and Samsung. The majority of those devices have access to NEON instructions. Bernstein and Schwabe have shown that cryptographic primitives can benefit greatly from the NEON instruction set [3], and Sánchez and Rodrígues-Henríquez show similar results by applying these instructions to an attribute-based encryption scheme [19].

Hash-based signature schemes calculate a large amount of hashes of messages of a fixed input-size. Many of these calculations can be done in parallel, which can greatly benefit the efficiency of the signature scheme. Our contribution is a NEON vector implementation of SHA-256, specifically for ARM Cortex A8 processors, which calculates four hashes in parallel. It can be used to improve the efficiency of implementations of hash-based signature schemes like XMSS.

Chapter 2

Preliminaries

2.1 Hash-Based Signature Schemes

This section provides a brief overview of how hash-based signature schemes work. Hash-based signatures were introduced by Lamport in 1979, so we begin by taking a look at the Lamport signature scheme [13], which is a one-time scheme. We fix the hash function H as SHA-256.

This section is loosely based on a blog post by Langley from July 18th 2013 [14].

2.1.1 Lamport signature scheme

To create a signature using the Lamport signature scheme, we first need to generate a public and private key pair. We do this by generating 256 pairs of 256-bit random numbers (so 512 numbers), where pair i consists of x_i and y_i. These 256 pairs comprise our private key. Our public key now consists of the (256-bit) hashes of these 512 numbers, which we publish. To sign a message m, we calculate $H(m)$, and for each bit of $H(m)$ we publish either x_i or y_i; we choose x_i if bit i is 0, or y_i if bit i is 1. This results in a sequence of 256 numbers, which is our signature. To verify this signature, the verifier hashes m, and chooses 256 hashes of our public key the same way we chose the secret numbers when signing the message; if bit i is 0 he chooses $H(x_i)$, if it is 1 he chooses $H(y_i)$. The verifier then checks whether the hashes of the numbers in our signature match those in our public key; if they do, the signature is correct.

There are some problems with this scheme. First off, the public and private keys are 16 KB big, which is a problem because a public key can only be used once; if the same public key is used twice, an attacker can forge a signature because for some of the private number pairs both numbers have been used. We can use a PRG to generate the secret number pairs, which reduces the private key to a single seed. To solve the problems with the public key size and one-time signatures, we can use a Merkle hash tree [16].

2.1.2 Merkle signature scheme (MSS)

A Merkle hash tree is a perfect binary tree with N leaves. To construct a Merkle hash tree, we first generate N public/private key pairs of a one-time signature scheme, like Lamport's. The leaves of the tree are hashes of the generated public keys (being the hashes of the 512 secret numbers in case Lamport's scheme is used). Every other node in the tree consists of the hash of the concatenation of its children's values. In the example of Figure 2.1, the node with children A and B consists of $H(AB)$. The value of the root of the tree is the new public key.

To sign a message, you choose one of the leaves of the tree, and use the corresponding public/private keys to sign that message using the one-time scheme. To enable someone to verify your signature, you will have to convince him or her that the public key you used is part of your overall public key, or the root of the Merkle tree. Therefore, aside from the entire public key you used, you will have to give the path up the tree. In the example of Figure 2.1; if you used the key pair from leaf A, you will have to give that public key, and nodes B and $H(CD)$. The verifier can then calculate $H(AB)$ and $H(H(AB)H(CD))$, and compare the resulting root of the tree with your overall public key.

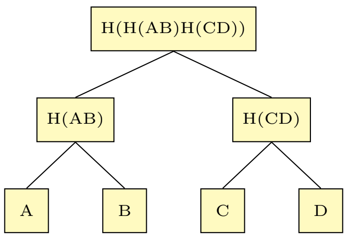

Figure 2.1: Example of a Merkle Hash Tree

This construction eliminates the big public key size, and enables you to use one public key for many signatures, as you can make the tree arbitrarily large. Because we use a stream cipher to determine the one-time private keys, the whole tree is fixed when we determine a seed. However, the entire hash tree is going to be enormous if we want to be able to sign a lot of messages with one public key. We can choose public keys at random in the hope of never colliding, but to avoid collisions the tree needs to be huge; in the order of 2^{128} entries. This means calculating the entire tree would take an immense amount of time. Therefore, we will need to defer calculating parts of the tree until we need them. We can do this by not creating one big tree, but a top-level tree with sub-trees, which have sub-trees of their own etc. Instead of signing a message with the key-pair of a leaf, we sign the root of a sub-tree, and we use the lowest level of trees to actually sign our messages. Again, because we use a stream cipher to generate the private keys, the private number pairs of all leaves are fixed. We can therefore calculate sub-trees when we need them. We still need to remember which leaves we have already used though.

There is one last problem with this construction, which is the size of resulting signatures. Since a signature has to contain the entire public key used, which is already 16 KB big, plus the path up the tree to the root node, its size is very unpractical when used to sign emails or other relatively small documents. In his original paper, Merkle describes a method to shorten the signature size called the Winternitz scheme [16], which we describe in the next section. However, even when using this scheme, signatures remain far bigger than those generated by signature schemes that are in use today.

2.1.3 Winternitz one-time signature scheme (W-OTS)

The Winternitz scheme effectively trades signature size for computation time by iterating a hash function. The size of the private and public keys, as well as the length of the function/hash chains (the amount of iterations), are dependent on the Winternitz parameter w, which is usually a small power of two; $w = 2^t$. The private key consists of a list of random numbers $x_1 \dots x_N$, and the public key Y is calculated as

$$Y = f(f^{w-1}(x_1))||\dots||f^{w-1}(x_N)),$$

where f^i denotes i iterations of a hash function (the output of one iteration being the input for the next).

