

BACHELOR THESIS
COMPUTER SCIENCE



RADBOUD UNIVERSITY

Optimization of the NTT function
on ARMv8-A SVE

Author:

Gia Linh Hoang
s4553519

First supervisor/assessor:

Dr. Peter Schwabe
peter@cryptojedi.org

Second assessor:

Prof. Dr. Lejla Batina
lejla@cs.ru.nl

June 25, 2018

Abstract

Quantum computers could break today's (2018) public-key cryptosystems if they are ever built. To prevent this from happening NIST started a post-quantum cryptography standardization process [1]. Kyber [2] is a key encapsulation mechanism for post-quantum cryptography submitted to this process and got accepted to the first round. It uses the NTT function to compute the product of two polynomials efficiently. In this research we will optimize on assembly level the NTT function for Kyber on the ARMv8-A SVE processor architecture, which comes with the new feature that the vectors are scalable [3]. The NTT version made for this thesis could be assumed to be optimized for ARMv8-A SVE.

Acknowledgements

I want to thank Peter Schwabe for being my supervisor, helping out with optimizing the code and providing insights on the subject. Furthermore I want to thank Lejla Batina for being the second assessor. Finally I want to thank Thom Wiggers for taking the time to help with setting up the ODROID-C2.

Contents

1	Introduction	3
2	Preliminaries	4
2.1	Kyber	4
2.2	DFT, FFT and NTT	5
2.3	Reduction methods	6
2.3.1	Montgomery reduction	7
2.3.2	Barrett reduction	7
2.4	ARMv8-A SVE	7
2.4.1	Registers	8
2.4.2	Most used registers and instructions	9
2.4.3	SVE vectorization	11
3	Optimizing the NTT	15
3.1	NTT in Kyber	15
3.2	Optimizing the seventh NTT level	16
3.2.1	The idea	16
3.2.2	The route to assembly	18
3.2.3	Optimizing the assembly code	19
3.3	Optimizing the sixth NTT level	21
3.3.1	The idea	21
3.3.2	The route to assembly	22
3.4	The levels 7 until 0	23
3.4.1	A code framework for the levels 6 until 0	25
3.5	Problems concerning vector lengths?	26
3.6	Interleave for vector length 128	28
3.6.1	Level 0	28
3.6.2	Level 1	30
3.6.3	Level 2	34
3.7	Interleaving for vector length 256	37
3.7.1	Level 0	37
3.7.2	Level 1	37
3.7.3	Level 2	37

3.7.4	Level 3	40
3.8	Interleaving for vector length 512	42
3.8.1	Level 0, 1 and 2	42
3.8.2	Level 3	42
3.8.3	Level 4	45
3.9	Interleaving for vector length 1024	45
3.9.1	Level 0, 1, 2 and 3	45
3.9.2	Level 4	46
3.9.3	Level 5	49
3.10	Interleaving for vector length 2048	49
3.10.1	Level 0, 1, 2 and 3	49
3.10.2	Level 4	49
3.10.3	Level 5	56
3.10.4	Level 6	56
4	Results	57
4.1	Counting instructions	57
4.2	Summary	59
5	Related Work	60
6	Conclusions	62
A	Appendix	66
A.1	How to set up the compiler and emulator	66
A.2	How to compile and run code	67
A.3	Level 7 written out in C	67
A.4	Level 7 compiled with O3 in assembly	67
A.5	Level 6 with 2 loops	67
A.6	Code framework version	67
A.6.1	Level 7	68
A.6.2	Level 6	68
A.6.3	Level 5	68
A.7	Interleaved version	68

Chapter 1

Introduction

Many public-key cryptosystems which are used today (2018) are not secure anymore if quantum computers are ever built, because their computing power is drastically improved with respect to current computers [1]. There has been a lot of research on quantum computers so it is not unlikely they will be built in the future [1].

NIST wants to prevent quantum computers from breaking the current public-key cryptosystems and started a post-quantum cryptography standardization process [1]. Kyber, a key encapsulation mechanism for post-quantum cryptography [2], was one of the algorithms submitted to this process and got accepted to the first round.

In Kyber the NTT function is used to compute the product of two polynomials efficiently, because the naive multiplication of two polynomials has a high running time because of the nested for loop in the computation. For this thesis we will optimize the NTT function as it is implemented in Kyber in assembly on ARMv8-A SVE.

The ARMv8-A SVE (Scalable Vector Extension) architecture was presented by ARM in 2016 at the Hot Chips symposium in Cupertino [3]. The architecture comes with the new feature that vectors are scalable [3] and is not implemented (yet), but the compiler and emulator are currently (May 2018) available on the ARM website [4] and can be run on ARMv8-A.

This leads to our research question: How to optimize the NTT function in Kyber on ARMv8-A SVE?

After optimizing the NTT, we can check if it is really optimized by counting the number of operations executed and compare them with the number of operations for the less optimized version.

Chapter 2

Preliminaries

2.1 Kyber

NIST set up a post-quantum cryptography public-key algorithm contest because they want to standardize an algorithm to be secure on computers nowadays and post-quantum ones. The research into building a post-quantum computer is going on and today's cryptography will not be secure if large post-quantum computers will be implemented [1].

Avanzi, Bos, Ducas, Kiltz, Lepoint, Lyubashevsky, Schanck, Schwabe, Seiler and Stehlé submitted Kyber, which is a key encapsulation mechanism for post-quantum cryptography [2], to this contest and it got accepted to the first round.

The following explanation is a simplified version of Kyber, but shows the main idea. We will look at the key generation, encapsulation and decapsulation. Kyber has an active and passive part; in this thesis we will only discuss the passive part which means that Kyber uses a passively secure encryption as a building block.

We begin with defining the polynomial ring in which we operate: $R_q = \mathbb{Z}_{7681}[X]/(X^{256+1})$. The multiplication \cdot is a coefficient-wise multiplication. We also have a centred binomial distribution ψ to create noise which takes as input $a_i, b_i \in \{0, 1\}$ and gives an output $\sum_{i=1}^x a_i - b_i$.

Listing 2.8 shows the key generation algorithm, where **Parse()** samples elements in R_q and **Shake128()** gives a secure random output. The algorithm in the listing works with polynomials to keep it simple, but in the real implementation of Kyber those are matrices over R_q . Listing 2.9 shows the encapsulation of a key. Listing 2.10 shows the decryption of a message. In these listings a letter with a hat stands for the NTT form of that variable.

```
1  r ← {0,1}256
2  â ← Parse( Shake128( r ) )
3  (s, e) ←  $\psi$ 
4  ŝ = NTT(s)
5  b = NTT-1(ŝ · â) + e
```

```

6
7 PublicKey = (b, r)
8 PrivateKey =  $\hat{s}$ 

```

Listing 2.1: Key generation

```

1 Input: (PublicKey pk = (b, r), Message m)
2  $\hat{a} \leftarrow \mathbf{Parse}(\mathbf{Shake128}(r))$ 
3  $(s', e', e'') \leftarrow \psi$ 
4  $\hat{s}' = \mathbf{NTT}(s')$ 
5  $u = \mathbf{NTT}^{-1}(\hat{a} \cdot \hat{s}') + e'$ 
6  $\hat{b} = \mathbf{NTT}(b)$ 
7  $v = \mathbf{NTT}^{-1}(\hat{b} \cdot \hat{s}') + e'' + \mathbf{Encode}(m)$ 
8
9 Ciphertext = (u, v)

```

Listing 2.2: Encryption

```

1 Input: (Ciphertext c = (u, v), PrivateKey s)
2  $\hat{u} = \mathbf{NTT}(u)$ 
3  $\kappa = \mathbf{NTT}^{-1}(\hat{u} \cdot \hat{s})$ 
4  $m = \mathbf{Decode}(v - \kappa)$ 

```

Listing 2.3: Decryption

To encrypt we need to encode the message m in order to add noise for security. This happens in the **Encode()** function. By choosing the centred binomial distribution ψ we define how big the noise is. If the noise is too big, there is a chance decryption does not give the correct plain text. **Encode()** takes as input a vector of polynomials and encodes each polynomial and outputs the byte arrays concatenated in one byte array [2]. The **Decode()** function decodes the message. If the encoded integer in the message is close to 0, the decoded bit is 0, if it is closer to 3840, the bit is 1.

We see that the NTT is used multiple times in Kyber, which is the part we want to speed up.

2.2 DFT, FFT and NTT

The NTT is a specialization of the DFT, which stands for Discrete Fourier Transform. The DFT maps a vector x to a vector y linearly [5]. The difference between NTT and DFT is that NTT does a transform over the quotient ring $\mathbb{Z}/p\mathbb{Z}$ where DFT does it over the complex numbers [6].

The DFT can be computed with the Fast Fourier Transform (FFT) [7] and has a long history [8]. In 1965 Cooley and Tukey described a general form of the FFT [9] which was seen as a new version [8]. Following Fürer [7] the DFT is a linear mapping of vector $x = (x_0, x_1, \dots, x_{n-1})^T$ to vector $y = (y_0, y_1, \dots, y_{n-1})^T$, which looks like: $y_k = \sum_{j=0}^{n-1} x_j \cdot \omega^{jk}$, $k \in \{0, \dots, n-1\}$:

$$\begin{bmatrix} \omega^{0 \cdot 0} & \omega^{1 \cdot 0} & \omega^{2 \cdot 0} & \dots & \dots & \omega^{(n-1) \cdot 0} \\ \omega^{0 \cdot 1} & \omega^{1 \cdot 1} & \omega^{2 \cdot 1} & \dots & \dots & \omega^{(n-1) \cdot 1} \\ \omega^{0 \cdot 2} & \omega^{1 \cdot 2} & \omega^{2 \cdot 2} & \dots & \dots & \omega^{(n-1) \cdot 2} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ \omega^{0 \cdot (n-1)} & \omega^{1 \cdot (n-1)} & \omega^{2 \cdot (n-1)} & \dots & \dots & \omega^{(n-1) \cdot (n-1)} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_3 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_3 \\ \vdots \\ \vdots \\ y_{n-1} \end{bmatrix}$$

For the n -th root of unity it holds that $\omega^n = 1$. If n is the smallest integer such that $k \in \{1, \dots, n\}$ and $\omega^k = 1$, then ω is the primitive n -th root of unity [10]. The fact that it is a primitive n -th root of unity has the effect that the NTT has an inverse. This property is used to compute the multiplication of two polynomials efficiently.

To compute the inverse of the NTT of vector y , which is vector x , we have the formula $x_j = \frac{1}{n} \cdot \sum_{k=0}^{n-1} y_k \cdot \omega^{-jk}$, $j \in \{0, \dots, n-1\}$:

$$\frac{1}{n} \cdot \begin{bmatrix} \omega^{-0 \cdot 0} & \omega^{-0 \cdot 1} & \omega^{-0 \cdot 2} & \dots & \dots & \omega^{-0 \cdot (n-1)} \\ \omega^{-1 \cdot 0} & \omega^{-1 \cdot 1} & \omega^{-1 \cdot 2} & \dots & \dots & \omega^{-1 \cdot (n-1)} \\ \omega^{-2 \cdot 0} & \omega^{-2 \cdot 1} & \omega^{-2 \cdot 2} & \dots & \dots & \omega^{-2 \cdot (n-1)} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ \omega^{-(n-1) \cdot 0} & \omega^{-(n-1) \cdot 1} & \omega^{-(n-1) \cdot 2} & \dots & \dots & \omega^{-(n-1) \cdot (n-1)} \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_3 \\ \vdots \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_3 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix}$$

We can see the vectors x and y as polynomials, where each element in the vector is a coefficient in the polynomial. With this we can compute the product of two polynomials a and b as: $DFT^{-1}(DFT(a) \cdot DFT(b))$. Generally we need to zero pad the polynomials for the DFT.

For Kyber we use a specialized version of the DFT, namely the negacyclic NTT. So the product of two polynomials a and b is computed in Kyber as: $NTT^{-1}(NTT(a) \cdot NTT(b))$, where \cdot is a pointwise multiplication [11]. In Kyber [11] we compute the NTT of a polynomial $g = \sum_{i=0}^{255} g_i X^i$. The authors chose the constants $\omega = 3844$, $\psi = \sqrt{\omega} = 62$, $n = 256$, $q = 7681$. The NTT of g is then computed as: $\sum_{i=0}^{255} (\sum_{j=0}^{255} \psi^j g_j \omega^{ij}) X^i$ working over the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. This version of the DFT, the NTT, can be used without zero padding because of the reduction modulo $(X^n + 1)$.

2.3 Reduction methods

Kyber uses two reduction methods modulo $q = 7681$ in the NTT to compute modular reductions faster. Those reductions are the Montgomery and

Barrett reductions. NewHope [12] was the first to propose this for the NTT in lattice-based cryptography.

2.3.1 Montgomery reduction

In 1985 Montgomery introduced a modular reduction algorithm which is faster than the schoolbook reduction modulo N [13]. This Montgomery reduction is used in the NTT in Kyber to do faster reductions. The algorithm needs as input: $m = (m_{n-1} \dots m_1 m_0)_b$ and $\gcd(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \pmod{b}$ and $T = (t_{2n-1} \dots t_1 t_0)$ [14]:

```

1  A  $\leftarrow (a_{2n-1} \dots a_1 a_0) \leftarrow T$ 
2  for  $i$  in range  $(0, n-1)$ :
3       $u_i \leftarrow a_i m' \pmod{b}$ 
4       $A \leftarrow A + u_i m b^i$ 
5   $A \leftarrow A/b^n$ 
6  if  $A \geq m$ , do  $A \leftarrow A - m$ 
7  Return  $A \equiv T \cdot R^{-1} \pmod{m}$ 

```

2.3.2 Barrett reduction

Another reduction method which is used in the NTT in Kyber is the Barrett reduction introduced by Barrett [15]. The algorithm has as input $x = (x_{2k-1} \dots x_1 x_0)_b$, $m = (m_{k-1} \dots m_1 m_0)_b$ with $m_{k-1} \neq 0$ and $\mu = \lfloor b^{2k}/m \rfloor$ [14]:

```

1   $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor, q_2 \leftarrow q_1 \cdot \mu, q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$ 
2   $r_1 \leftarrow x \pmod{b^{k+1}}, r_2 \leftarrow q_3 \cdot m \pmod{b^{k+1}}, r \leftarrow r_1 - r_2$ 
3  if  $r < 0$ , do  $r \leftarrow r + b^{k+1}$ 
4  while  $r \geq m$ , do:
5       $r \leftarrow r - m$ 
6  Return  $r$ 

```

2.4 ARMv8-A SVE

For this thesis we will optimize the NTT as it is implemented in Kyber on ARMv8-A SVE (Scalable Vector Extension) architecture, which was presented by ARM in 2016 at the Hot Chips symposium in Cupertino [3]. It is an extension to the ARMv8-A architecture [16], with key features which are new compared to ARMv8-A. Two of those key features which are going to be used in this thesis are:

- Vector registers are scalable
- Instructions can be predicated (relates to the scalable feature)

SVE has not been implemented (yet) but the compiler and emulator are already available on the ARM website (May 2018) [4]. The compiler and

emulator can run on both ARMv8-A architecture and non-ARMv8-A processors. Working with the non-ARMv8-A architecture requires more steps to get it working. Appendix A.1 explains how to set up the compiler and emulator on the ARMv8-A architecture based on the tutorial on the ARM site [17]. Appendix A.2 explains how to use it. For this thesis an ODROID-C2 (with ARMv8-A architecture) is used.

