UNIVERSITÀ DI PISA

Department of Information Engineering
Bachelor's Degree in Computer Engineering

Vectorizing 4-bit Arithmetic Operations Using 64-bit Registers on the x86-64 Architecture

Advisor:
Prof. Marco Cococcioni

Student:
Lorenzo Grassi

May 28, 2025

Abstract

Modern processors often support SIMD[1] instructions, which allow the execution of the same operation on multiple data elements with just one CPU instruction. This approach is especially effective in scenarios involving large, uniform data transformations, such as image processing—where, for example, increasing the brightness of an image means adding a fixed value to every pixel (or sub-pixel, if we are talking about a color image) in an array. By reducing instruction overhead and enabling parallel computation across multiple ALUs (or FPUs), SIMD offers significant performance gains.

These instructions rely on specialized registers and hardware, and predominantly work with the standard 8-bit, 16-bit, 32-bit, and 64-bit data types, packed into vector registers that are typically 128, 256, or 512 bits wide.[2]

AI models—especially large language models like ChatGPT or Gemini—perform massive numbers of calculations involving billions of parameters. To reduce memory usage and speed up computations, these models are increasingly using lower-precision formats. Tiny data types can also be useful when storing large images with a very narrow bit depth. However, these formats are not natively supported by most modern processors.

The goal of this thesis is to simulate SIMD instructions for 4-bit integers using 64-bit general-purpose registers and operations for Intel-AMD x86-64 processors to try and speed up these low precision calculations through software-based vectorization.

Benchmarks show that the proposed 4-bit vectorized algorithms achieve a significant speedup over scalar code in some operations such as matrix multiplication.

[1] Single Instruction Multiple Data.

[2] Or scalable in the case of ARM SVE/SVE2.

Contents

1 Introduction **5**

2 Addition and Subtraction **9**

2.1 Addition 9
2.2 The Double XOR Method for Addition 10
2.3 Subtraction 13
2.4 The Double XOR Method for Subtraction 13
2.5 Recursive Double-XOR-Based Solution for Addition and Subtraction 14
2.6 Iterative Double-XOR-Based Solution for Addition and Subtraction 17
2.7 The High-Low Nibble Method for Addition 20
2.8 The High-Low Nibble Method for Subtraction 21
2.9 Addition with the High Bit XOR Method 23
2.10 Subtraction with the High Bit XOR Method 24
2.11 Further Implementations 25

3 Saturating Addition and Subtraction **27**

3.1 The Concept of Saturating Arithmetic 27
3.2 Saturating Addition with the Double XOR Method 27
3.3 Saturating Subtraction with the Double XOR Method 29
3.4 Saturating Addition with a Hybrid Solution 30
3.5 Saturating Subtraction with a Hybrid Solution 33
3.6 Saturating Addition with the High-Low Nibble Method 35
3.7 Saturating Subtraction with the High-Low Nibble Method 38
3.8 Saturating Addition with the High Bit XOR Method 39
3.9 Saturating Subtraction with the High Bit XOR Method 40

4 Multiplication and Saturating Multiplication **43**

4.1 Multiplication 43
4.2 Multiplication with the High-Low Nibble Method 44
4.3 Multiplication with the High Bit XOR Method 48

4.4 Saturating Multiplication with the High-Low Nibble Method 49

5 Vectorized Operations with a Lookup Table 53
5.1 The Lookup Table Approach 53
5.2 Lookup Table Implementation with 8-bit Entries 54
5.3 Lookup Table Implementation with 4-bit Entries 55

6 Performance Analysis and Comparisons 57
6.1 Performance of Vectorized Algorithms 57
6.1.1 Addition 58
6.1.2 Saturating Addition 58
6.1.3 Subtraction 59
6.1.4 Saturating Subtraction 59
6.1.5 Multiplication 60
6.1.6 Saturating Multiplication 60
6.2 Comparisons with the Lookup Table Approach 61
6.3 Comparisons with Non-Vectorized Operations 62

7 Vectorized Composite Linear Algebra Operations 65
7.1 Dot Product 65
7.2 Multiply-Accumulate by Scalar 67
7.3 Saturating Multiply-Accumulate by Scalar 68
7.4 Matrix Multiplication and Saturating Matrix Multiplication 69