To calculate a signature for a message m, we first hash this message (using a 256-bit hash), and then break this hash up into t-bit blocks. For example, if $w = 16 = 2^4$, we get 64 blocks of 4 bits, which we denote with $b_1 \dots b_{64}$. We also compute a checksum C (which is necessary to stop an attacker from forging signatures, more details can be found in [8]), which we break up into parts of t bits as well to get N blocks in total: $b_1 \dots b_N$. We then calculate the signature s by using the integer values of these blocks as the amount of iterations of the hash function on the values of the secret key;

$$s_1||\dots||s_N = f^{b_1}(x_1)||\dots||f^{b_N}(x_N)$$

so, for example, if $b_1 = 1001$ in binary, we iterate the hash function nine times on x_1; $f^9(x_1)$.

To verify this signature, we first calculate the parts $b_1 \dots b_N$ as described above, then iterate the hash function on the parts of the signature until we reach w iterations, and then hash the concatenation of these results one last time. We accept the signature if and only if the result is equal to the public key:

$$f(f^{w-b_1-1}(s_1)||\dots||f^{w-b_N-1}(s_N)) = Y$$

By using a Winternitz parameter $w = 16$, we can sign four bits of our (hashed) message with each part of our private key, as opposed to one bit per part when using Lamport's scheme. This significantly reduces the size of the generated signatures, but at the cost of some computation time since we need to calculate more hashes.

2.1.4 Improvements on MSS

An extension and improvement to MSS is GMSS [6], which is a highly flexible scheme that can be adjusted to meet requirements and constraints of the environment it is used in while having great signing time, signature capacity and reasonable signature sizes. More recent work is XMSS [5], an efficient and forward-secure signature scheme with minimal security requirements (being a secure and efficient second-preimage resistant function and a pseudo-random function), and XMSS+ [12], which expands on XMSS to generate keys in $\mathcal{O}(\sqrt{n})$ instead of $\mathcal{O}(n)$, which makes XMSS+ practical for use on smart-cards. The authors therefore provide the first full implementation of a hash-based signature scheme on a smart-card. These schemes are based on the Merkle signature scheme described above, and their performance is comparable to RSA and ECDSA. XMSS and XMSS+ use the Winternitz scheme described above [4], which is improved in [11] to provide a bigger reduction in signature size while reaching a higher level of security. These developments suggest that hash-based signature schemes are now becoming practical, and might in the near future become the standard for digital-signature algorithms.

2.1.5 Fixed-size input and parallelization

As can be seen in the schemes described above, most of the hashes used in hash-based signature schemes are performed on either 256 or 512 bits input messages. Using Lamport's scheme, all hashes (except when hashing the message to be signed) are performed on 256-bit private key values, and when using W-OTS, we calculate hashes on either 256-bit private key values or 256-bit output of a previous hash iteration (again, except when hashing the message to be signed). When using Merkle trees, we hash 256-bit public keys in leaves or, in other nodes, the concatenation of two 256-bit child nodes. Because of this, we restrict our implementation to fixed-size inputs of either 256-bit or 512-bit input messages. This allows us to get rid of some unnecessary overhead, and thus boost the performance of our implementation.

Also, since most of the hashes calculated in hash-based signature schemes do not depend on output of others, many can be calculated in parallel. A vectorized implementation of hash functions can therefore significantly improve the performance of these signature schemes.

2.2 SHA256 Algorithm

SHA-256 is part of the SHA-2 hash function family, which was designed by the NSA and first published in 2002 [17]. It produces 256-bit hashes (or 'message digests') for messages of at most $2^{64}-1$ bits long. This section provides a brief overview of the structure of the SHA-256 algorithm. For more detailed information on SHA-256 and the other SHA-2 hash algorithms, we refer to the official SHA-2 documentation [18]. All functions and constants described in this section can be found there.

The SHA-256 algorithm consists of two stages, preprocessing and hash computation. It uses eight working variables of 32 bits each (which we will denote with a to h), a message schedule of sixty-four 32-bit words (denoted with w_0 to w_{63}) and a hash value of eight 32-bit words (H_0 to H_7).

2.2.1 Preprocessing

During the preprocessing stage the hash value is initialized using constants which can be found in section 5.3.3 of [18], and the message M is padded to a multiple of 512 bits. This padding is done as follows: first the bit 1 is appended to the message, followed by 0's until the length of the padded message modulo 512 is equal to 448 bits. Finally the length of the message is appended using a 64-bit block.

$$\text{Padded message} : M||1||0||...||0||len(M)$$

2.2.2 Hash computation

The hash computation stage consists of the looping of four steps over all 512-bit blocks of the padded message. During each loop i, the i^{th} 64-byte block of the message M is used to calculate an intermediate hash value. In this section (and the rest of this thesis) we use $\oplus$ to denote XOR operations, *SHR* for shift-right and *ROTR* for rotate-right.

First the message schedule, w_0 to w_{63}, is prepared as follows, where M_t^i denotes 32-bit word t of block i of the input message:

$$w_t = \begin{cases} M_t^i & \text{for } 0 \leq t \leq 15, \text{ and} \\ \sigma_1(w_{t-2}) + w_{t-7} + \sigma_0(w_{t-15}) + w_{t-16} & \text{for } 16 \leq t \leq 63 \end{cases} \tag{2.1}$$

where

$$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x), \text{ and}$$

$$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x).$$

We will refer to the message scheduling calculation (and specifically to the second line in the above equation) as the function M.