2.4.1 Registers

The ARMv8-A SVE manual describes the registers which are different from the ARMv8-A registers. The descriptions of the following registers come from the SVE manual [16]. SVE has four different classes of registers:

- Vector registers
- Predicate registers
- Scalar registers
- First fault registers (FFR)

Only the first three registers are relevant for this thesis, thus only those will be discussed.

Vector Registers

Vector registers are the registers Z0–Z31. The size of the registers is defined before the program is executed by passing it as a value to the emulator; on real hardware this will be fixed by the hardware itself. This size holds for all vector registers during compiling and can be a value between 128 and 2048 bits and is a multiple of 128 bits.

The elements in one vector register are all 8, 16, 32, 64 or 128 bits, which is defined within an assembly instruction. Some instructions require the length of an element as a parameter, defined in the ‘size specifier’ field. For example `add <Zd>.<T>, <Zn>.<T>, <Zm>.<T>` adds up the values in the first source vector `Zn` and second source vector `Zm` and stores the result in the destination vector `Zd`. `<T>` is the size specifier and depending on the instruction the size of elements in a vector could be:

Size specifier	Length in bits
.q	128
.d	64
.s	32
.h	16
.b	8

Table 2.1: Size specifiers

Predicate registers

Predicate registers are the registers P0-P15. The length of a predicate register is 1/8 the length of a vector register. For example if the vector length is defined at 256 bits, the length of the predicate registers is 32 bits.

Because the vector lengths range between 128 and 1024 bits and a predicate register is 1/8 the length of a vector register, the length of a predicate register ranges between 16 and 256 bits. Thus, the elements in one predicate register are all 1, 2, 4 or 8 bits. If the lowest bit of an element is 1 the predicate element is true, and false if it is set at 0.

One predicate element is mapped to a vector element. The corresponding vector element is active if the predicate element is true and inactive otherwise. The state of such an predicate element can be seen as a flag state in ‘traditional’ instruction sets.

To make use of scalable vectors, predicated instructions are needed. An instruction which supports predication only affects the active elements of the destination vector. The inactive elements are set to zero (zeroing predication Pg/Z) or remain untouched (merging predication Pg/M).

To be able to execute those predicated instructions ARM introduced governing predicates. One vector register is linked to a governing predicate register. But one governing predicate register can be linked to multiple vector registers. This linking happens by mapping an element in the governing predicate register to an element in the vector register. If the element in the predicate register is set to TRUE, this means the corresponding element in the vector register is active and inactive if the predicate register element is FALSE.

Scalar registers

Scalar registers (X0-X30) are the ‘basic’ registers. They consist of 64-bit single data elements. One scalar register can be used to hold a scalar element or multiple registers can be used to hold elements of an array. When used as an array, the first scalar register refers to the first element of the array, the next register to the second element and so on.

When referred to W0-W30 registers, the first 32 bits of the elements of the X0-X30 registers are meant. For example W0 represents the first 32 bits of register X1. Register X31 or W31 does not exist, but can be used as a 64-bit (XZR) or a 32-bit (WZR) zero register. It can also be used as the stack pointer (SP) [18].

2.4.2 Most used registers and instructions

The following registers and instructions are relevant instructions and registers for this thesis, their description can be found in the manual [16]:

- Normally `wzr` and `xzr` are used as a zero register, and sometimes as SP. In this thesis we use them as zero registers.
- `mov dest, src`: move `src` to `dest`.
- `whilelo pred, A, B`: go on if `A` is lower than `B`.
- `lsl`: logical shift left.
- `msl`: logical shift left, but fill the lower bits with ones [19].
- `ld1h dest, governing scalable predicate register, [src, offset register, lsl #1]`: predicated load of half words (16 bit elements) from addresses `[src : src+(value in offset register)*2]` to `dest`.
- `st1h src, governing scalable predicate register, [dest, offset register, lsl #1]`: predicated store of half words from `src` to addresses `[dest : dest+(value in offset register)*2]`.
- `sub dest, src1, src2`: `dest = src1 - src2`.
- `uzp1 dest, src1, src2`: concatenate even numbered elements of `src1` and `src2` and place them in `dest`.
- `uzp2 dest, src1, src2`: does the same as `uzp1`, but with odd numbered elements.

src1	0	1	2	3
src2	4	5	6	7
dest	0	2	4	6

- `trn1 dest, src1, src2`: interleave even elements of `src1` and `src2` and place them in `dest`.

src1	0	1	2	3
src2	4	5	6	7
dest	0	4	2	6

- `trn2 dest, src1, src2`: does the same as `trn1`, but with odd numbered elements.

- **zip1 dest, src1, src2**: interleave between elements of the lower halves of **src1** and **src2** and put the result in **dest**.

src1	0	1	2	3
src2	4	5	6	7
dest	0	4	1	5

- **zip2 dest, src1, src2**: does the same as **zip1**, but with the upper halves of **src1** and **src2**.
- **ext dest.b, src1.b, src2.b, #imm**: extract the bottom of **src1** from the byte indicated in **#imm**, and fill the rest of **dest** with the rest of **src2**.
- **:lo12:address**: group relocation, in this thesis it is used to initialize the array **zetas**.

2.4.3 SVE vectorization

SVE vectorization will be used to optimize the NTT function. We will first look at what vectorization is and then at vectorization on SVE.

Vectorization

A loop can be vectorized in order to let it run more efficiently. For example let us consider the following code:

```

1 int A = {1,2,3,4,5,6,7,8};
2 int B = {9,10,11,12,13,14,15,16};
3 int C[8];
4
5 for(int i = 0; i < 8; i++){
6     C[i] = A[i] + B[i];
7 }

```

Listing 2.4: Adding two arrays

We can see the loop iterates eight times, doing one addition every iteration. A vectorized version of this code could be:

```

1 int A = {1,2,3,4,5,6,7,8};
2 int B = {9,10,11,12,13,14,15,16};
3 int C[8];
4
5 for(int i = 0; i < 2; i+=4){
6     C[i:i+3] = A[i:i+3] + B[i:i+3];
7 }

```

Listing 2.5: Adding two arrays (vectorized version)

The loop iterates two times in this vectorized version. In each iteration four additions are done in parallel; the following operations will happen in the first iteration:

```

1 C[0] = A[0] + B[0];
2 C[1] = A[1] + B[1];
3 C[2] = A[2] + B[2];
4 C[3] = A[3] + B[3];

```

Listing 2.6: First iteration of vectorized version

And the second iteration:

```

1 C[4] = A[4] + B[4];
2 C[5] = A[5] + B[5];
3 C[6] = A[6] + B[6];
4 C[7] = A[7] + B[7];

```

Listing 2.7: Second iteration of vectorized version

Vectorization is much more efficient because operations are done in parallel, which causes the program to run in less time.

SVE vectorization

Predicated instructions can run with different vector lengths, this is the idea of SVE. We can see the difference between predicated instructions and normal instructions in the following example.

The `example.c` program in the tutorial [17] initializes two arrays `b` and `c`, each filled with 1024 random integers, subtracts `c` from `b` and stores the result in array `a`. At last the program prints the result array `a`. The `subtract_arrays()` function in the program, the part that does subtraction, is the interesting part. We can see this program listed below.

```

1 #define ARRAYSIZE 1024
2 int a[ARRAYSIZE];
3 int b[ARRAYSIZE];
4 int c[ARRAYSIZE];
5
6 void subtract_arrays(int *restrict a, int *restrict b, int *restrict c){
7     for (int i = 0; i < ARRAYSIZE; i++){
8         a[i] = b[i] - c[i];
9     }
10 }

```

Listing 2.8: `subtract_arrays()` of tutorial

If we put the `subtract_arrays()` function in a C file and compile it to assembly with optimization level 1 we get unpredicated instructions which make use of scalar registers. The code is represented in Listing 2.6 (with modified labels, added comments and removed compiler generated comments to make it more clear):

```

1  .text
2  .file "subtract.c"
3  .globl subtract_arrays
4  .p2align 2
5  .type subtract_arrays,@function
6
7  subtract_arrays:
8  .cfi_startproc
9  mov    x8, xzr                //x8 = 0
10
11 .looptop:
12     ldr    w9, [x1, x8]        //load in b[x8]
13     ldr    w10, [x2, x8]       //load in a[x8]
14     sub    w9, w9, w10         //c[x8] = b[x8] - a[x8]
15     str    w9, [x0, x8]        //store c[x8]
16     add    x8, x8, #4          //increment x8
17     cmp    x8, #1, lsl #12     //if x8 < 4096, go on
18     b.ne   .looptop           //return to .looptop
19     ret
20
21 .Lfunc_end0:
22 .size subtract_arrays, .Lfunc_end0-subtract_arrays
23 .cfi_endproc

```

Listing 2.9: Assembly of subtract_arrays() in level O1

When we compile this function to assembly with optimization level 3 we get following code (also with modified labels and removed comments to make it more clear):

```

1  .text
2  .globl subtract_arrays
3  .p2align 2
4  .type subtract_arrays,@function
5
6  subtract_arrays:
7  .cfi_startproc
8  .cfi_def_cfa_offset 0
9  orr    w9, wzr, #1024        //store 1024 in w9
10 mov    x8, xzr                //x8 = 0
11 whilelo p0.s, xzr, x9        //if (0 < x9), proceed
12
13 .looptop:
14     //x0 = &a[0], x1 = &b[0], x2 = &c[0], x9 = arraylength
15     //z0 and z1 are temporary vector registers
16     //VL = vector length
17     ld1w   {z0.s}, p0/z, [x1, x8, lsl #2] //load b[x8 : x8+VL] in z0
18     ld1w   {z1.s}, p0/z, [x2, x8, lsl #2] //load c[x8 : x8+VL] in z1
19     sub    z0.s, z0.s, z1.s         //z0 = z0 - z1
20     st1w   {z0.s}, p0, [x0, x8, lsl #2] //store z0 in a[x8 : x8+VL]
21     incw   x8                      //increment x8 by vector length
22     whilelo p0.s, x8, x9          //if x8 < x9, go on
23     b.mi   .looptop              //if x8 < x9, return to .looptop

```



```

24 | ret
25 | .cfi_endproc

```

Listing 2.10: Assembly of `subtract_arrays()` in level O3

In every iteration of the loop, parts of array `b` and `c` are loaded in:

```

17 | ld1w {z0.s}, p0/z, [x1, x8, lsl #2] //load b[x8 : x8+VL] in z0
18 | ld1w {z1.s}, p0/z, [x2, x8, lsl #2] //load c[x8 : x8+VL] in z1

```

At line 16, `ld1w` loads words (32 bit elements) of array `b` in vector registers `z0.s` (also 32 bit elements) [16]. Here, `x1` is the starting address of array `b`, and `x8` is the loop counter. Loading is done by computing the starting address from where to load: starting address of `b` + `x8`·4. The same idea also happens at line 17 for array `c`.

At line 20 `x8` is incremented by the vector length in each iteration:

```

20 | incw x8 //increment x8 by vector length

```

`ld1w` and `st1w` are linked with the predicate `p0`, which causes the load and store to affect only the part of the array which is needed for that iteration; the unnecessary parts will be set to zero (zeroing predication). Thus for every iteration the predicated instructions take into account the vector length, which makes it possible to run the code with any vector length.

Chapter 3

Optimizing the NTT

3.1 NTT in Kyber

The NTT is used in Kyber to compute the product of two polynomials. As we can see in the code snippet below, the NTT function is implemented in Kyber with a triple nested for-loop. The outer for-loop iterates through the eight levels of the NTT and the inner loops iterate through the elements of the given array `p` while doing operations on those elements. The array `p` represents the polynomial and the elements of `p` represent the coefficients of this polynomial. This array always has 256 elements, thus the polynomial has 256 coefficients. The array `zetas` is the precomputed power of ψ for Kyber. `montgomery_reduce()` and `barrett_reduce()` are used instead of the normal modulo operation in order to reduce the number of cycles [10].

```
1 extern const uint16_t zetas[];
2 KYBER_N = 256;
3 KYBER_Q = 7681;
4
5 void ntt(uint16_t *p){
6     int level, start, j, k;
7     uint16_t zeta, t;
8     k = 1;
9
10    for(level = 7; level >= 0; level--){
11        for(start = 0; start < KYBER_N; start = j + (1<<level)){
12            zeta = zetas[k++];
13            for(j = start; j < start + (1<<level); ++j){
14                t = montgomery_reduce((uint32_t)zeta * p[j + (1<<level)]);
15                p[j + (1<<level)] = barrett_reduce(p[j] + 4*KYBER_Q - t);
16                p[j] = barrett_reduce(p[j] + t);
17            }
18        }
19    }
20 }
```

Listing 3.1: NTT function in C

3.2 Optimizing the seventh NTT level

3.2.1 The idea

The first level of the NTT is actually the seventh level. We will count from 7 to 0 for the levels. The seventh level is the first iteration of the most outer for-loop in Listing 3.1. We can see that the seventh level only loops through the second nested for-loop once, because the condition `start < KYBER_N` is not met anymore after one iteration. Thus for this level, the second nested for-loop has not to be taken into account and we only have to look at the third nested for-loop.

The seventh level of the NTT can be thought of as follows. Array `p` is used to compute the new version of array `p` after the for-loop, which we will call `p'` from now on. In the first iteration we see:

$$\begin{aligned} p[0]' &= p[0] + (\text{zeta} \cdot p[128]) \\ p[128]' &= p[0] + 4 \cdot 128 - (\text{zeta} \cdot p[128]) \end{aligned}$$

This is represented graphically in Figure 3.1.

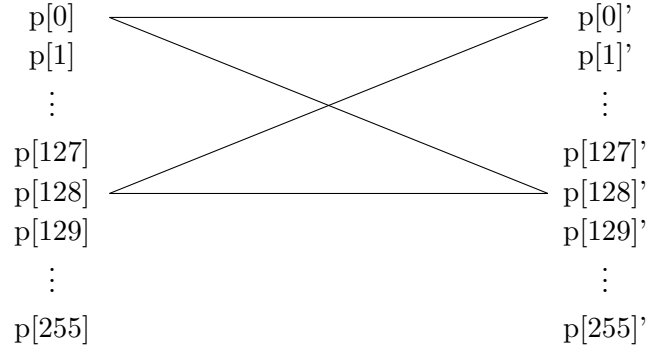


Figure 3.1: The first butterfly operation in level 7

This is a Gentleman-Sande butterfly operation of the FFT [20] which happens pairwise for every $p[j]$ and $p[j+128]$ for $j \in [0, \dots, 127]$. In Figure 3.2 we can see the second butterfly operation $p[1] * p[129]$. Figure 3.3 displays what happens in the last operation $p[127] * p[255]$ (green lines).