8 Performance of Linear Algebra Operations 75
8.1 Dot Product 75
8.2 Multiply-Accumulate and Saturating Multiply-Accumulate 76
8.3 Matrix Multiplication 77

9 Conclusions 79
9.1 Key Takeaways 79
9.2 Future Works 79

A Helper and Compute Functions 81
A.1 Helper Functions 81
A.2 Compute Functions to Check the Results 82

B Benchmark Functions 85
B.1 Benchmark Functions for Basic Operations 85
B.2 Benchmark Functions for Composite Operations 96

Chapter 1

Introduction

Intel x86-64 processors feature sixteen 64-bit general-purpose software registers. This means that if we interpret every individual 4-bit section as an independent value, each register can effectively hold a vector of sixteen 4-bit variables. With two of these vectors loaded into registers, we can write procedures that perform operations between each corresponding pair of values in those registers by using a combination of arithmetic and logic instructions that operate across the entire register width. The underlying idea behind this is data parallelism: although we use general-purpose instructions, they process sixteen data elements simultaneously. This approach of performing parallel operations on data within a processor register is commonly referred to as SWAR[1]. Our hope is that this software-based vectorization can yield some speed gains since processors take roughly the same amount of time to execute most instructions, no matter how large the operands are—especially memory reads and writes.

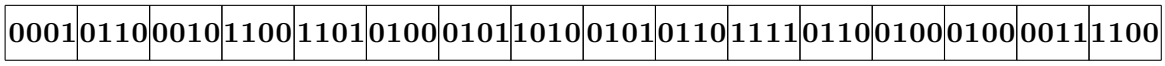

Figure 1.1: A 64-bit register interpreted as sixteen separate 4-bit values.

We chose C++ as our primary programming language for this project because it offers both low-level control over the hardware and high-level features like classes. For simplicity, we are starting off by using unsigned integers as our values, and we have adopted naming conventions similar to those used in NEON SIMD libraries (though we are working on x86-64).

Our first step was to define the `uint4x16_t` class. This class has a single 64-bit data field that stores our sixteen unsigned 4-bit integers. Given that instances of this class are

[1] SIMD within a register.

only 8 bytes in size, the compiler[2] will pass them to functions in a register[3]. This allows for direct manipulation of our vectors without needing to access main memory. For testing purposes, we also added a convenient way to access individual 4-bit sections using a helper class called `uint4_t`, which serves only as a temporary container to hold the information needed to access each nibble[4]. Since it is not meant to be used outside of `uint4x16_t`, its constructor is private.

```
class uint4x16_t
{
public:
    uint64_t reg;
    uint4x16_t(uint64_t _reg)
    {
        reg = _reg;
    }
    uint4x16_t() {}
    uint4_t operator [] (int i)
    {
        return uint4_t(((uint8_t*)&reg) + (i / 2), i % 2);
    }
};
```

Listing 1.1: Definition of `uint4x16_t`.

```
class uint4_t
{
    uint8_t* addr;
    bool upper;
    uint4_t(uint8_t* _addr, bool _upper)
    {
        upper = _upper;
        addr = _addr;
    }
public:
    uint4_t& operator = (uint8_t b)
    {
        if(upper)
        {
            *addr = (*addr & 0x0F) | (b << 4);
            return *this;
```

[2]GCC 11.4.0 on a Linux system.

[3]In particular, using the System V calling convention, `%rdi` and `%rsi` will be used.

[4]A nibble refers to four consecutive bits, essentially the 4-bit equivalent of a byte.

```
17          }
18          *addr = (*addr & 0xF0) | (b & 0x0F);
19          return *this;
20      }
21      operator uint8_t() const
22      {
23          if(upper)
24              return (*addr) >> 4;
25          return (*addr) & 0x0F;
26      }
27      friend std::ostream& operator << (std::ostream& o, const uint4_t& n)
28      {
29          o << +uint8_t(n);
30          return o;
31      }
32      friend class uint4x16_t;
33  };
```

Listing 1.2: Definition of `uint4_t`.

The initial basic operations we are considering will be implemented as functions that take two `uint4x16_t` arguments and return another `uint4x16_t`.[5] More complex operations, that we will present at the end of this thesis, will deviate from this standard. Bitwise operations are trivial in our setup—they are inherently parallel and work independently on each bit—so we will not cover them here.