Next, the eight working variables a to h are initialized with the $(i-1)^{st}$ hash values (when i is 0, these are constants defined in [18]):

$$\begin{aligned} a &= H_0^{i-1} & e &= H_4^{i-1} \\ b &= H_1^{i-1} & f &= H_5^{i-1} \\ c &= H_2^{i-1} & h &= H_6^{i-1} \\ d &= H_3^{i-1} & g &= H_7^{i-1} \end{aligned}$$

During the third step some calculations are performed on the working variables a to h, using the temporary variables T_1 and T_2, and sixty-four 32-bit constants (denoted k_0 to k_{63}, see section 4.2.2 of [18] for their values):

```
**for** $t = 0$ to $63$ :
    $T_1 = \Sigma_1(e) + Ch(e, f, g) + k_t + w_t + h$
    $T_2 = \Sigma_0(a) + Maj(a, b, c)$
    $h = g$            $d = c$                    (2.2)
    $g = f$            $c = b$
    $f = e$            $b = a$
    $e = d + T_1$      $a = T_1 + T_2$
```

where

$$\begin{aligned} \Sigma_0(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x), \\ \Sigma_1(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x), \\ Ch(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z), \text{ and} \\ Maj(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z). \end{aligned} \tag{2.3}$$

We will refer to these calculations on the working variables with the function F.

Finally, the i^{th} intermediate hash value is calculated by adding the intermediate values to the current hash values:

$$\begin{aligned} H_0^i &= a + H_0^{i-1} & H_4^i &= e + H_4^{i-1} \\ H_1^i &= b + H_1^{i-1} & H_5^i &= f + H_5^{i-1} \\ H_2^i &= c + H_2^{i-1} & H_6^i &= g + H_6^{i-1} \\ H_3^i &= d + H_3^{i-1} & H_7^i &= h + H_7^{i-1} \end{aligned} \tag{2.4}$$

After repeating these four steps for all n blocks of an input message M, the resulting message digest is obtained by appending the last intermediate hash values:

$$H_0^n || H_1^n || H_2^n || H_3^n || H_4^n || H_5^n || H_6^n || H_7^n$$

2.3 ARM Cortex-A8 Processor

The Cortex-A8 processor implements the ARMv7-A architecture, which includes the 32-bit ARM instruction set and the NEON Advanced SIMD architecture. It also includes the Thumb-2 and ThumbEE instruction sets, as well as the Security Extensions architecture, but as these have not been used in our implementation we will not address them here. In this section we briefly discuss the relevant aspects of the Cortex-A8 for this thesis. For more detailed information, we refer to the Cortex-A8 Technical Reference Manual [15]. For a short programmer's guide, we recommend [23].

All operations on the Cortex-A8 are performed in little-endian mode by default. It is possible to switch to big endian and back by using the *setend* instruction, but this only affects memory accesses: arithmetic operations are still performed in little endian mode. We use this instruction occasionally in our implementation to give hashes consistent with standard SHA256 implementations.

We created and tested our implementation on a BeagleBone Black development board, which includes a 1 GHz ARM Cortex A8 processor. More detailed information can be found on the BeagleBone website: `http://beagleboard.org/BLACK`.

2.3.1 Architecture

Registers

The Cortex-A8 has 16 32-bit registers, r_0 through r_{15}. The first four of these are used as the input parameters of a program. The stack pointer is stored in r_{13}, and r_{15} holds the program counter, which leaves 14 general-purpose registers. The NEON architecture has 16 registers of 128 bits, denoted q_0 through q_{15}. These can also be used as 32 registers of 64 bits, denoted d_0 through d_{31}; q_0 is equivalent to d_0 and d_1 combined, q_1 is equivalent to d_2 and d_3, etc. As Bernstein and Schwabe state in [3], one of the most obvious benefits of using NEON is that you have much more space available in registers, being 2048 bits opposed to the 448 bits of the basic ARM architecture. This can reduce the amount of loads and stores needed.

Pipeline

The Cortex-A8 is a pipelined processor, which means that multiple instructions can be processed simultaneously; they can be issued directly after each other, as long as the results of the first instruction are not directly needed for the second. If they are, execution will stall until those results are available.

Dual issue

The NEON architecture includes separate pipelines for the arithmetic unit and the load/store unit. This enables the processor to dispatch two instructions during one cycle, which are then processed in parallel. The order of the two instructions is not important. Interleaving load/store and arithmetic operations can improve code efficiency significantly. This is illustrated best in an example from [3]: assuming no other latencies, the sequence '*A LS A LS A LS*' (where '*A*' stands for an arithmetic instruction and '*LS*' for a load or store) costs three cycles, because this sequence is issued as '*A LS, A LS, A LS*'. The sequence '*LS LS LS A A A*' costs five cycles ('*LS, LS, LS A, A, A*').