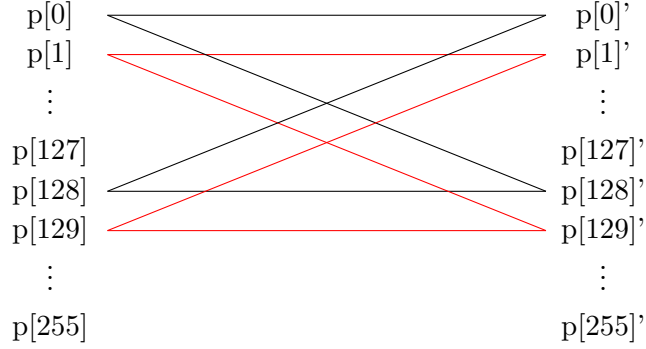


Figure 3.2: The first and second butterfly operation in level 7

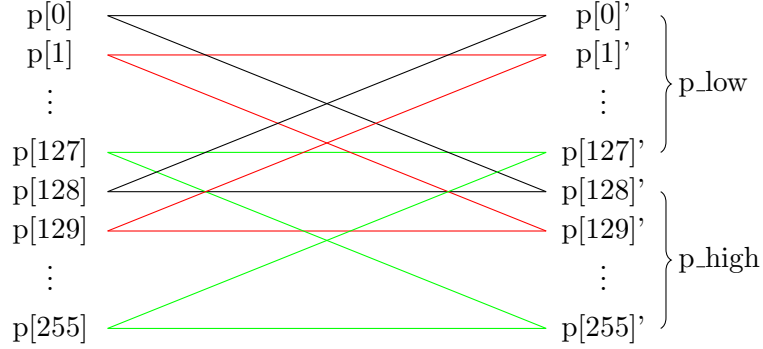


Figure 3.3: The first, second and last butterfly operation in level 7

We see that the operations are done independently, thus vectorization can be done easily. The idea is to take $p[128:255]$ (p_high) to compute t . Then we take $p[0:127]$ (p_low) to compute $p[0:127]'$ and $p[128:255]'$. With vector length 2048 bits this all happens in one iteration of the loop, instead of in 128 iterations when no vectorization has taken place.

From now on we will refer to one iteration of the second nested for-loop in Listing 3.1 as a packet of butterfly operations. The inputs to a packet of butterfly operations are a p_low and a p_high . For instance at this level, the butterfly operations of $p[0:127]'$ (p_low) and $p[128:255]'$ (p_high) happen in one iteration of this second nested for-loop, and thus one packet of butterfly operations.

Every packet multiplies p_high with an element of the array `zetas` to compute p' . The first packet uses `zetas[1]`, the second packet uses `zetas[2]`, and so on.

3.2.2 The route to assembly

To optimize the code in assembly, we first take the level out of the for-loop. The `ntt()` function then becomes:

```

1 void ntt(uint16_t *p){
2     int level, start, j, k;
3     uint16_t zeta, t;
4
5     k = 1;
6
7     nttlevel7(p, &k);
8
9     for(level = 6; level >= 0; level--){
10         for(start = 0; start < KYBER_N; start = j + (1<<level)){
11             zeta = zetas[k++];
12             for(j = start; j < start + (1<<level); ++j){
13                 t = montgomery_reduce((uint32_t)zeta * p[j + (1<<level)]);
14
15                 p[j + (1<<level)] = barrett_reduce(p[j] + 4*KYBER_Q - t);
16                 p[j] = barrett_reduce(p[j] + t);
17             }
18         }
19     }
20 }
```

Then we rewrite the C function such that it does not contain any other function calls. That way we can compile the C code to assembly using optimization level O3 and understand the assembly code more easily. We can then use the assembly code as a starting point and optimize it further. So we have to write out `montgomery_reduce()` and `barrett_reduce()` in `nttlevel7()`. The result of `nttlevel7()` can be seen in Appendix A.3.

When this is compiled to assembly with optimization level O3 we get the assembly code in Appendix A.4. Lines 13-32 initialize the variables and lines 33-84 represent the for-loop we take into account for this level. As we can see, the compiler already optimized the code, but we could make it more efficient.

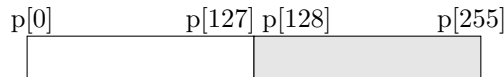
The loop in Appendix A.4 does the following:

1. Line 34: compute the begin address of `p`.
2. Line 35:

```

35 ld1h    {z5.h}, p1/z, [x11, x10, lsl #1]
```

Take the upper half of `p` (`= p[128:255] = p_high`), which is the grey coloured part in the figure.



3. Lines 36-37:

36	<code>uunpklo</code>	<code>z6.s, z5.h</code>
37	<code>uunpkhi</code>	<code>z5.s, z5.h</code>

Take the lower half of `p_high` (vector register `z5`), zero extend each element (16 bit) to double the size (32 bit) and place them in vector register `z6` using the instruction `uunpklo`. Do the same with the upper half of `p_high` and place the result in `z5` using the instruction `uunpkhi`.

In instruction `uunpklo z6.s,z5.h` we use the size specifiers `.s` and `.h` because `z5.h` has 16 bit elements and we turn them into 32 bit elements in `z6.s`. The same applies to `uunpkhi z5.s,z5.h`.

<code>z5.h</code>	<code>p[128]</code>	<code>p[129]</code>	<code>...</code>	<code>p[191]</code>	<code>p[192]</code>	<code>p[193]</code>	<code>...</code>	<code>p[255]</code>
<code>z6.s</code>	<code>p[128]000</code>	<code>p[129]000</code>	<code>...</code>	<code>p[191]000</code>				
<code>z5.s</code>	<code>p[192]000</code>	<code>p[193]000</code>	<code>...</code>	<code>p[255]000</code>				

These operations must occur, because the elements in `z5` are 16 bits, but we need the elements to be 32 bits in order to do operations on them without the elements overflowing.

4. Lines 38-54 (line 48 excluded): compute `t` and `KYBER_Q_quadrupled_decremented`.
5. Line 48: take the lower half of `p` (`= p[0:127] = p_low`).
6. Lines 55-56: do the same with `p_low` as done in step 3 with `p_high`.
7. Lines 57-76: compute `barrett_param` and `barrett_param2` using `-KYBER_Q`.
8. Lines 78-81: combine the lower and upper half of `barrett_param` and `barrett_param2`, because it was split in step 2 and 4.
9. Line 82-84: increment `x8` (the loop counter) with the vector length and continue with the loop if applicable.

3.2.3 Optimizing the assembly code

The code generated by the compiler was already optimized because it was vectorized. But we can optimize it further, because in this version we work with 16-bit instead of 32-bit elements while doing the Barrett reduce, which the compiler did not optimize by itself. The optimized code can be seen in Appendix A.6.1, where U means the upper half of an array and L means the lower half.

Lines 13-31 initialize the variables, lines 33-70 represent the for-loop we take into account for this level.

Lines 13-31 are the same as lines 13-32 in Appendix A.4 except for a couple of lines. These include the `orr x,xzr,y` operations, with `x` and `xzr` as the source registers and `y` the destination register. These are changed into `mov x,y` because `x OR xzr` is the same as `mov x,y` but it is more in the clear what happens here when using `mov`.

Furthermore, `z4` was initialized in Appendix A.4 as `z4.s`, but we now do 16 bit operations on `z4`, thus we initialize it as `z4.h`.

The loop in Appendix A.6.1 does the following:

1. Lines 33-51: the same as lines 33-52 in Appendix A.4.
2. Line 52:

52	<code>uzp1 z6.h, z6.h, z5.h</code>
----	-----------------------------------------

The operations after line 48 do not need to operate on 32 bits anymore, so we can return to vector registers having 16 bits elements. This is done by concatenating the lower and upper halves of `t` with instruction `uzp1`.

In instruction `uzp1 z6.h,z6.h,z5.h` we use the size specifier `.h` because we split each 32 bit element of `z6` and `z5` in two 16 bit elements. One containing an element of `t` and the other containing zero's. We now have `z5` and `z6` in the form of how it is represented in the two upper bars in the figure below. Then executing `uzp1` takes each `t` element out of both source vector registers and puts them in `z6`.

z6.h	t[0]	000	t[1]	000	...		t[127]	000
z5.h	t[128]	000	t[129]	000	...		t[255]	000
z6.h	t[0]	t[1]	...	t[127]	t[128]	t[129]	...	t[255]

3. Line 54: the same as step 5 in section 3.2.2. We do not need to extend the elements of `p_low` to 32 bit like in Appendix A.4, because the operations will not let the elements overflow as stated in the previous item.
4. Lines 54-66: compute `barrett_param` and `barrett_param2` and store them respectively in `p[0:127]` and `p[128:255]`.
5. Lines 68-70: the same as step 9 in section 3.2.2

3.3 Optimizing the sixth NTT level

3.3.1 The idea

This level differs from the seventh level, because it must use the second nested for-loop in Listing 3.1. We now iterate twice through this for-loop, thus we have two packets of butterfly operations.

The first packet of butterfly operations includes pairwise operations for every $p[j]$ and $p[j+64]$ for $j \in [0, 63]$. In Figure 3.4 we can see what happens in the first (black lines) and last operation (green lines).

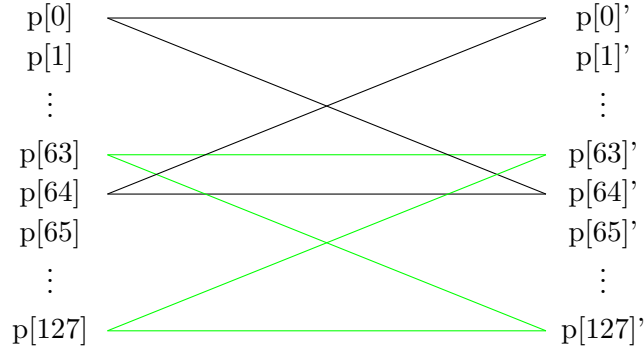


Figure 3.4: The first and last operation of the first packet of butterfly operations

The operations in the second packet happen pairwise for every $p[j]$ and $p[j+64]$ for $j \in [64, 191]$. In Figure 3.5 we can see what happens in the first packet (explained in the paragraph above), and in the first (purple lines) and last operation (yellow lines) of this second packet. We can also see what parts are p_low and what parts are p_high .

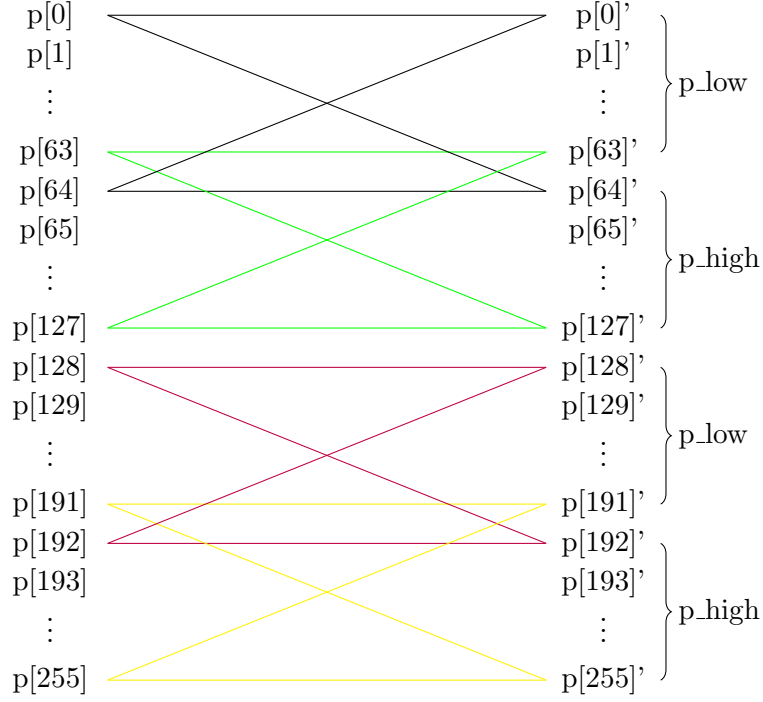


Figure 3.5: The first and last operations of the two packets of butterfly operations

3.3.2 The route to assembly

To optimize this level we could put the level in a separate function and look at the assembly code like in the previous level. But when we do this, the compiler does not use predicated instructions, but normal vector registers. In order to optimize this level we use the optimized assembly code of the previous level and modify it.

Because the sixth level does two packets of butterfly operations, it actually runs the assembly code of the seventh level twice. As we can see in Appendix A.5 the loop of the seventh level is copied and pasted twice in this level.

Besides copying we also change a couple of lines:

- In the first loop, `p_high` is `p[64:127]` and `p_low` is `p[0:63]`.
- After the first loop we prepare a couple of variables for the second loop:
 - Re-initialize `zeta`

```

72      add    x15, x15, w5, lsl #1    //x10 =
      zetas[zeta_counter]
73      ld1rh  {z2.s}, p0/z, [x15]    //broadcast x10 to z2
      = zeta

```

– Increment **k** by 1

```

74      add    w5, w5, #1    //w5 = w5 + 1 = zeta_counter

```

– Set the loop counter (**x8**) to 0 again

```

75      mov    x8, xzr    //x8 = 0 (32 bits)

```

- In the second loop, **p_high** is **p[192:255]** and **p_low** is **p[128:191]**.
- At the end of the program we store the new value of **k** in RAM.

```

119     str    w5, [x1]    //x1 = w5 = zeta_counter to RAM

```

3.4 The levels 7 until 0

As we can see, there is a pattern across the levels and in the C code (Listing 3.1), the butterfly operations occur at a narrower range for a higher level.

The whole seventh level happens in one packet of butterfly operations. With the first operation being **p[0] * p[128]**, as seen in Figure 3.3.

The sixth level happens in two packets of butterfly operations. The first operations for the packets can be seen in Figure 3.5 and they are:

- **p[0] * p[64]**
- **p[128] * p[192]**

The next level, the fifth one, has four packets of butterfly operations with the first operations for each packet being:

- **p[0] * p[32]**
- **p[64] * p[96]**
- **p[128] * p[160]**
- **p[192] * p[224]**

This is also graphically displayed in Figure 3.6.