Now that we have laid the groundwork, we are ready to dive into the operations.

[5]Which, using the System V calling convention, will be returned in the `%RAX` register.

Chapter 2

Addition and Subtraction

2.1 Addition

We have two registers, each storing sixteen unsigned integers—one per nibble. Our goal is to compute the sum of each of the sixteen pairs of values. We can easily see that just adding the two input registers together does not work. The issue is that if one of the 4-bit sums overflows, the carry bit spills into the next nibble to the left. That messes up the result and breaks the idea of treating each nibble as its own 4-bit number. What we get is not completely useless, though; all we would need to make it work is a way to correct for those overflows so they do not affect neighboring nibbles.

	1		1					1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
	0	0	0	1	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	0	1	+
	1	1	0	1	1	0	0	1	0	0	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	0	1	0	1	1	=
	1	1	1	0	1	1	1	0	1	1	0	1	0	1	0	0	0	1	1	0	0	1	0	1	0	0	1	1	1	1	0	

Figure 2.1: Addition of two 64-bit registers, where each one stores sixteen 4-bit unsigned integers. Carry bits are shown above the registers. Blue carry bits indicate propagation contained in the nibble. Red bits indicate carry propagation from one nibble to the next.

It is pretty easy to notice that the value of a nibble is altered if the carry-in to the least significant bit of that nibble is set. If we could disable every fourth carry bit, we would effectively have sixteen independent 4-bit adders, and our problem would be solved. Unfortunately, we cannot do that in software. What we can do, though, is detect those carry-in bits.

2.2 The Double XOR Method for Addition

Let's take a step back and delve deeper into how binary addition works. A one-bit full adder is a combinational logic circuit that adds three one-bit numbers, often written as $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C_{in}}$; $\boldsymbol{A}$ and $\boldsymbol{B}$ are the operands, and $\boldsymbol{C_{in}}$ is a bit carried in from the previous less significant stage. The circuit produces a two-bit output. Output carry and sum are typically represented by the signals $\boldsymbol{C_{out}}$ and $\boldsymbol{S}$, where the sum equals $\boldsymbol{S + 2C_{out}}$ [1]. In multiple-bit adders, each stage's $\boldsymbol{C_{out}}$ is passed to the next more significant stage as $\boldsymbol{C_{in}}$.

Something interesting happens if we compute $\boldsymbol{X = A \oplus B \oplus S}$.[1] As shown in table 2.1, $\boldsymbol{X}$ is equal to the carry-in bit $\boldsymbol{C_{in}}$.

$\boldsymbol{A}$	$\boldsymbol{B}$	$\boldsymbol{C_{in}}$	$\boldsymbol{S}$	$\boldsymbol{C_{out}}$	$\boldsymbol{X}$
0	**0**	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**0**	**1**
0	**1**	**0**	**1**	**0**	**0**
0	**1**	**1**	**0**	**1**	**1**
1	**0**	**0**	**1**	**0**	**0**
1	**0**	**1**	**0**	**1**	**1**
1	**1**	**0**	**0**	**1**	**0**
1	**1**	**1**	**1**	**1**	**1**

Table 2.1: Truth table for a single-bit full adder. $\boldsymbol{A}$ and $\boldsymbol{B}$ are the operands, $\boldsymbol{S}$ is the sum, $\boldsymbol{C_{in}}$ and $\boldsymbol{C_{out}}$ are the carry-in and carry-out bits. $\boldsymbol{X = A \oplus B \oplus S}$.

We can now go back to our registers. What we just described applies to every bit in them, which means we have a way to detect all the carry-in bits—entirely in software. Suppose our input registers are called $\boldsymbol{a}$ and $\boldsymbol{b}$, and we compute their sum as $\boldsymbol{r = a + b}$ and then calculate $\boldsymbol{w = a \oplus b \oplus r}$,[2] then each bit in $\boldsymbol{w}$ tells us whether the corresponding bit in the sum had a carry-in. Now, since we are working with 4-bit integers, we only care about the carry-ins that land at the least significant bit of each nibble (i.e., bit positions 4, 8, 12, etc.). We can isolate those by masking out the other bits—this can easily be done with a bitmask, like this: $\boldsymbol{x = w \wedge \mathtt{0x1111111111111111}}$.[3] The result $\boldsymbol{x}$ is an indicator that tells us exactly which nibble results were affected by a carry from the one to their right.