2.3.2 Cycle timing

Instructions on the Cortex-A8 go through several pipeline stages, denoted with N_0 (when the instruction is issued), N_1 etc, each of which takes one cycle. For example, a *vadd* instruction (addition on vectors) requires its inputs at pipeline stage N_2 (one cycle after it is issued), and produces its result in stage N_3. This result can then be used by another instruction from N_4 onwards. Note that the official ARM Cortex-A8 manual [15] always reports the stage just before the result is available as the output stage. A *vsub* instruction (vector subtraction) behaves a bit differently, as it requires its first input one stage earlier, in N_1. Aligned load/store instructions occupy the processor for one cycle, producing their result in stage N_2. Unaligned loads and stores require one extra cycle. When the result(s) of one instruction are used in another, the latter will have to wait for the former. By cleverly ordering instructions, it is often possible to hide these latencies (more on this in section 4.3.1).

To get an idea of the performance and latency problems of our implementation, we used an online Cortex-A8 cycle counter by Sobole [21]. This tool can be somewhat pessimistic at times, but displays most latencies correctly. Moreover, it shows how long an instruction is stalling and for which register it is waiting, which is a great help when optimizing assembly code.

Figure 2.2: Equivalent code

(a) NEON Assembly

```
vshr.u32 q2, q1, #15
veor     q0, q0, q2
vshr.u32 q5, q1, #19
veor     q4, q4, q5
```

(b) Qhasm equivalent

```
4x a = b unsigned>> 15
d ^= a
4x c = b unsigned>> 19
e ^= c
```

2.3.3 Cycle counter

The Cortex A8 has a cycle counter which can only be accessed from kernel mode. Bernstein published a tool that exposes the cycle count to a Linux device file which is included in the SUPERCOP benchmarking suite [2]. Unfortunately, at the time of writing SUPERCOP does not support benchmarking for parallelized hashes. Our results are therefor measured on the BeagleBone Black development board we used for our implementation, using the cycle counter tool by Bernstein.

2.4 Qhasm

Programming in assembly is a notoriously delicate and time-consuming task. The Qhasm tool developed by Bernstein provides a preprocessor which translates a C-like syntax directly to working assembly code; every line in Qhasm corresponds to exactly one line in assembly. Figure 2.2 shows an example of the Qhasm syntax; it displays a short assembly program in *(a)*, and its Qhasm equivalent in *(b)*. Another major feature of Qhasm, and perhaps its most useful, is that it takes care of register allocation. The programmer only needs to worry about not using more registers than are available, as Qhasm will not automatically spill variables to the stack.

From here on, all our code examples will use the Qhasm syntax, as it is easier to read than raw assembly.

We added some instructions to Qhasm in order to properly load and store values from and to memory in big endian mode. The version of Qhasm used for our implementation can be found at https://gitlab.science.ru.nl/wlinde/Qhasm.

Chapter 3

Related Work

In 2004 Açıızmez explored the possibilities of using SIMD architectures to improve the performance of several SHA hash functions (specifically SHA-1, SHA-256, SHA-384 and SHA-512) [1], concluding that the hashing of two or four independent messages can be made significantly faster using this strategy. In [10] Gueron and Krasnov expand on Açıızmez's study by demonstrating performance gains on hashing multiple independent messages of arbitrary size on (at the time) contemporary processors. In [9] Gueron describes a tree mode implementation of SHA-256, using the AVX architecture on a third generation Intel Core processor, to speed up the hashing of a single message. The goal of this thesis is different from those of the above three papers in that our implementation is specifically meant for use in hash-based signatures, and therefore focuses on multiple independent inputs of fixed-size length.

In [7], Oliveira and López present an efficient implementation of the hash-based signature scheme MSS, using the AVX2 instruction set on an Intel Haswell processor, using vectorized versions of SHA-384 and SHA-256 to speed up key generation, signing and verification. The aim of this thesis is to provide a similar SHA-256 implementation for ARM Cortex-A8 processors, to speed up hash-based signatures for more resource-restricted devices.

Chapter 4

Implementation

4.1 Lower-bound Performance Analysis

In order to properly evaluate the performance of our NEON Vector Implementation, we give a lower-bound performance analysis of the SHA-256 algorithm to determine an absolute minimum on the amount of cycles needed to calculate a single SHA-256 hash. Because all NEON arithmetic and shift instructions used in our implementation occupy the processor for only one cycle, it is sufficient to count how many of these instructions are required. Since we are aiming for an absolute lower bound, we do not take into consideration any latencies between instructions. With proper instruction scheduling, it should be possible to eliminate most (if not all) latencies anyway. We also do not count any cycles for loading or storing data to or from memory: dual issuing enables us to load and store data for free, as we explained in Section 2.3.1, when pairing memory access with an arithmetic or shift instruction. Since we use NEON vector instructions in this analysis, the calculated lower bound applies to the simultaneous hashing of four input messages.

As explained in Section 2.2.2, when preparing the message schedule, the functions σ_0 and σ_1 are used. Both of these use two XORs, two rotate-right functions and one shift-right instruction. The rotate-right functions consist of a shift-left, a shift-right and an XOR instruction (for example, $ROTR^7(x)$ is equivalent to the sequence $SHL^{25}(x) \oplus SHR^7(x)$ for 32-bit instructions)[1]. Therefore, the functions σ_0 and σ_1 both use (at least) nine cycles. Equation 2.1 also uses three additions, resulting in 21 cycles, for 48 entries of the message schedule array. This gives us a total of $48 \cdot 21 = 1008$ cycles for the function M.

Since the second step of hash computation consists only of storing values, no cycles are added here.

[1] A rotate-right instruction can also be coded using a shift-left and shift-right-insert instruction, but this solution generally introduces more latency, as we show in Section 4.3.1.