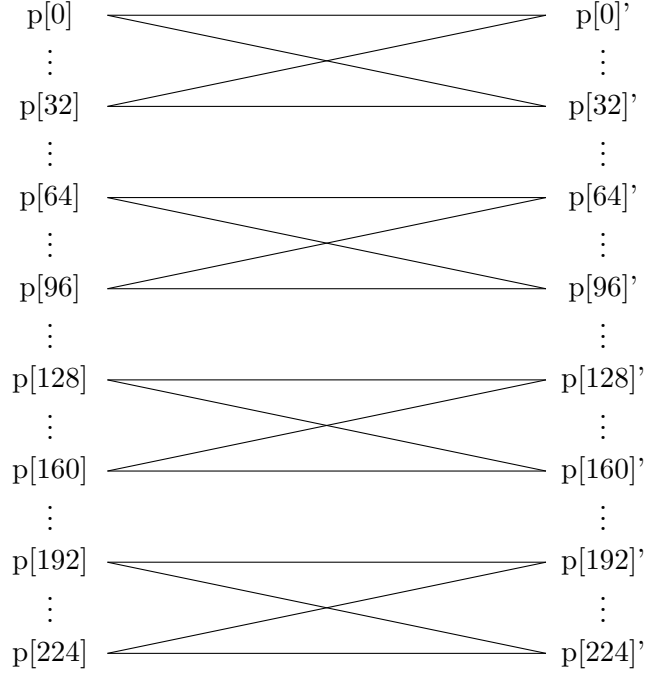


Figure 3.6: The first operation of each butterfly packet in level 5

The fourth level has eight packets of butterfly operations. The first operations of each packet are:

- $p[0] * p[16]$
- $p[32] * p[48]$
- $p[64] * p[80]$
- $p[96] * p[112]$
- $p[128] * p[144]$
- $p[160] * p[176]$
- $p[192] * p[208]$
- $p[224] * p[240]$

We can see the pattern; the number of packets is doubled by each level. And the addresses for the very first butterfly operation for a level is computed by $p[0] * p[1 \ll \text{level}]$.

For the first operations of the remaining packets we increase the addresses of p with $2(1 \ll \text{level})$ for each level. Thus for the second packet it would be $p[0 + 2(1 \ll \text{level})] * p[(1 \ll \text{level}) + 2(1 \ll \text{level})]$. With this in mind we can compute the first operations of each packet of butterfly operations for the remaining levels.

3.4.1 A code framework for the levels 6 until 0

Now we see the pattern, the two loops of level six in Appendix A.5 can be rewritten to one loop and then we can reuse the code for the following levels after that.

The rewritten code for level six is represented in Appendix A.6.2 and the main idea is: do one packet of butterfly operations in the loop `.loop`. The number of packets is maintained by the outer loop, `.loops`.

To visualize what happens in this level, we can look back at Figure 3.5. The same procedure takes place, but now the first packet of butterfly operations (black and green lines) happen in the first iteration of `.loops`, and the second packet (purple and yellow lines) happen in the second iteration of `.loops`.

Furthermore it has the following code fragments worth noticeable:

1. Set `1 << level`, the number of butterfly operations per packet:

```
24      mov    w9, #64                //w9 = 64 = level_shift
```

2. Set the number of butterfly operations packets:

```
27      mov    w16, #2                //.loops condition
```

3. Set `1 << level`, the starting address of `p_high` for one packet:

```
29      mov    x10, #64
```

4. Set 0, the starting address of `p_low`

```
30      mov    x12, xzr
```

5. Increment `x10` and `x12` for the next (second) packet of butterfly operations

```
80      add    x10, x10, #128
81      add    x12, x12, #128
```

With a couple of those values changed for each level, we can reuse the code in Appendix A.6.2 for the levels 5 up to and including 0. For example for level 5, which code can be found in Appendix A.6.3:

1. Set `1 << level`, the number of butterfly operations per packet:

```
24      mov    w9, #32                //w9 = 32 = level_shift
```

2. Set the number of butterfly operations packets:

```
27      mov    w16, #4                //.loops condition
```

3. Set `1 << level`, the starting address of `p_high` for one packet:

```
29      mov    x10, #32
```

4. Increment `x10` and `x12` for the next packet of butterfly operations

```
80      add    x10, x10, #64
81      add    x12, x12, #64
```

The code for level 4 to 0 can be found in Appendix A.6.

3.5 Problems concerning vector lengths?

The optimized version of level 7 would run perfectly with all vector lengths. If the vector length is 2048 and we have 16 bit integers in array `p`, we can store $2048/16 = 128$ elements per vector register. This means we can load `p_low` in one vector register and `p_high` in one vector register. Then `.loop` just runs one time. The code will also run for smaller vector lengths, then `.loop` would run more times.

The optimized level 6 would run perfectly with vector length 1024. We can then store $1024/16 = 64$ elements in one vector register. Thus `p_low` could be stored in one vector register and this also applies to `p_high`. But we would think level 6 will not run with vector length 2048, because we want to load in fewer elements than the vector length. However, this is handled by the governing predicate `p1`. It only uses the part of the vector register needed. Thus for vector length 2048, only the half of it would be used. This means using vector length 2048 for level 6 does not actually use all of the vector register.

This problem occurs for every level below the seventh level. To optimize this, we can interleave elements in a vector register in order to use the space in the register more efficiently. Interleaving means rearranging the elements in a vector register. For instance we want to interleave elements in level 6 of the NTT for vector length 2048. We rearrange the elements such that the vector register is fully filled with elements. To create the new `p_high` for the first iteration we take `p_high` of the first packet of butterfly operations (`p[64:127]`) and `p_high` of the second packet `p[192:255]`, and we put them behind each other in one register:

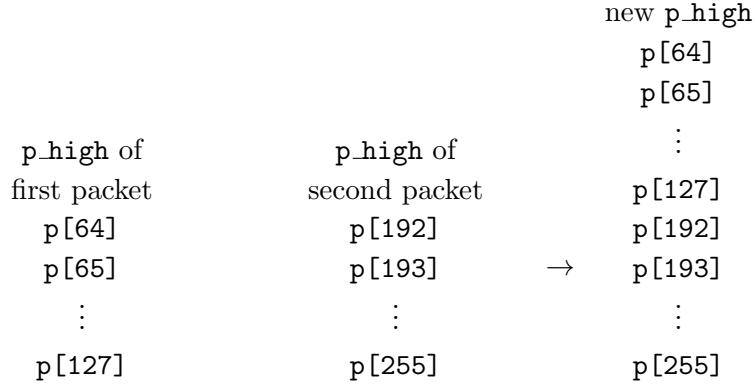


Figure 3.7: Create the new **p_high**

We also create a new **p_low** by putting the **p_low** of the first packet of butterfly operations (**p**[0:63]) and the **p_low** of the second packet (**p**[128:191]) in one register:

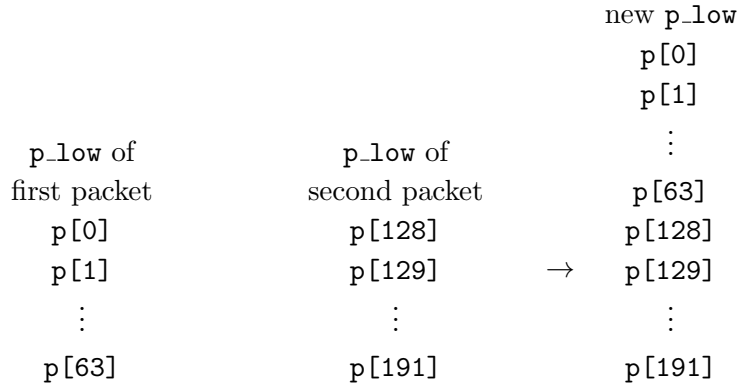


Figure 3.8: Create the new **p_low**

Now that we created the new **p_high** and **p_low** we can go on with the rest of the operations. As mentioned before in every packet of butterfly operations **p_high** multiplies with an element of the array **zetas**. For this level we multiply **zetas**[2] with the the lower half of the new **p_high**, and **zetas**[3] with the upper half of the new **p_high**. In order to do this we fill a vector register with **zetas**[2] in the lower half and **zetas**[3] in the upper half. We can then use the filled vector to multiply with the new **p_high** in one operation.

new p_high		zetas
p[64]		zetas[2]
p[65]		zetas[2]
⋮		⋮
p[127]	×	zetas[2]
p[192]		zetas[3]
p[193]		zetas[3]
⋮		⋮
p[255]		zetas[3]

Figure 3.9: Multiply the new **p_high** with **zetas**

After doing the rest of the operations we have to revert **p** to the old format by taking the upper and lower half of new **p_high** and put them back in **p_high** of the first packet and **p_high** of the second packet. We will also do this for the new **p_low**.

We interleave for the following levels when:

- Level 7: never interleave
- Level 6: interleave if vector length > 1024
- Level 5: interleave if vector length > 512
- Level 4: interleave if vector length > 256
- Level 3: interleave if vector length > 128
- Level 2: interleave if vector length > 64
- Level 1: interleave if vector length > 32
- Level 0: interleave if vector length > 16

So if we want to make it more space efficient, we always have to interleave in level 2,1 and 0, because the minimum vector length in SVE is 128. In the following sections we will go through the interleaved levels for each vector length. The code for each level can be found in Appendix A.7.

3.6 Interleave for vector length 128

3.6.1 Level 0

In level 0 a packet of butterfly operations consist of one butterfly operation. Which means a butterfly operation happens on a pairwise level, thus every even numbered element of **p** is a **p_low** for a packet and every odd numbered element is **p_high** for a packet. We can make full use of the vector length by putting the **p_low** elements of every butterfly operation in one vector register and all the **p_high** elements in another register. The vector length is 128,

meaning we can store $128/16 = 8$ elements in one vector register. Thus we can store eight elements of `p_high` in one register and eight elements of `p_low` in one register. Therefore we load the first (`p1`) and second (`p2`) eight elements of `p` and interleave them to fill the new `p_low` and `p_high`:

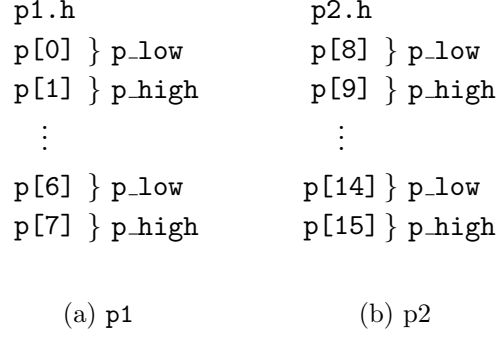


Figure 3.10: Load in the first and second eight elements

Then we create the new `p_low` and new `p_high`:

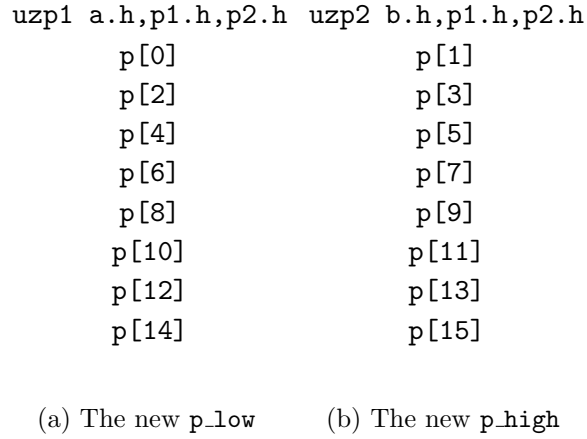


Figure 3.11: The new `p_low` and `p_high`

In this level we need `zetas[128:255]`, where each butterfly operation needs one element of `zetas`. Thus for the first iteration we load in `zetas[128:135]` to multiply with `p_high`:

new p_high		zetas
p[1]		zetas[128]
p[3]		zetas[129]
p[5]		zetas[130]
p[7]	×	zetas[131]
p[9]		zetas[132]
p[11]		zetas[133]
p[13]		zetas[134]
p[15]		zetas[135]

Figure 3.12: Multiply the new **p_high** with **zetas**

After finishing the rest of the operations we need to revert **p** to the old format:

zip1 c.h,a.h,b.h	zip2 d.h,a.h,b.h
p[0]	p[8]
p[1]	p[9]
⋮	⋮
p[6]	p[14]
p[7]	p[15]

(a) Original format of **p1** (b) Original format of **p2**

Figure 3.13: The original format of **p1** and **p2**

We will do this way of interleaving for every iteration. In each iteration we load 16 elements, thus we need to iterate $256/16=16$ times to complete a whole array.

3.6.2 Level 1

The idea of interleaving is the same as for level 2; we put elements of **p_low** in a vector register and the elements of **p_high** in one register.

In this level a packet of butterfly operations consists of two butterfly operations, causing **p_low** and **p_high** being:

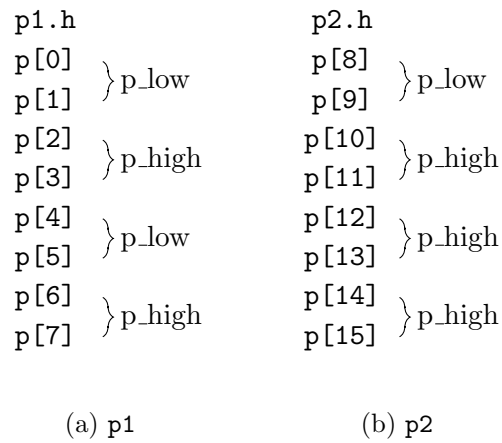
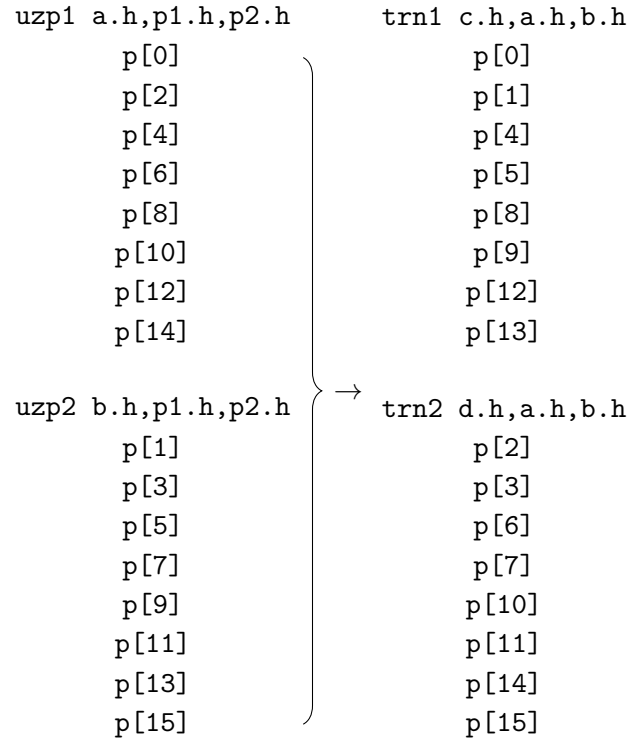


Figure 3.14: Load in the first and second eight elements

Then we get the even and odd elements of **p1** (a) and **p2** (b) in two different registers and then create the new **p_low** (c) and new **p_high** (d):



(a) Even and odd elements

(b) The new `p_low` and new `p_high`

Figure 3.15: Create the new `p_low` and new `p_high`

For the first iteration we need `zetas[64:67]`, which we will get by loading in `zetas[64:71]` and getting the first half of it:

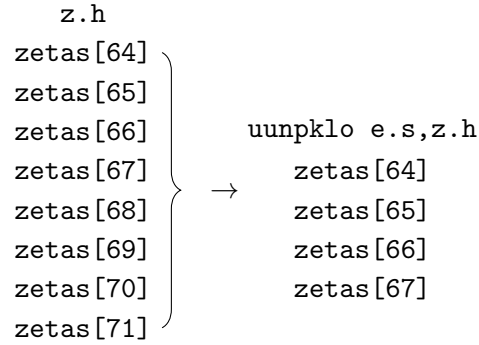
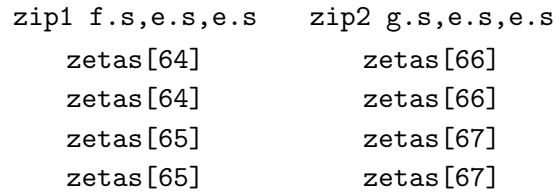