[1] Where $\oplus$ symbolizes the `XOR` operation.

[2] Here, $\oplus$ represents the bitwise `XOR` between registers.

[3] Where $\wedge$ is the bitwise `AND` operator and **0x1111111111111111** is a bitmask that has the least significant bit of every nibble set.

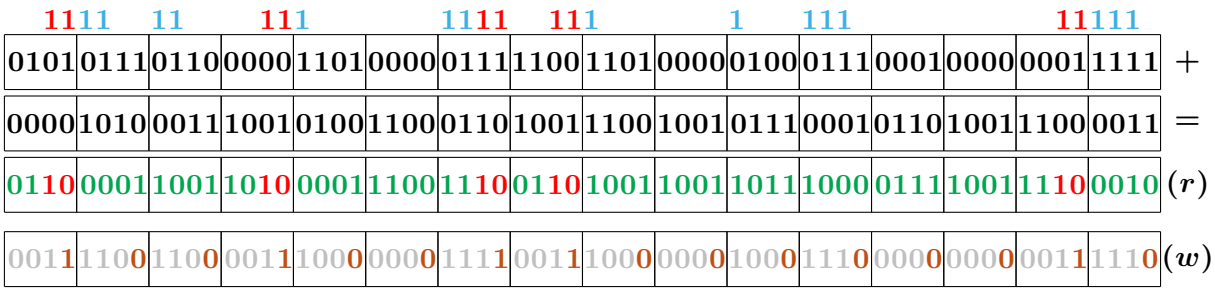

Figure 2.2: Same as 2.1, followed by the `XOR`-based carry-in detector $\boldsymbol{w} = \boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{r}$, where $\boldsymbol{a}$, $\boldsymbol{b}$ are the input registers and $\boldsymbol{r}$ is their sum. In $\boldsymbol{w}$, orange bits show whether carry-ins crossed the nibble boundaries. Gray bits will be masked out to compute $\boldsymbol{x}$.

At first glance, it might seem like we can just subtract our indicator $\boldsymbol{x}$ from the result $\boldsymbol{r}$ to fix the results. After all, if a nibble received a carry-in of one, subtracting one from it should compensate for the unwanted carry-in bit. Unfortunately, it does not. That is because this approach can end up overcorrecting. Imagine a case where a carry-in bit causes a nibble $\boldsymbol{n_1}$ to change from **1111** to **0000** with an overflow of **1** into the next nibble to the left $\boldsymbol{n_2}$. If we then subtract one from $\boldsymbol{n_1}$ to cancel out that overflow, it causes a borrow from $\boldsymbol{n_2}$. But here is the catch: we are already subtracting one from $\boldsymbol{n_2}$ too, because since $\boldsymbol{n_1}$ overflowed, our $\boldsymbol{x}$ detected a carry-in there as well. So now we are actually subtracting two from $\boldsymbol{n_2}$, altering the result.

To fix this properly, we would need to handle each nibble separately in this correction stage too. Instead of subtracting the entire $\boldsymbol{x}$ value at once, what we should do is a nibble-by-nibble subtraction. That is exactly what we will cover in the next section.

Figure 2.3: Same as 2.2, but showing how subtracting $\boldsymbol{x} = \boldsymbol{w} \wedge \mathbf{0x1111111111111111}$ from $\boldsymbol{r}$ does not fully fix the problem. In the subtraction, borrow bits are shown above the registers in yellow, and red bits indicate borrow propagation from one nibble to the next. In particular, the 14th nibble from the right ends up overcorrected because the borrow from the 13th nibble causes a subtraction of two instead of one.

2.3 Subtraction

Subtraction and addition work in surprisingly similar ways. Just like with addition, straight-up computing the difference between the two input registers does not work—if one of the 4-bit subtractions causes a borrow from the nibble to the left, the result gets altered.