During the third step, 64 iterations of calculations are performed on the working variables $a, b, \ldots, h$. The functions Σ_0 and Σ_1 from Equation 2.3 both use eleven cycles (three $ROTR$ of three cycles each and two additions), the function Ch uses three[2], the function Maj uses five, and we have seven more additions, giving a total of 37 cycles each iteration. Therefore, the function F uses at least $64 \cdot 37 = 2368$ cycles.

Lastly, we have eight more additions when calculating the intermediate hash value in Equation 2.4, which gives a lower bound of $1008 + 2368 + 8 = 3384$ cycles per 64-byte block of the padded input message. Since 256-bit messages are padded to one 512-bit block, the minimum amount of cycles needed to hash a 256-bit message is 3384. 512-Bit messages are padded to two 512-bit blocks, resulting in a minimum of 6768 cycles.

4.2 Code Structure

Our implementation is based on Bernstein's SHA-256 implementation included in the SUPERCOP benchmarking suite [2]. This implementation follows the structure outlined in Section 2.2, but differs in that it only calculates parts of the message schedule (w_0 through w_{63}) when they are needed. An overview of our code structure can be found in Appendix A. We adapted this C implementation to hash a vector of four inputs, and then translated it to assembly using the NEON SIMD instruction set.

4.3 Optimisations

Next we describe how we optimized our implementation in order to gain a significant speed-up compared to the reference implementation. We start by looking at minimizing latencies between instructions, which can be done by rearranging instructions that depend on each other's output and by interleaving functions. We also look at substituting the shift-right-insert instruction with a more efficient option. We then discuss how we eliminated some load and store instructions, and briefly look at utilizing big-endian mode and how it can affect performance.

4.3.1 Minimizing latencies

Rearranging instructions

As we briefly mentioned in Section 2.3.2, when the result of one instruction is used in a subsequent one, the latter has to wait for the result of the first to be available. Generally, NEON arithmetic instructions produce their result two cycles (pipeline stage N_3) after they were issued, so a subsequent instruction

[2]Here we use the BIC (bit-clear) instruction, which uses one cycle.

Figure 4.1: Equivalent code, but different performances

(a) Much latency: 18 cycles

```
1   z aligned= mem128[e]; e += 16
2   4x Sigma = z << 26
3   4x Sigma insert= z >> 16
4
5   4x r = z << 21
6   4x r insert= z >> 11
7   Sigma ^= r
8
9   y aligned= mem128[e]; e += 16
10  4x ChMaj = z & y
11
12  y aligned= mem128[e]; e += 16
13  4x r = y & ~z
14  ChMaj ^= r
```

(b) Less latency: 12 cycles

```
1   z aligned= mem128[e]; e += 16
2   4x Sigma = z << 26
3
4   y aligned= mem128[e]; e += 16
5   4x ChMaj = z & y
6   4x r = z << 21
7   4x Sigma insert= z >> 6
8   4x r insert= z >> 11
9
10  y aligned= mem128[e]; e += 16
11  4x v = y & ~z
12
13  Sigma ^= r
14  ChMaj ^= v
```

can use this result one cycle after that (stage N_4 of the first instruction). If the input for an instruction is not yet available, the processor stalls until it is. The stage in which the inputs need to be available differs per instruction. *vadd*, *vand* (logical 'AND'), *vorr* (logical 'OR'), *veor* (logical 'XOR') and *vbic* (logical 'AND NOT') all require both their inputs to be available in stage N_2 (when operating on 128-bit registers). A *vsub* instruction is a bit different in that it requires its second input to be available at stage N_1. All shift instructions require their input at stage N_1. To avoid the processor from stalling, and thereby wasting precious cycles, instructions have to be ordered in a way that minimizes the amount of latency between them.

Figure 4.1 shows how this can be done. Lines with stalling instructions have been colored red. Program (a) will for example stall at line number three; the *vsri* instruction requires *Sigma* to be available, but because of the previous shift-left instruction, it will not be available for two more cycles. The *vsri* instruction on line six stalls for the same reason. The *veor*'s on lines seven and nine stall for one cycle instead of two, since they require r at stage N_2 instead of N_1. These four latencies are resolved in (b) by interleaving lines nine to fourteen with lines one to seven, and renaming register r on lines 13 and 14 to v to avoid conflicts with the value we already assigned to r. At the cost of one extra register, we reduced the cycle count by one third.

Shift-right/XOR instead of shift-right-insert

In the above example we use the *vsri* instruction to shift a value to the right and insert the result of that shift into a destination vector, using only one instruction. This instruction occupies the processor for two cycles, which means that during the second cycle of this instruction no other instructions can be issued. In our vectorized SHA-256 implementation we have replaced this instruction with a separate shift right and XOR, which gives the same result but eliminates the unused cycle. Even though we still use two cycles for the same result, we can now issue a memory-access instruction during the second cycle as well. In general, as Bernstein and Schwabe mentioned in [3] as well, *vsri* instructions cause more latency problems than a separate shift and XOR.

4.3.2 Eliminating loads and stores

Even though aligned-load-and-store instructions are free to execute when dual issued with an arithmetic instruction, they are not entirely free. It takes about 20 cycles for stored data to be available for the ARM unit of the processor. Keeping values in registers instead of storing and loading them again later is still beneficial for performance. This is where NEON's large register space is especially helpful.