Figure 3.16: Get the lower half of `z`

Later on the new `p_high` will be split in half and zero extended. We will call the first half `p_highL` and the other half `p_highU`. `zetas` must be in the same format as those two registers in order to be multiplied with them:



(a) `zetas` for `p_highL` (b) `zetas` for `p_highU`

Figure 3.17: Create `zetas` for `p_highL` and `p_highU`

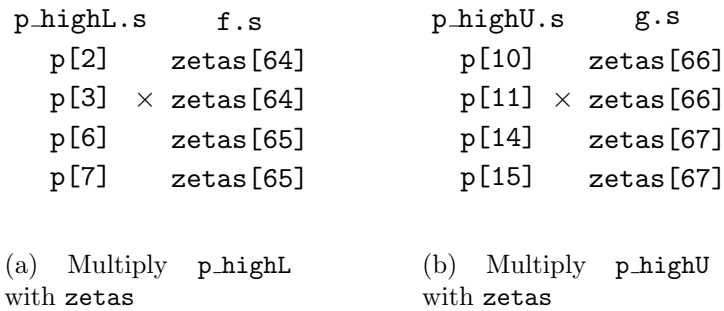


Figure 3.18: Multiply `p_high` with `zetas`

Then we return to the old format of `p` after finishing the rest of the

operations by returning to the odd and even elements of **p1** (**h**) and **p2** (**i**) and then the original format of **p1** (**j**) and **p2** (**k**):

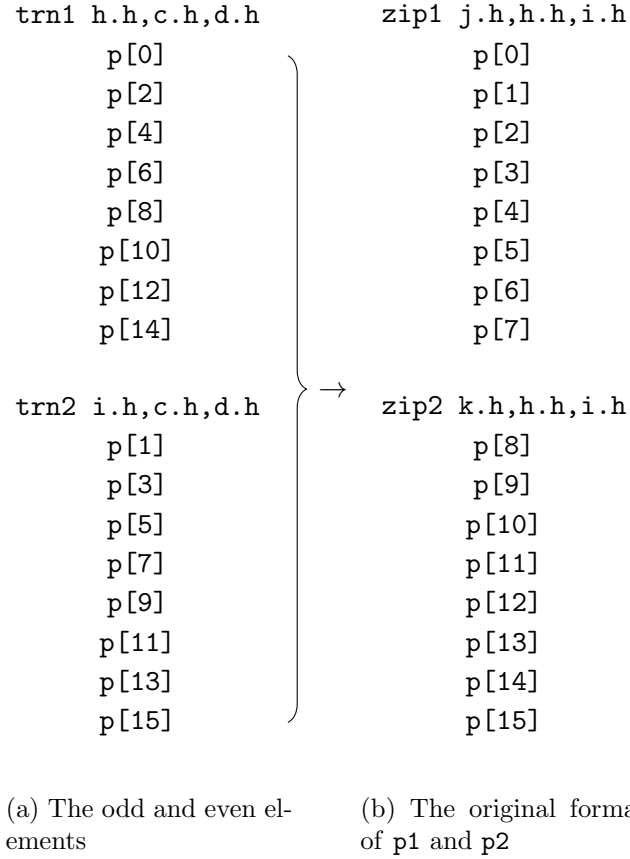


Figure 3.19: Return to the original format of **p1** and **p2**

3.6.3 Level 2

For this level the idea is the same as for level 0 and 1.

A packet of butterfly operations consists of four butterfly operations, so **p_high** and **p_low** are:

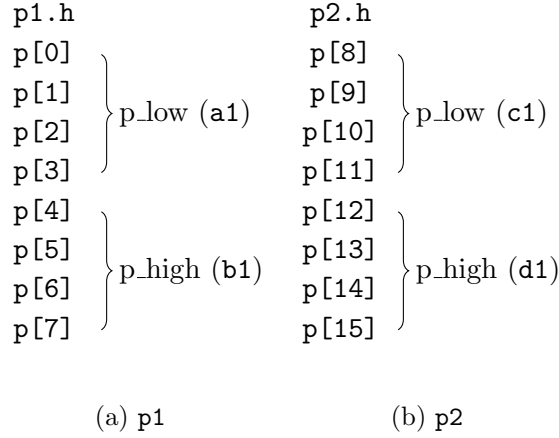


Figure 3.20: Load in the first and second eight elements

Then we interleave by putting the `p_low`s (`a1` and `c1`) in one register. A `p_low` or a `p_high` in this level consists of 4 elements, which is $4 \cdot 16 = 64$ bits. This means we can use the `.d` specifier to move chunks of 64 bits in a vector. So we can get the new `p_low` in one instruction.

For the `p_high`s (`b1` and `d1`) we put them in two separate registers with 32-bit elements (they need to be 32 bits later on):

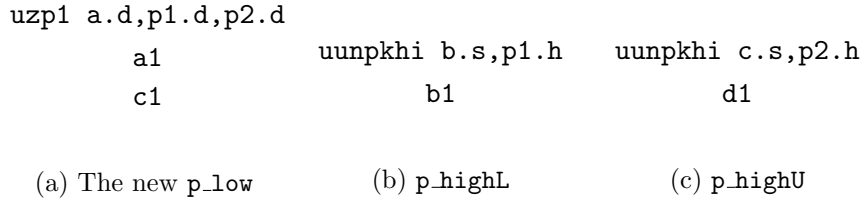


Figure 3.21: Make the new `p_low` and `p_highL` and `p_highU`

In the first iteration we need `zetas[32:33]`. Because `p_highL` and `[p_highU]` are 32-bit we can load in `zetas` in 32 bit elements immediately:

ldrh d.s	ldrh e.s
zetas[32]	zetas[33]
zetas[32]	zetas[33]
zetas[32]	zetas[33]
zetas[32]	zetas[33]

(a) zetas for p_highL (b) zetas for p_highU

Figure 3.22: Create zetas for p_highL and p_highU

Then we can multiply zetas with p_highL and p_highU:

p_highL.s	d.s	p_highU.s	e.s
p[4]	zetas[32]	p[12]	zetas[33]
p[5] ×	zetas[32]	p[13] ×	zetas[33]
p[6]	zetas[32]	p[14]	zetas[33]
p[7]	zetas[32]	p[15]	zetas[33]

(a) Multiply p_highL
with zetas

(b) Multiply p_highU
with zetas

Figure 3.23: Multiply p_high with zetas

Later on the two halves (p_highL and p_highU) will be merged into the new p_high. We will call this new p_high, f in the following operations.

After all the operations we need to get back to the old format of p1 and p2:

uzp1 g.d,a.d,f.d	uzp1 g.d,a.d,f.d
a1	c1
b1	d1

(a) The original format of p1 (b) The original format of p2

Figure 3.24: Get back to the original format of p1 and p2

3.7 Interleaving for vector length 256

3.7.1 Level 0

Level 0 for vector length 256 uses the same method as for vector length 128, because in this level we can still interleave by taking the odd and even elements in order to make the new `p_low` and new `p_high`; it is not affected by the vector length. The difference is that we now work with 16 elements in a vector register instead of 8 elements. Therefore `p1`, `p2`, the new `p_low` and the new `p_high` contain 16 elements. For vector length 256 we have to modify the code for vector length 128 so that it loads in 16 elements for `p1` and 16 for `p2`.

3.7.2 Level 1

The method for level 1 also stays the same, as interleaving by taking the odd and even elements and thereafter the `trn1` and `trn2` of those elements is not affected by the vector length. Like for level 0, we have to modify the code such that it loads in 16 elements for `p1` and 16 for `p2` instead of 8 elements.

3.7.3 Level 2

Creating the new `p_low` can be done in the same way as in level 2 for vector length 128, because interleaving by taking chunks of 64-bit elements is not affected by the vector length. But we cannot create `p_highL` and `p_highU` directly from `p1` and `p2` as we did for vector length 256, thus we use the same method for `p_low` for `p_high`:

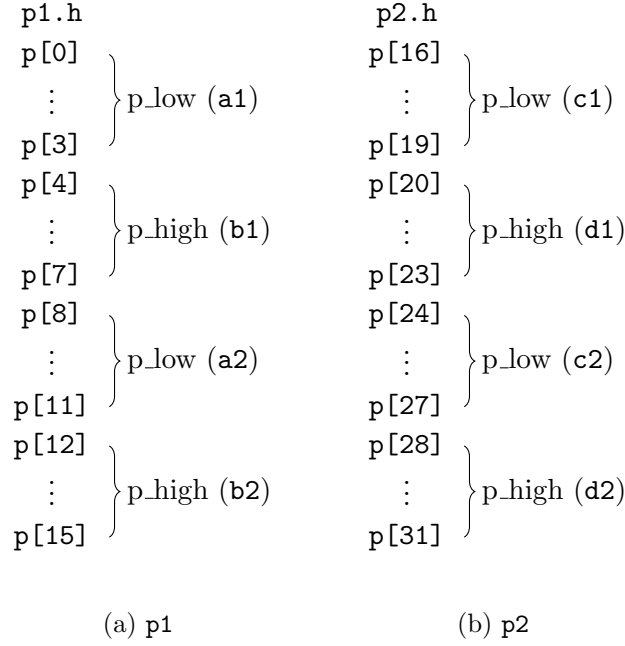


Figure 3.25: Load in the first and second 16 elements

Then we get the `p_low`s of `p1` (`a1` and `a2`) and `p2` (`c1` and `c2`) to create the new `p_low` (`a`) and we do the same for the `p_high`s to create the new `p_high` (`b`):

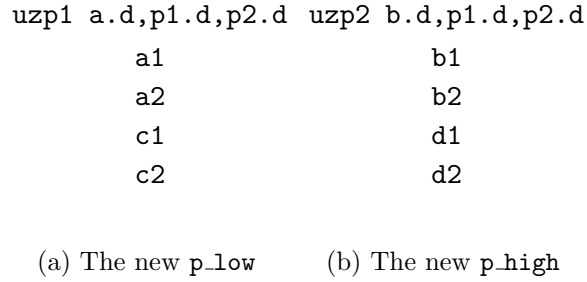


Figure 3.26: Create the new `p_low` and new `p_high`

To get `zetas` in the right format we also use the same method as in level 1 for vector length 128. Now we need `zetas[32:35]` for the first iteration, `zetas[32:33]` for `p_highL` and `zetas[34:35]` for `p_highU`. This can be done by first getting the half of the loaded in `zetas` it and zipping until we get the right format:

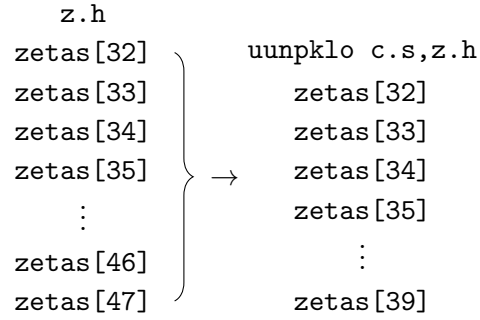
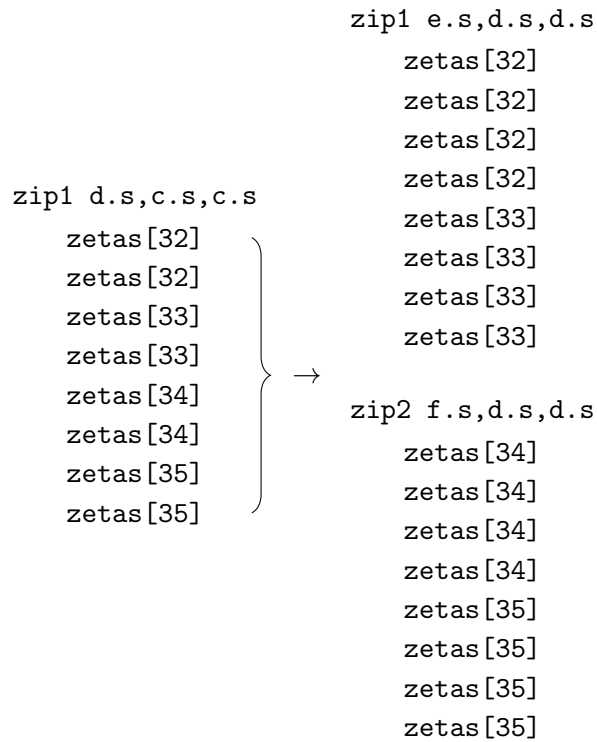


Figure 3.27: Get the lower half of z

And then we zip until we have the right format of **zetas** to be multiplied with **p_highL** (e) and **p_highU** (f):



(a) The first zip

(b) The right format of **zetas**

Figure 3.28: The right format of **zetas** for **p_highL** and **p_highU**

To go back to the original format of **p1** (g) and **p2** (h) we do:

zip1 g.d,a.d,b.d	zip2 h.d,a.d,b.d
a1	c1
b1	d2
a2	c1
b2	d2

(a) The original format of p1	(b) The original format of p2
-----------------------------------------	-----------------------------------------

Figure 3.29: Return to the original format of **p1** and **p2**

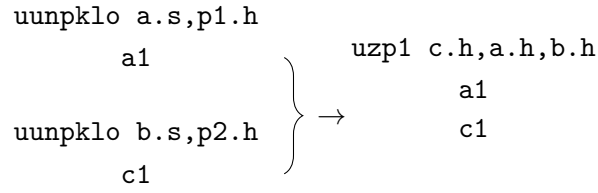
3.7.4 Level 3

In level 3 **p_low** and **p_high** are the halves of a vector length, which means we can merge the lower halves of **p1** (a) and **p2** (b) to get the new **p_low** (c). For **p_highL** (d) we take the upper half of **p1**, and for **p_highU** (e) the upper half of **p2**:

<p>p1.h</p> <p>p[0] </p> <p style="text-align: center;">⋮</p> <p>p[7] </p> <p>p[8] </p> <p style="text-align: center;">⋮</p> <p>p[15] </p>	<p>p2.h</p> <p>p[16] </p> <p style="text-align: center;">⋮</p> <p>p[23] </p> <p>p[24] </p> <p style="text-align: center;">⋮</p> <p>p[31] </p>
<p style="font-size: 2em;">}</p> <p style="text-align: center;">p_low (a1)</p>	<p style="font-size: 2em;">}</p> <p style="text-align: center;">p_low (c1)</p>
<p style="font-size: 2em;">}</p> <p style="text-align: center;">p_high (b1)</p>	<p style="font-size: 2em;">}</p> <p style="text-align: center;">p_high (d1)</p>

(a) **p1**
(b) **p2**

Figure 3.30: Load in the first and second 16 elements



(a) The lower halves of p1 and p2 (b) The new p_low

Figure 3.31: Create the new p_low

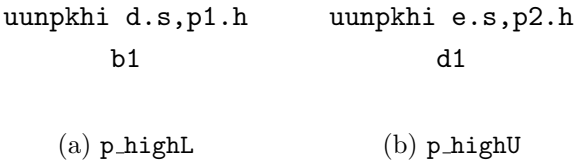


Figure 3.32: Create p_highL and p_highU

For this level we load in **zetas** the same way as for level 2 for vector length 128; we load in **zetas** for **p_highL** immediately and this also holds for **zetas** for **p_highU**.

We go back to the original format of **p1** (**k**) and **p2** (**l**) by taking the halves (**g,h,i,j**) and merging them again. Later in the program **p_highL** and **p_highU** are merged in one register, let it be register **f**.

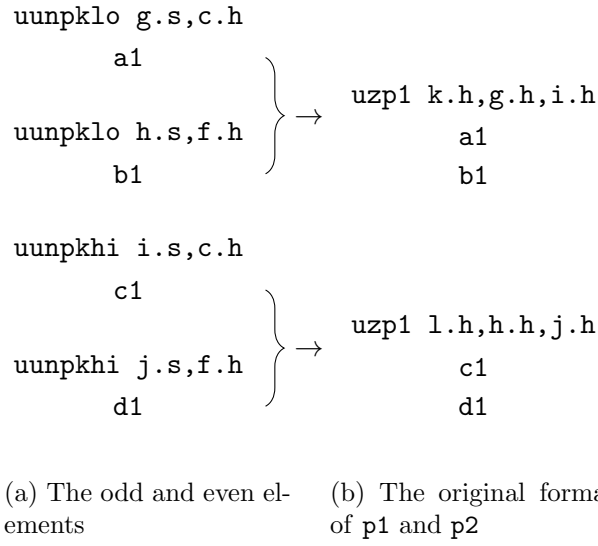


Figure 3.33: Return to the original format of p1 and p2

3.8 Interleaving for vector length 512

3.8.1 Level 0, 1 and 2

Level 0, 1 and 2 are done in the same way as for vector length 256, because the methods used are not affected by the vector length. We do have to change some constants so that we load in 32 elements instead of 16 elements for p1 and p2.

3.8.2 Level 3

For level 3 we also make use of the `.d` specifier; a `p_low` or `p_high` consists of 8 elements, which means it is $8 \cdot 16 = 128$ bits long, so using this specifier we would work with halves of the elements. It would be more convenient to work with the `.q` specifier, but we would then need the operation `uzp1` and `uzp2` which do not support `.q`.

We load in `p1` and `p2` and get the lower halves of the `p_lows` and `p_highs` in two separate registers (`a` and `b`); in which we indicate the lower part of a `p_low` or `p_high` with a ‘L’ and the upper part with a ‘U’. Then we shuffle them to the new `p_low` (`c`) and new `p_high` (`d`):

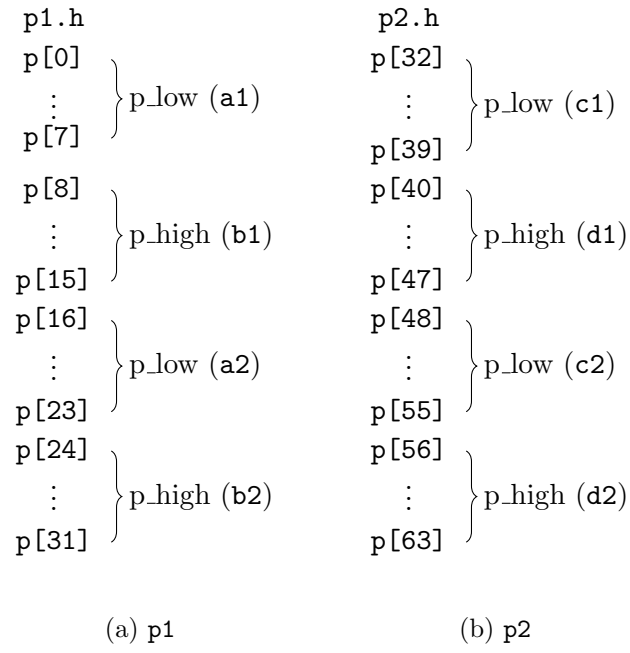
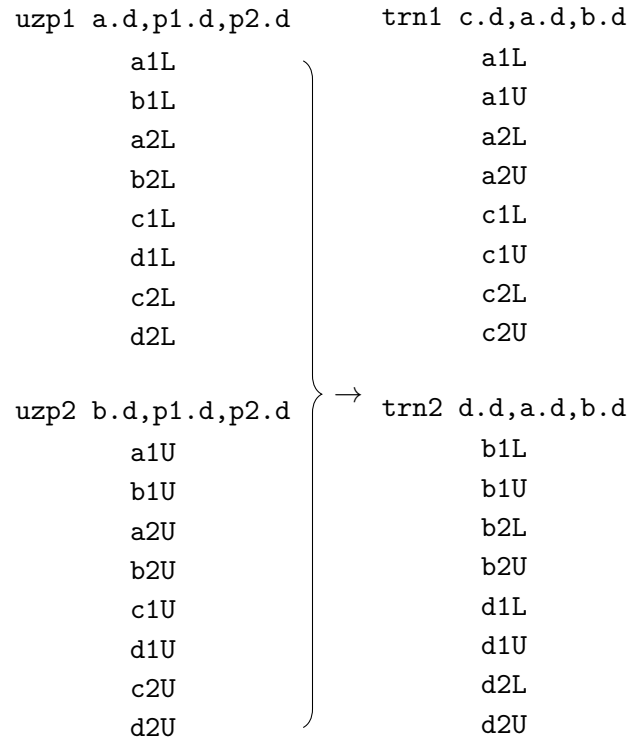


Figure 3.34: Load in the first and second 32 elements



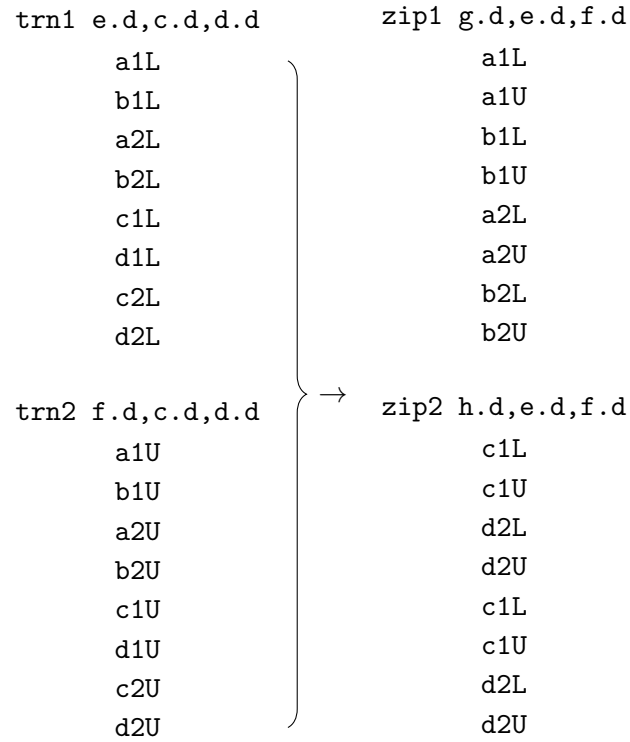
(a) The lower and upper halves of the `p_lows` and `p_highs`

(b) The new `p_low` and new `p_high`

Figure 3.35: Create the new `p_low` and new `p_high`

We load in `zetas` the same way as we did in level 2 for vector length 256. We now do an extra `zip1` after the first `zip1`.

To go back at the original format of `p1` (g) and `p2` (h) we first get the lower halves of the `p_lows` and `p_highs` in a register (e) and the upper halves in a register (f):



(a) The lower and upper halves of the p_low and p_high

(b) The original format of p1 and p2

Figure 3.36: Go back to the original format of p1 and p2

3.8.3 Level 4

Level 4 is the same as level 3 for vector length 256, because a p_low or p_high is again the half of the vector length.

3.9 Interleaving for vector length 1024

3.9.1 Level 0, 1, 2 and 3

Those levels are done in the same way as for vector length 512, because the way it has been done is not affected by the vector length. But we also do not have to forget to change some constants so p1 and p2 consist of 64 elements instead of 32 elements.

3.9.2 Level 4

For level 4 we take the lower and upper parts of `p1` and `p2` to work with them:

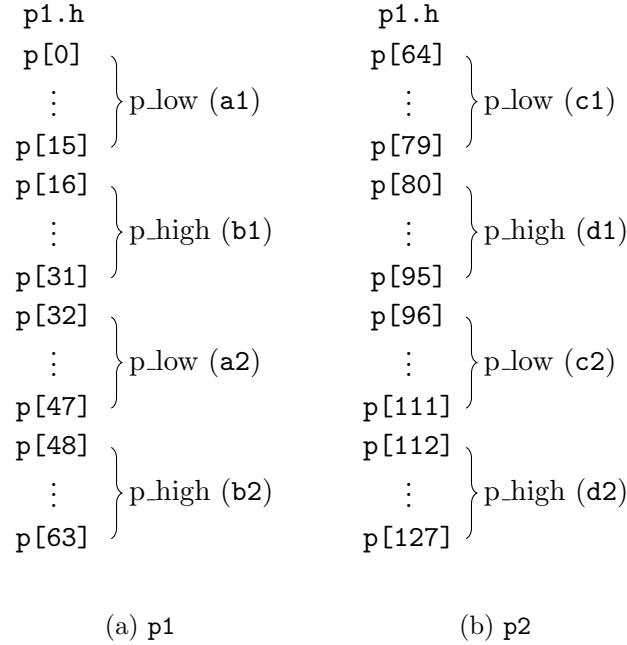


Figure 3.37: Load in the first and second 64 elements

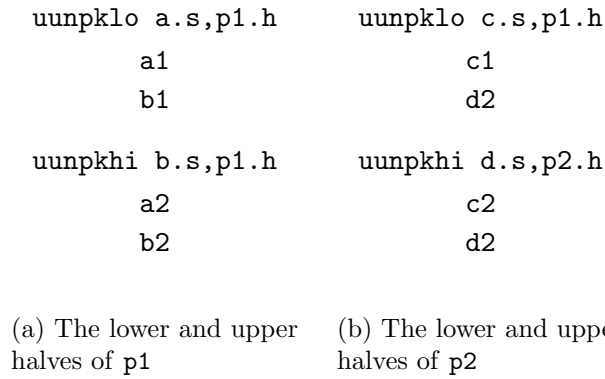
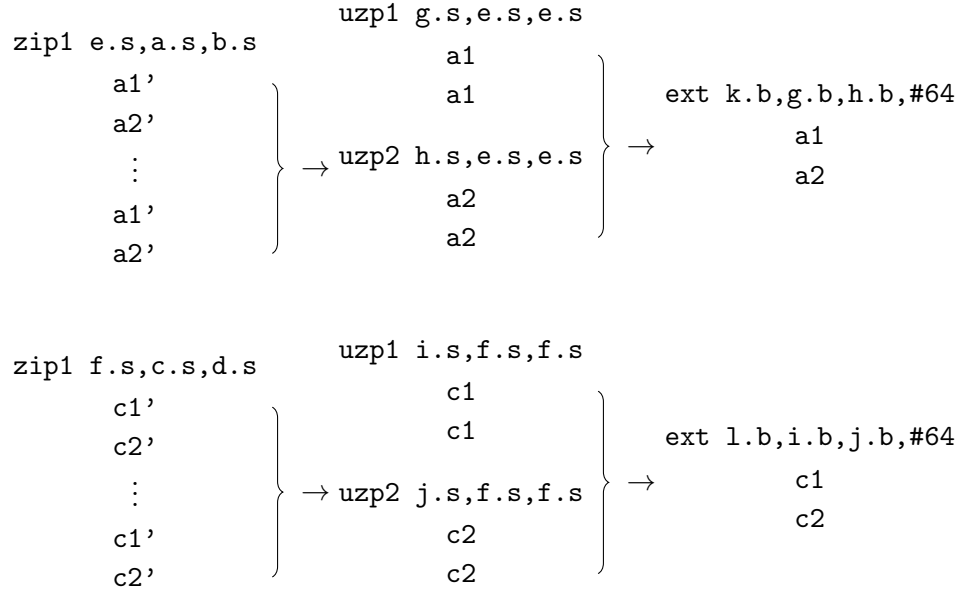


Figure 3.38: Getting the lower and upper halves of `p1` and `p2`

Then we `zip` the elements, so the elements of the halves become intertwined. We indicate an intertwined `p_low` or `p_high` by a apostrophe, for instance `zip1 e.s,a.s,b.s`, gives us `{p[0], p[32], p[1], p[33], ...}`,

$p[15], p[47]\}$, which we will indicate by $\{a1', a2', \dots a1', a2'\}$.

Then we shuffle the zipped elements to get a the lower and upper halves of the new p_low :



(a) The zip operation (b) Shuffle the zipped elements (c) Halves of the new p_low

Figure 3.39: Return to the original format of $p1$ and $p2$

We will also do this to get the lower and upper halves of the new p_high :

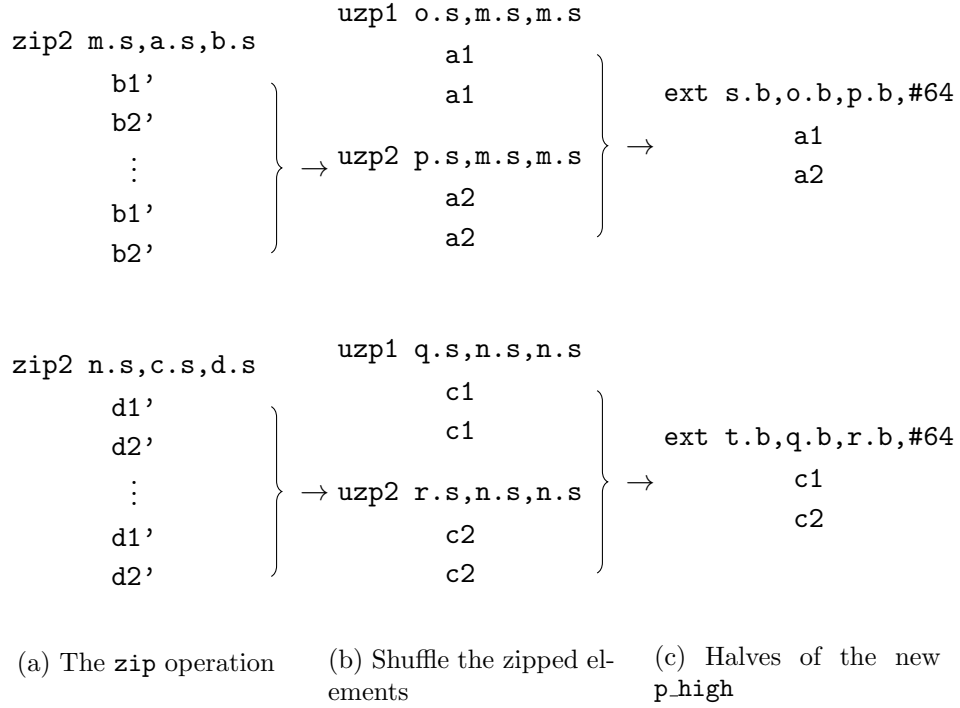


Figure 3.40: Return to the original format of p1 and p2

Now we can make the new p_low and new p_high with the created halves:

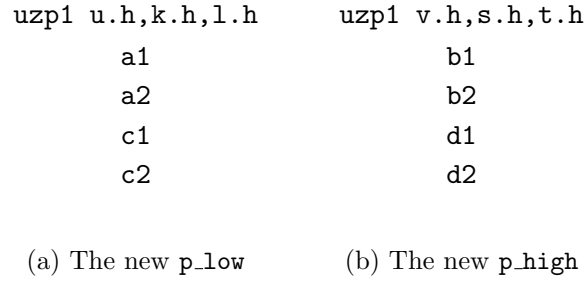
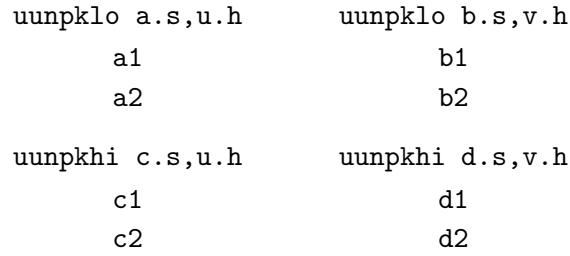


Figure 3.41: Create p_highL and p_highU

Loading in **zetas** happens in the same way as for level 2 for vector length 256. But now we do three **zip1s** instead of one at the beginning.