4.3.3 Bigendian mode

In order to be compliant with existing implementations of SHA-256, the C-implementation by Bernstein uses two functions, being *load_bigendian()* and *store_bigendian()*, to make sure that certain values are loaded from and stored to memory in big endian mode. Bernstein's code uses two blocks of function calls to *load_bigendian()*, one to load the initialization constants and one to load the input message, and one block of *store_bigendian()* to store the final hash to memory. These values are loaded and stored (respectively) in big endian format, despite the endianness of the host system. These three blocks cost a few hundred cycles each, and are therefore rather expensive. By using the *setend* instruction available in the ARMv7-A architecture, we were able to eliminate the block of *store_bigendian()* calls and the second block of *load_bigendian()* calls[3].

As explained in Section 2.3, the *setend* instruction can be used to switch the endianness of the Cortex-A8, but this only affects memory access instructions. In Qhasm, the keyword *bigendian* is translated to the instruction *setend be*, and *littleendian* to *setend le*, switching endianness to big endian and little endian respectively. Figure 4.2 provides a code example of the use of these instructions. The first instruction in both (a) and (b) copies a 32-bit integer into all four lanes of the destination register. In (a), this register is stored in little-endian mode, writing to memory a vector with $0x12345678$ in all lanes. In (b) big-endian mode is used, which stores the value $0x78563412$. Notice the use of *mem4x32* instead of *mem128* in (b); the former is translated to the *vst1.32* instruction by Qhasm, while the latter results in a *vst1.8* instruction. This distinction is important because of the way the data is actually stored in memory. When storing a value byte

[3]We explain why we were not able to eliminate the first block of *load_bigendian()* function calls in Section 5

Figure 4.2: An example of switching endianness

(a) Store little-endian

```
4x a = 0x12345678

mem128[i] = a
```

(b) Store big-endian

```
4x a = 0x12345678

bigendian
mem4x32[i] = a
littleendian
```

by byte, the endianness of the instruction is not relevant, but when storing blocks of four bytes it is. Load instructions have the same property of course.

The use of the *setend* instruction comes at the cost of two cycles, one for switching to big endian and one to switch back. We also observed that switching endianness a lot comes with additional latency. The first 16 words of the message schedule array contain the input message (see Section 2.2.2 and Section 4.2) in little endian format. We have to read this input in big endian mode (using the *vld1.32* instruction) during the first round of functions F and M [4]. However, since normal assembly memory accesses are also affected by the switched endianness, we would have to switch 16 times in this round of F to avoid reading constants the wrong way. We solved this problem by rewriting the constants used in this round to their little endian format; loading them in big endian mode reverts them back to their correct representation. This little trick resulted in 15 less switches in endianness, accompanied by a performance boost of about 200 cycles.

[4]This is no longer necessary after the first round of function M, as the values in the message schedule array are then results of calculations on big endian input.

Chapter 5

Results

5.1 Benchmark Results

As mentioned in Section 2.3.3, we used a program written by Bernstein to benchmark our implementation on a BeagleBoard Black, because the SUPERCOP benchmarking suite did not support benchmarks for parallel hashes at the time of writing. The reported values are the median result of 10.000 iterations. Tables 5.1 and 5.2 show the total cycle counts, and Tables 5.3 and 5.4 show the amount of cycles used for the functions M and F.

Implementation	**Median cycle count**
Reference implementation (C)	9386
Our implementation (ARM NEON)	6240
Improvement	33.5%

Table 5.1: Total cycle counts for 256-bit input messages

Implementation	**Median cycle count**
Reference implementation (C)	15833
Our implementation (ARM NEON)	11494
Improvement	27.5%

Table 5.2: Total cycle counts for 512-bit input messages

5.2 Discussion

The results above show that a significant speed-up can be obtained by using the NEON SIMD architecture on ARM processors. We are positive that

Implementation	**M**	**F**	**Others**
Reference implementation (C)	2062	6447	2939
Our implementation (ARM NEON)	1829	2771	1640
Improvement	11.3%	57.0%	-

Table 5.3: Cycle counts per function for 256-bit input messages

Implementation	**M**	**F**	**Others**
Reference implementation (C)	4105	12818	3015
Our implementation (ARM NEON)	3767	5602	2125
Improvement	8.2%	56.3%	-

Table 5.4: Cycle counts per function for 512-bit input messages

more can be done to improve the efficiency of our implementation, but due to time restrictions we were unable to do further optimizations.

As the tables above demonstrate, most of the performance gain was obtained by optimizing the function F. This can be explained by comparing the performance of the reference implementation to the lower bound we determined in Section 4.1; as the lower bound for function F is about 4000 cycles below the amount the reference implementation uses (as opposed to about 1000 cycles for M), this function had most room for improvement.

The 'others' category in the tables above contains everything not included in the functions F and M. This includes message padding, copying the initialization constants to the working variables and initial hash value, and storing the computed hash to memory. In comparison to the actual hash computation, this category is quite expensive in our implementation, but because of the limited time frame available we were not able to address this any further in the context of this thesis. We have omitted our improvement on this category as the compiler interleaves this code with other instructions to eliminate latencies, essentially reducing the cost of this category. This is why the total cycle count of the reference implementation is lower than the sum of the costs of the individual parts.

An example of further optimizations would be to eliminate the last remaining block of *load_bigendian()* function calls. This can be done by reading the initialization constants directly into registers during the first round of the function F, instead of copying those constants into the working variables. We were not able to do this optimization because we ran into instruction scheduling problems, resulting in a significant drop in performance.