To get back to the original format of p1 and p2 we get the lower and upper halves of the new p_low and new p_high, zip them and shuffle them. It is actually the same method as making the new p_low and new p_high, but now the input is different:



(a) The lower and upper halves of the new `p_low` (b) The lower and upper halves of the new `p_high`

Figure 3.42: Getting the lower and upper halves of `p1` and `p2`

3.9.3 Level 5

Level 5 is the same as level 4 for vector length 512, because a `p_low` or `p_high` is again the half of the vector length. We also have to change the constants so that they load in 64 elements instead of 32 elements.

3.10 Interleaving for vector length 2048

3.10.1 Level 0, 1, 2 and 3

Also for vector length 2048 level 0, 1, 2 and 3 use the same method. We now load in 128 elements instead of 64 elements for `p1` and `p2`.

3.10.2 Level 4

For level 4 we will take the upper and lower halves of `p1` and `p2`:

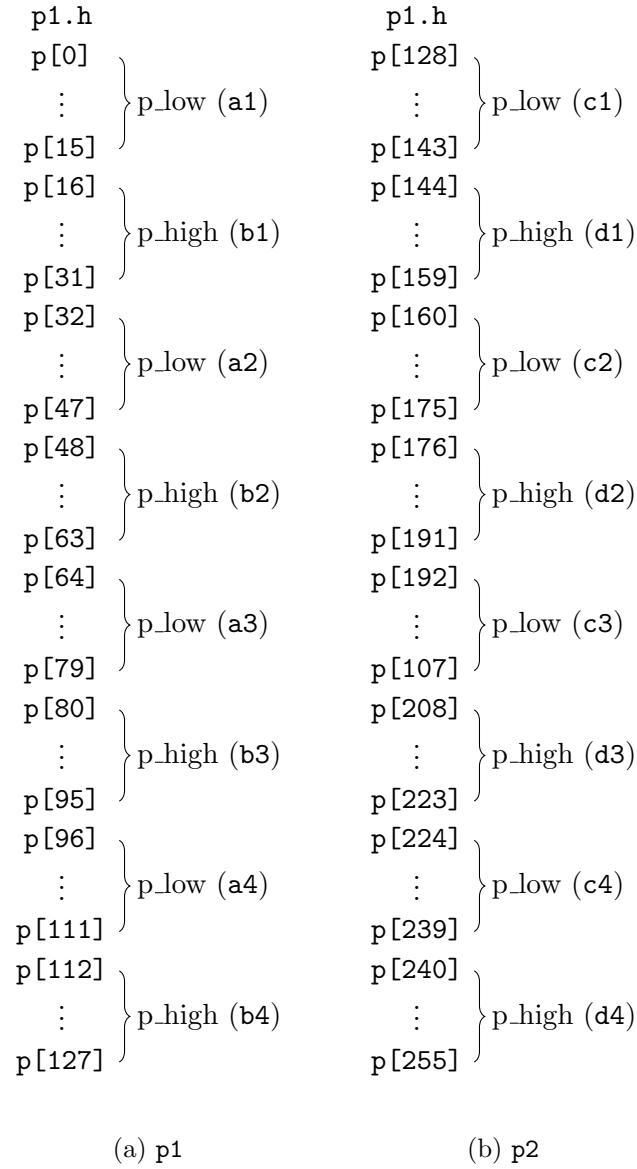


Figure 3.43: Load in the first and second 128 elements

uunpklo a.s,p1.h	uunpklo c.s,p2.h
a1	c1
b1	d1
a2	c2
b2	d2
uunpkhi b.s,p1.h	uunpkhi d.s,p2.h
a3	c3
b3	d3
a4	c4
b4	d4

(a) The lower and upper halves of p1 (b) The lower and upper halves of p2

Figure 3.44: Getting the lower and upper halves of p1 and p2

We can than **zip** the halves so they become intertwined:

zip1 e.s,a.s,b.s	zip1 g.s,c.s,d.s
a1'	c1'
a3'	c3'
a1'	c1'
a3'	c3'
⋮	⋮
b1'	d1'
b3'	d3'
b1'	d1'
b3'	d3'
zip2 f.s,a.s,b.s	zip2 h.s,c.s,d.s
a2'	c2'
a4'	c4'
a2'	c2'
a4'	c4'
⋮	⋮
b2'	d2'
b4'	d4'
b2'	d2'
b4'	d4'

(a) The zip operations (b) More zip operations

Figure 3.45: Getting the zipped parts

Then we shuffle the zipped parts to get the lower and upper halves of the new `p_low`:

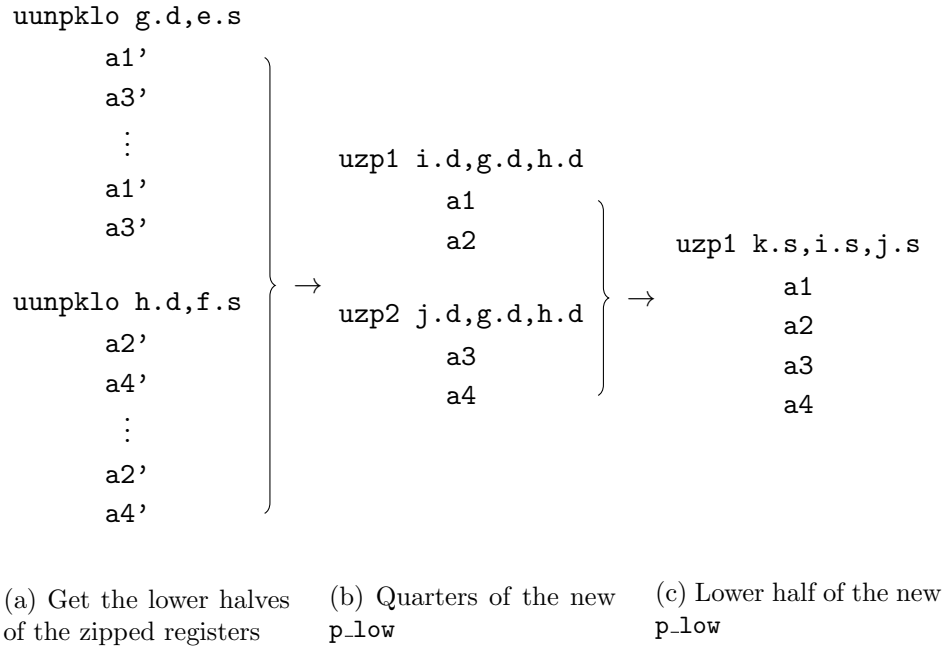


Figure 3.46: Create the lower half of the new `p_low`

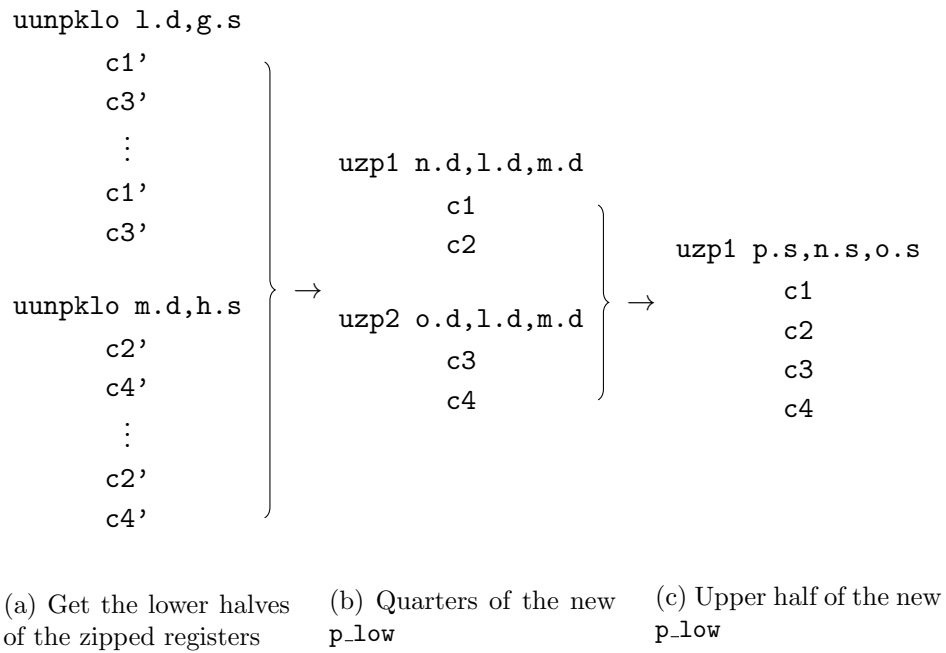


Figure 3.47: Create the upper half of the new `p_low`

Then we get the new `p_low` (`q`) by merging `k` and `p`. The new `p_high` (`r`) is created in the same way, but then we take the upper halves instead of lower halves of the zipped registers:

uzp1 q.h,k.h,p.h	r
a1	b1
a2	b2
a3	b3
a4	b4
c1	d1
c2	d2
c3	d3
c4	d4

(a) The new `p_low` (b) The new `p_high`

Figure 3.48: Create the new `p_low` and new `p_high`

Getting `zetas` in the format we want is done in the same way as for level 4 for vector length 1024.

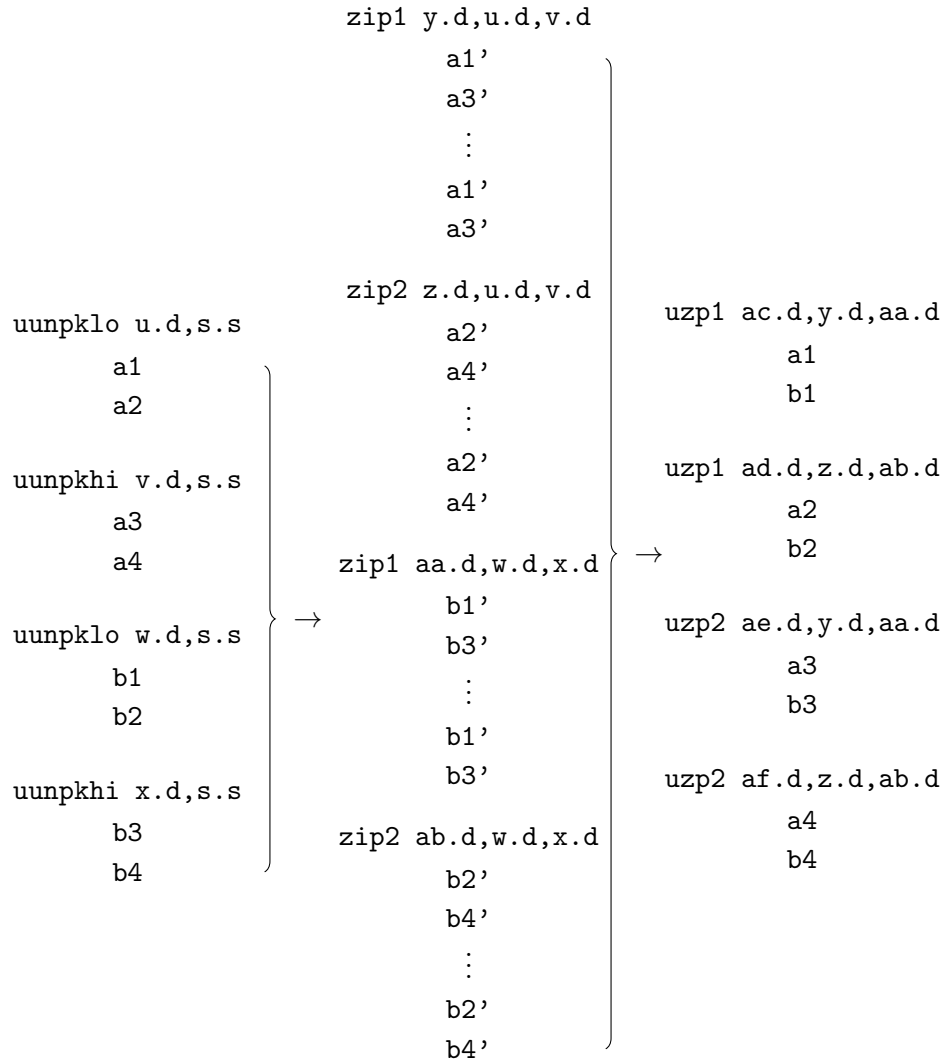
To go back to the old format of `p1` and `p2` we first take the lower halves of the new `p_low` and new `p_high`:

uunpklo s.s,q.h	uunpklo t.s,r.h
a1	b1
a2	b2
a3	b3
a4	b4

(a) The lower half of the new `p_low` (b) The lower half of the new `p_high`

Figure 3.49: Get the lower halves of the new `p_low` and new `p_high`

Then we can go back to the old format of `p1`:



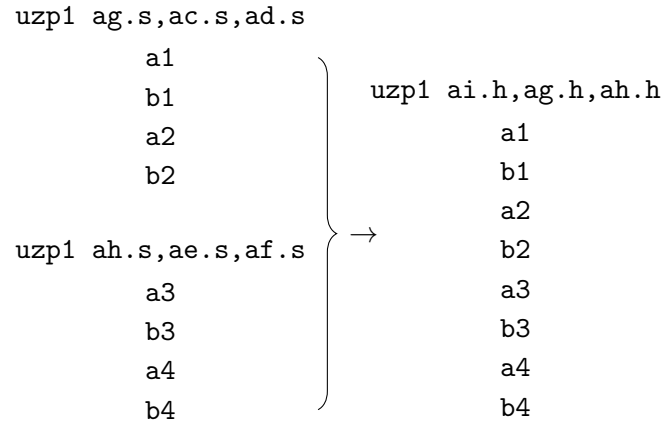
(a) The first two quarters
of the new `p_low` and of
the new `p_high`

(b) `zip` those elements

(c) All the quarters of
the original format of `p1`

Figure 3.50: Create the quarters of the original format of `p1`

Finally we merge those quarters to get back to the original format of `p1`:



(a) Get the lower and upper half of the original format of `p1`

(b) The original format of `p1`

Figure 3.51: The right format of `zetas` for `p_highL` and `p_highU`

To get back to the original format of `p2` we can do it in the same way.