Also, currently there are still some small latencies in our implementation. These were introduced when we managed to switch from unaligned memory access to aligned memory access. Since the first costs one additional cycle to execute, our performance was improved significantly, but some latencies that were ‘hidden’ by the additional cycle used by unaligned memory accesses were exposed after this optimization. Hiding these latencies should improve our results even further.

Bibliography

[1] Onur Açıızmez. Fast hashing on pentium SIMD architecture. Master's thesis, Oregon State University, 2004. 14

[2] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to, accessed: 30 May 2016. 13, 16

[3] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer-Verlag Berlin Heidelberg, 2012. http://cryptojedi.org/papers/#neoncrypto. 3, 11, 12, 18

[4] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the Security of the Winternitz One-Time Signature Scheme. In Abderrahmane Nitaj and David Pointcheval, editors, *Progress in Cryptology – AFRICACRYPT 2011: 4th International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings*, volume 6737 of *Lecture Notes in Computer Science*, pages 363–378. Springer Berlin Heidelberg, 2011. http://dx.doi.org/10.1007/978-3-642-21969-6_23. 8

[5] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 – December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer Berlin Heidelberg, 2011. http://dx.doi.org/10.1007/978-3-642-25405-5_8. 3, 8

[6] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle Signatures with Virtually Unlimited Signature Capacity. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security: 5th International*

Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007. Proceedings, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-72738-5_3. 3, 8

[7] Ana Karina D. S. de Oliveira and Julio López. An Efficient Software Implementation of the Hash-Based Signature Scheme MSS and Its Variants. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 366–383. Springer International Publishing, 2015. http://dx.doi.org/10.1007/978-3-319-22174-8_20. 3, 14

[8] C. Dods, N. P. Smart, and M. Stam. Hash based digital signature schemes. In Nigel P. Smart, editor, *Cryptography and Coding: 10th IMA International Conference, Cirencester, UK, December 19-21, 2005. Proceedings*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer Berlin Heidelberg, 2005. http://dx.doi.org/10.1007/11586821_8. 7

[9] Shay Gueron. A j-lanes tree hashing mode and j-lanes sha-256. *Journal of Information Security*, 4(1):7, 2013. 14

[10] Shay Gueron, Vlad Krasnov, et al. Simultaneous hashing of multiple messages. *Journal of Information Security*, 3(04):319, 2012. 14

[11] Andreas Hülsing. W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-38553-7_10. 8

[12] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. Forward Secure Signatures on Smart Cards. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography: 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, volume 7707 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-35999-6_5. 3, 8

[13] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, CSL-98, SRI International Palo Alto, 1979. 3, 5

[14] Adam Langley. Hash Based Signatures, 2013. https://www.imperialviolet.org/2013/07/18/hashsig.html, accessed: 02-29-2016. 5

[15] ARM Limited. Cortex-A8 Technical Reference Manual. Revision: r3p2, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html, accessed: 30 May 2016. 11, 12

[16] Ralph C. Merkle. A Certified Digital Signature. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer New York, 1990. http://dx.doi.org/10.1007/0-387-34805-0_21. 3, 5, 7

[17] National Institute of Standard and Technology. FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2. Technical report, DEPARTMENT OF COMMERCE, August 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf. 8

[18] National Institute of Standard and Technology. FIPS 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4. Technical report, DEPARTMENT OF COMMERCE, August 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4. 9, 10

[19] Ana Helena Sánchez and Francisco Rodríguez-Henríquez. NEON implementation of an attribute-based encryption scheme. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 322–338. Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-38980-1_20. 3

[20] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science. Proceedings*, pages 124–134. IEEE, 1994. http://dx.doi.org/10.1109/SFCS.1994.365700. 3

[21] Etienne Sobole. Cycle Counter for Cortex-A8. http://pulsar.webshaker.net/ccc/result.php?lng=us, accessed: 28 May 2016. 12

[22] Fang Song. A Note on Quantum Security for Post-Quantum Cryptography. In Michele Mosca, editor, *Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 246–265. Springer International Publishing, 2014. http://dx.doi.org/10.1007/978-3-319-11659-4_15. 3

[23] Timothy B. Terriberry. SIMD Assembly Tutorial: ARM NEON. https://people.xiph.org/~tterribe/daala/neon_tutorial.pdf, accessed: 23 May 2016. 11

Appendix A

C-code Structure

Here we provide a short overview of the structure of our code, in a C-like syntax. We have left out the padding and some repetitive parts for the sake of brevity and readability.

A.1 SHA-256 Code Structure (256-bit input)

```
int hash256_vect(unsigned char *out, unsigned char *in) {
        unsigned char padded[256];

        // Copy input to padded
        for (int i = 0; i < 128; ++i) padded[i] = in[i];

        // Padding: as described in section 2.2.1
        ...

        // Perform hashing calculations
        blocks_vect_256(out, padded);
}
void blocks_vect_256(unsigned char *statebytes, unsigned char *in) {
        uint32x4 abcdefgh[8];    // Working variables
        uint32x4 state[8];       // Intermediate hash values
        uint32x4 w[16];          // Message schedule array
        memcpy(w[0], in, 256);   // Copy input to message scheduling array

        // Initialize working variables and hash values
        load_bigendianx4(ivx4 +   0, abcdefgh[0]); state[0] = abcdefgh[0];
        load_bigendianx4(ivx4 +  16, abcdefgh[1]); state[1] = abcdefgh[1];
        ...                                         ...
        load_bigendianx4(ivx4 +  96, abcdefgh[6]); state[6] = abcdefgh[6];
        load_bigendianx4(ivx4 + 112, abcdefgh[7]); state[7] = abcdefgh[7];

        // Start NEON assembly code
        // vk -> contains constants as specified in FIPS 180-4
        F(w[0], vk, abcdefgh[0]); // Compress w0 through w15
        M(w[0]);                  // Prepare w16 through w31
        F(w[0], vk, abcdefgh[0]); // Compress w16 through w31
        M(w[0]);                  // Prepare w32 through w47
        F(w[0], vk, abcdefgh[0]); // Compress w32 through w47
        M(w[0]);                  // Prepare w48 through w63
        F(w[0], vk, abcdefgh[0]); // Compress w48 through w63

        // Add calculated block to intermediate hash value and
        // store result in output (done with NEON instructions)
        statebytes[0] = state[0] + abcdefgh[0];
        statebytes[1] = state[1] + abcdefgh[1];
        ...             ...
        statebytes[6] = state[6] + abcdefgh[6];
        statebytes[7] = state[7] + abcdefgh[7];

        return;
}
```

A.2 SHA-256 Code Structure (512-bit input)

```
int hash512_vect(unsigned char *out, unsigned char *in) {
        unsigned char padded[512];

        // Copy input to padded
        for (int i = 0; i < 256; ++i) padded[i] = in[i];

        // Padding: as described in section 2.2.1
        ...

        // Perform hashing calculations
        blocks_vect_512(out, padded);
}
void blocks_vect_512(unsigned char *statebytes, unsigned char *in) {
        uint32x4 abcdefgh[8];    // Working variables
        uint32x4 state[8];       // Intermediate hash values
        uint32x4 w[16];          // Message schedule array

        // Copy first input block to message scheduling array
        memcpy(w[0], in, 256);

        // Initialize working variables and hash values
        load_bigendianx4(ivx4 +   0, abcdefgh[0]); state[0] = abcdefgh[0];
        load_bigendianx4(ivx4 +  16, abcdefgh[1]); state[1] = abcdefgh[1];
        ...                                        ...
        load_bigendianx4(ivx4 +  96, abcdefgh[6]); state[6] = abcdefgh[6];
        load_bigendianx4(ivx4 + 112, abcdefgh[7]); state[7] = abcdefgh[7];

        // vk -> contains constants as specified in FIPS 180-4
        F(w[0], vk, abcdefgh[0]); // Compress w0 through w15
        M(w[0]);                  // Prepare w16 through w31
        F(w[0], vk, abcdefgh[0]); // Compress w16 through w31
        M(w[0]);                  // Prepare w32 through w47
        F(w[0], vk, abcdefgh[0]); // Compress w32 through w47
        M(w[0]);                  // Prepare w48 through w63
        F(w[0], vk, abcdefgh[0]); // Compress w48 through w63

        // Add calculated block to intermediate hash value
        abcdefgh[0] += state[0]; state[0] = abcdefgh[0];
        abcdefgh[1] += state[1]; state[1] = abcdefgh[1];
        ...                      ...
        abcdefgh[6] += state[6]; state[6] = abcdefgh[6];
        abcdefgh[7] += state[7]; state[7] = abcdefgh[7];

        // Copy second input block to message scheduling array
        in += 256;
        memcpy(w[0], in, 256);

        F(w[0], vk, abcdefgh[0]); // Compress w0 through w15
        M(w[0]);                  // Prepare w16 through w31
        F(w[0], vk, abcdefgh[0]); // Compress w16 through w31
        M(w[0]);                  // Prepare w32 through w47
        F(w[0], vk, abcdefgh[0]); // Compress w32 through w47
        M(w[0]);                  // Prepare w48 through w63
        F(w[0], vk, abcdefgh[0]); // Compress w48 through w63

        // Add calculated block to intermediate hash value and
        // store result in output (done with NEON instructions)
        statebytes[0] = state[0] + abcdefgh[0];
        statebytes[1] = state[1] + abcdefgh[1];
        ...             ...
        statebytes[6] = state[6] + abcdefgh[6];
        statebytes[7] = state[7] + abcdefgh[7];

        return;
}
```

Appendix B

Running our implementation

Our implementation can be found at `https://gitlab.science.ru.nl/wlinde/neon_sha256`. To run our test code, you will also need the Qhasm implementation we used, which can be found at `https://gitlab.science.ru.nl/wlinde/Qhasm`. The *qhasm* folder should be placed in the *Implementation* folder found in the 'neon_sha256' repository, outside of the *c-code* folder. Then simply run the Makefile.

To use just our SHA-256 implementation, you will need the following files: *hash_vect.c*, *hash_vect.h*, *vFMopt.s*, *vaddmov.s*, and *vaddstr.s*. The files with suffix .s contain the assembly code generated by Qhasm from the corresponding files with suffix .q, which contain Qhasm code. We have included the .s files in our repository for convenience. These can be manually generated using Qhasm; for example, to create vFMopt.s, use the command 'qhasm-arm vFMopt.q vFMopt.s'. To calculate a hash with our implementation, you need to call the functions *hash256_vect* for 256-bit input and *hash512_vect* for 512-bit input. These functions use the assembly code in the .s files mentioned above. The input for these functions must consist of four already interleaved messages of 256 or 512 bits each. We have provided a function that interleaves four messages of equal length (using 32-bit blocks) in the file 'util.c'. The output of the hash functions is still interleaved; use for example our 'deleave' function (also in util.c) to de-interleave the output into four separate hashes.