3.10.3 Level 5

The code for this level is the same as for level 4 for vector length 1024. The difference is that we load in 128 elements for `p1` and `p2` and that we do four `zips` to get `zetas` in the right format.

3.10.4 Level 6

Level 6 is the same as level 5 for vector length 1024, because a `p_low` or `p_high` is again the half of the vector length. Here we also have to load in a different amount of elements compared to level 5 for vector length 1024.

Chapter 4

Results

The interleaved version should work more efficient because it makes full use of the vector length, which means that the loop iterates less times. In order to check if it really is more efficient, we have to benchmark the code by counting clock cycles. However, we do not have the right equipment for this because hardware running SVE does not exist yet. The best option to measure the code is then to count the number of instructions executed because this relates to the number of clock cycles.

We can count the number of instructions by grouping the instructions by their category:

Group	Instructions
Load/Store	ldrsw, ld1h, ld1rh, str, st1h
Move	mov, movprfx, uzp1, uzp2, uunpklo, uunpkhi, zip1, zip2, trn1, trn2, ext
Predicate	ptrue, b.mi, whilelo, inch
Operations	add, adrp, sub, mul, and, mad, mla, lsr

Table 4.1: Instructions grouped by category

4.1 Counting instructions

We then have the following number of instructions for the whole NTT for each vector length, where the non-interleaved code can be found in Appendix A.6 and the interleaved version in Appendix A.7:

- Vector length 2048:

Group	Nr. of instructions
Load/Store	1291
Move	1630
Predicate	1543
Operations	6898
Total	11362

(a) Non-interleaved

Group	Nr. of instructions
Load/Store	43
Move	302
Predicate	48
Operations	209
Total	602

(b) Interleaved

- Vector length 1024:

Group	Nr. of instructions
Load/Store	1295
Move	1635
Predicate	1546
Operations	6920
Total	11396

(a) Non-interleaved

Group	Nr. of instructions
Load/Store	85
Move	377
Predicate	84
Operations	406
Total	952

(b) Interleaved

- Vector length 512:

Group	Nr. of instructions
Load/Store	1311
Move	1655
Predicate	1558
Operations	7008
Total	11532

(a) Non-interleaved

Group	Nr. of instructions
Load/Store	165
Move	462
Predicate	154
Operations	796
Total	1577

(b) Interleaved

- Vector length 256:

Group	Nr. of instructions
Load/Store	1359
Move	1715
Predicate	1594
Operations	7272
Total	11940

(a) Non-interleaved

Group	Nr. of instructions
Load/Store	319
Move	727
Predicate	290
Operations	1568
Total	2904

(b) Interleaved

- Vector length 128:

Group	Nr. of instructions
Load/Store	1487
Move	1875
Predicate	1690
Operations	7976
Total	13028

(a) Non-interleaved

Group	Nr. of instructions
Load/Store	617
Move	1088
Predicate	554
Operations	3096
Total	5355

(b) Interleaved

4.2 Summary

Summarizing the results we get the following table:

Vector length	Non-interleaved	Interleaved
2048	11362	602
1024	11396	952
512	11532	1577
256	11940	2904
128	13028	5355

Table 4.2: Number of instructions

Chapter 5

Related Work

This research is not the first one to optimize the NTT function. It has been optimized multiple times for different architectures.

Alkim, Jakubeit and Schwabe also optimized the NTT in NewHope, which is a post-quantum key exchange protocol created by Alkim, Ducas, Pöppelmann and Schwabe [12], on ARM Cortex-M [21].

Pöppelmann, Oder and Güneysu optimized the NTT on 8-bit ATxmega microcontrollers [22]. Liu, Seo, Roy, Großschädl, Kim and Verbauwhede also optimized the NTT, on 8-bit AVR processors [23].

There were also researches like this thesis optimizing the NTT using vector instructions. Güneysu, Oder, Pöppelmann and Schwabe optimized the NTT on Intel's Sandy Bridge and Ivy Bridge microarchitectures using floats [24]. Streit and De Santis did this on NEON for NewHope [25]. Also Seiler optimized the NTT function [26], but on an Intel instruction set architecture, namely AVX2.

The NTT of ten levels for polynomial length 1024 was optimized on ARMv8-A. The implementation of Streit and De Santis outperforms the C reference implementation of NewHope by 8.3 times. The optimization measures included three alternatives of the reduce functions and full vectorization of all ring operations [25]. In their implementation they have:

1. Merged levels: the first four levels are merged and the last six levels are merged.
2. Loaded in the elements in level 0 interleaved.
3. Interleaved the elements in level 1, by using the transpose function `trn`.

The operations addition, subtraction and multiplication were vectorized and interleaved in order to make them more efficient.

Seiler optimized the NTT for AVX2 (256-bit registers) with polynomial length 256 using integers. In this implementation extra reduction functions are called in the third and sixth level. In level 4 up and including 7 (or with our definition, level 3, 2, 1 and 0) Seiler interleaved elements, because

that is more efficient. In the original implementation a single precomputed root is loaded in and then broadcasted and shuffled. This implementation loads in precomputed vectors of roots so that it can be loaded directly in a vector register. This optimized NTT version of Seiler is faster than the original one in Kyber; it is used in the submitted version of Kyber to the NIST post-quantum cryptography standardization process [26].

Chapter 6

Conclusions

For this research we optimized the NTT function as it is implemented for Kyber on ARMv8-A SVE. The ARMv8-A SVE's key feature is that vectors are scalable. Previous work also optimized the NTT on different vector architectures, namely on ARMv8-A and AVX.

In an attempt to optimize the NTT we vectorized code and interleaved elements in assembly. We then compared the number of instructions for the non-interleaved and interleaved code for all vector lengths. The number of instructions for the interleaved version were significant lower than for the non-interleaved version for each vector length. For the non-interleaved version the number of instructions fluctuated around 12000 instructions for each vector length, but for the interleaved version it was clearly visible that the number of instructions depended on the vector length.

We cannot state for sure that this version is really optimized, because we do not have the equipment to measure it precisely. But we do have counted the number of instructions, which resulted in the interleaved version having significant less instructions than the non-interleaved version. So we can assume that this version of the NTT in Kyber is optimized for ARMv8-A SVE. Further work could be benchmarking the code with the right equipment to check if this version of the NTT is indeed faster.

Bibliography

- [1] “Post-Quantum cryptography.” <https://csrc.nist.gov/Projects/Post-Quantum-Cryptography>, 29-5-2018 (accessed 30-5-2018).
- [2] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS-Kyber: algorithm specification and supporting documentation..” Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. <https://cryptojedi.org/papers/kybernist-20171130.pdf>.
- [3] N. Stephens, “Technology update: The scalable vector extension (SVE) for the Armv8-A architecture.” <https://community.arm.com/processors/b/blog/posts/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture>, 22-08-2016 (accessed 17-02-2018).
- [4] “Installing Arm Instruction Emulator.” <https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator/installing-arm-instruction-emulator>, (accessed 23-05-2018).
- [5] M. Fürer, “Faster integer multiplication,” *SIAM Journal On Computing*, vol. 39, pp. 979–1005, 2009. <https://ivv5hpp.uni-muenster.de/u/cl/WS2007-8/mult.pdf>.
- [6] E. W. Weisstein, “Number theoretic transform.” <http://mathworld.wolfram.com/NumberTheoreticTransform.html>, (accessed 18-02-2018).
- [7] M. Fürer, “Faster Integer Multiplication.” <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.3775&rep=rep1&type=pdf>, 2007.
- [8] J. W. Cooley, P. A. W. Lewis, Peter, and D. Welch, “Historical notes on the fast fourier transform,” *IEEE Trans. Audio Electroacoust*, 1967. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.467.7209&rep=rep1&type=pdf>.

- [9] J. W. Cooley and J. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series,” 1965. <https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf>.
- [10] P. Jakubeit, “Newhope for ARM,” Master’s thesis, Radboud University Nijmegen, 2016.
- [11] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS – kyber: a CCA-secure module-lattice-based KEM,” in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, (London, United Kingdom), IEEE, April 2018. <https://cryptojedi.org/papers/kyber-20180226.pdf>.
- [12] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, “Post-quantum key exchange – a new hope,” in *Proceedings of the 25th USENIX Security Symposium*, USENIX Association, 2016. Document ID: 0462d84a3d34b12b75e8f5e4ca032869, <http://cryptojedi.org/papers/#newhope>.
- [13] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, pp. 44(170):419–521, 1985. <http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf>.
- [14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied Cryptography*. CRC Press, 1996. <http://cacr.uwaterloo.ca/hac/>.
- [15] P. Barrett, *Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor*. Lecture Notes in Computer Science, 2000. https://link.springer.com/chapter/10.1007/3-540-47721-7_24.
- [16] ARM, *ARM® Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for ARMv8-A*, 2017. https://static.docs.arm.com/ddi0584/a/DDI0584A.b_SVE_supplement_armv8A.pdf.
- [17] “Getting Started.” <https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator/get-started>, (accessed 23-05-2018).
- [18] “Registers in AArch64 state.” <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0801b/BABBGCAC.html>, 2014 (accessed 29-03-18).

- [19] ARM, *ARMv8 Instruction Set Overview*, 2011. <https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.ReferenceManual.pdf>.
- [20] W. Gentleman and G. Sande, “Fast fourier transforms - for fun and profit,” in *Fall Joint Computer Conference*, vol. 29, pp. 563–578, 1966. <https://www.computer.org/csdl/proceedings/afips/1966/5068/00/50680563.pdf>.
- [21] E. Alkim, P. Jakubeit, and P. Schwabe, “A new hope on arm cortex-m,” in *Security, Privacy, and Advanced Cryptography Engineering* (C. Carlet, A. Hasan, and V. Saraswat, eds.), vol. 10076 of *Lecture Notes in Computer Science*, pp. 332–349, Springer-Verlag Berlin Heidelberg, 2016. Document ID: c7a82d41d39c535fd09ca1b032ebca1b, <http://cryptojedi.org/papers/#newhopearm>.
- [22] Z. Liu, T. Pöppelmann, T. Oder, H. Seo, S. S. Roy, T. Güneysu, J. Großschädl, H. Kim, and I. Verbauwhede, “High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers,” *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 4, pp. 117:1–117:24, 2017. <https://eprint.iacr.org/2015/382.pdf>.
- [23] Z. Liu, H. Seo, S. S. Roy, H. Kim, and I. Verbauwhede, “Efficient RingLWE encryption on 8-bit AVR processors,” in *Handschuh (Eds.), CHES 2015, Vol. 9293 of LNCS*, pp. 663–682, Springer, 2015. <https://eprint.iacr.org/2015/410.pdf>.
- [24] T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe, “Software speed records for lattice-based signatures,” in *Post-Quantum Cryptography* (P. Gaborit, ed.), vol. 7932 of *Lecture Notes in Computer Science*, pp. 67–82, Springer-Verlag Berlin Heidelberg, 2013. Document ID: d67aa537a6de60813845a45505c313, <http://cryptojedi.org/papers/#lattisigns>.
- [25] S. Streit and F. D. Santis, “Post-Quantum Key Exchange on ARMv8-A – A New Hope for NEON made Simple.” Cryptology ePrint Archive, Report 2017/388, 2017. <https://eprint.iacr.org/2017/388>.
- [26] G. Seiler, “Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography.” Cryptology ePrint Archive, Report 2018/039, 2018. <https://eprint.iacr.org/2018/039>.
- [27] ““module” load command does not work.” <https://askubuntu.com/questions/343692/module-load-command-does-not-work/343721>, 10-9-2013 (accessed 24-5-2018).
- [28] “Download and install Arm Compilers and Libraries,” (accessed 24-5-2018).

Appendix A

Appendix

A.1 How to set up the compiler and emulator

This setup was done on an ODROID-C2 with Ubuntu Mate 16.04.4. The ARM compiler for HPC had version 18.0 and the ARM instruction emulator had version 1.2.1.

Following the tutorial on the website [17] and a thread on Stack Overflow [27] we had to do the next steps:

1. Downloading and installing the compiler [28]:
 - (a) Download the package Arm Compiler for HPC at <https://silver.arm.com/browse>. For this we need an ARM account, which can be easily set up for free.
 - (b) Extract the package and run the shell script as root.
 - (c) We then set the environment modules. We first need to install environment-modules. For Ubuntu we need to configure the environment-modules by running `add.modules`. Then we comment the second last and comment the last line in `./bashrc` [27]:

```
#module() { eval '/usr/Modules/$MODULE.VERSION/bin/modulecmd $modules_shell $*'; }  
module() { eval '/usr/bin/modulecmd $modules_shell $*'; }
```

2. Downloading and installing the emulator [4]:
 - (a) Download the emulator at <https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator/download>
 - (b) We can then extract the package and run the shell script as root.
3. Setting the modules: Now we can set the environment modules with the following commands:

```
export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/  
module load Generic-AArch64/Ubuntu/16.04/arm-hpc-compiler/18.0  
module load Generic-AArch64/Ubuntu/14.04/arm-instruction-emulator/1.2.1
```

We can change 18.0 and 1.2.1 to the version we are using, and the version of the OS can also be changed. The available modules can be found with the command `module avail` after running the `export` command in the code box above.

A.2 How to compile and run code

Before we can run any code we need to load the modules as described at step 3 in Appendix A.1. To compile an `example.c` file with optimization level 3 and run it with vector length 256 we execute the following commands:

```
armclang -O3 -march=armv8-a+sve -o example example.c  
armie -msve-vector-bits=256 ./example
```

We can change the optimization level to 2, 1 or 0. To compile from C to assembly we run:

```
armclang -O3 -march=armv8-a+sve -S -o example.s example.c
```

A.3 Level 7 written out in C

https://github.com/LittleberryPi/bachelor-thesis-code/blob/master/Level7_written_out.c

A.4 Level 7 compiled with O3 in assembly

https://github.com/LittleberryPi/bachelor-thesis-code/blob/master/Level7_O3.s

A.5 Level 6 with 2 loops

https://github.com/LittleberryPi/bachelor-thesis-code/blob/master/Level6_2loops.s

A.6 Code framework version

<https://github.com/LittleberryPi/bachelor-thesis-code/tree/master/framework>

A.6.1 Level 7

<https://github.com/LittleberryPi/bachelor-thesis-code/blob/master/framework/function7.s>

A.6.2 Level 6

<https://github.com/LittleberryPi/bachelor-thesis-code/blob/master/framework/function6.s>

A.6.3 Level 5

<https://github.com/LittleberryPi/bachelor-thesis-code/blob/master/framework/function5.s>

A.7 Interleaved version

<https://github.com/LittleberryPi/bachelor-thesis-code/tree/master/interleaved>