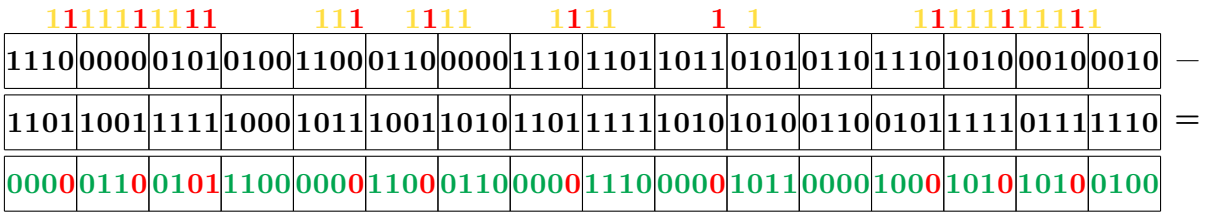

Figure 2.4: Subtraction between two 64-bit registers, where each one stores sixteen 4-bit unsigned integers. Borrow bits are shown above the registers. Yellow borrow bits indicate propagation contained in the nibble. Red bits indicate borrow propagation from one nibble to the next.

2.4 The Double `XOR` Method for Subtraction

What is interesting is that the `XOR`-based method we used to detect carry-in bits can be applied to subtraction to detect borrow-ins exactly as it is without needing any adjustments, as we can see from table 2.2.

A	C	B_{in}	D	B_{out}	X
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	0	0	0
1	1	1	1	1	1

Table 2.2: Truth table for a single-bit full subtractor. A and C are the operands, D is the difference, B_{in} and B_{out} are the borrow-in and borrow-out bits. $X = A \oplus C \oplus D$.

Following the same steps we used for addition, if we let $\boldsymbol{a}$ and $\boldsymbol{b}$ be our input registers and $\boldsymbol{r} = \boldsymbol{a} - \boldsymbol{b}$, we can compute $\boldsymbol{w} = \boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{r}$ and $\boldsymbol{x} = \boldsymbol{w} \wedge \mathbf{0x1111111111111111}$, where $\boldsymbol{x}$ tells us which nibbles received a borrow-in. Again, it is very tempting to just add $\boldsymbol{x}$ to $\boldsymbol{r}$ to correct for those borrows—but, just like with addition, this can lead to overcorrection in some cases.

Figure 2.5: Parallel of 2.3 for subtraction. In particular, when adding $\boldsymbol{x}$ to $\boldsymbol{r}$, the 10th nibble from the right ends up being overcorrected.

2.5 Recursive Double-XOR-Based Solution for Addition and Subtraction

In the previous sections, we found that when computing addition, the double XOR method gives us a way to detect the carry bits that alter the result. To correct for those carry bits, we would need to compute a nibble-wise subtraction between that altered result we got and the XOR-based indicator (which we called $\boldsymbol{x}$). For subtraction, it is exactly the same idea but flipped: this time, $\boldsymbol{x}$ tells us which nibbles had a borrow-in, and we would need to perform a nibble-wise addition to fix the result.

On that basis, we can sketch out an initial working solution by having our addition function call the subtraction function to fix the carry-ins, which then calls back the addition function again to correct for the borrow-ins, recursively. To prevent infinite recursion, we need to define a base case—that is pretty straightforward: we just have to check if $\boldsymbol{x}$

equals zero. If that is true, it means no nibble had a carry-in or borrow-in, and so we can just return our result without having to call the other function.

We can also prove that this algorithm always terminates in a finite number of steps—we will demonstrate it for addition, but the same proof also applies to subtraction. Looking at the first step, since the rightmost nibble cannot receive a carry-in, the first 4 bits of $\boldsymbol{x_1}$[4] are guaranteed to be zero. Next, we analyze the second step. Here, we are subtracting $\boldsymbol{x_1}$ from $\boldsymbol{r_1}$[5] and we already know the lowest 4 bits of $\boldsymbol{x_1}$ are zero. This, in turn, means that when computing the subtraction, the first and second nibbles cannot receive a borrow-in, so the first 8 bits of $\boldsymbol{x_2}$ must be zero. If we keep going, we notice that at each step $\boldsymbol{i}$, the first $\boldsymbol{4 \cdot i}$ bits of $\boldsymbol{x_i}$ are all zero. This means that after at most 16 steps, all 64 bits in $\boldsymbol{x_i}$ will be zero, and our algorithm will terminate.

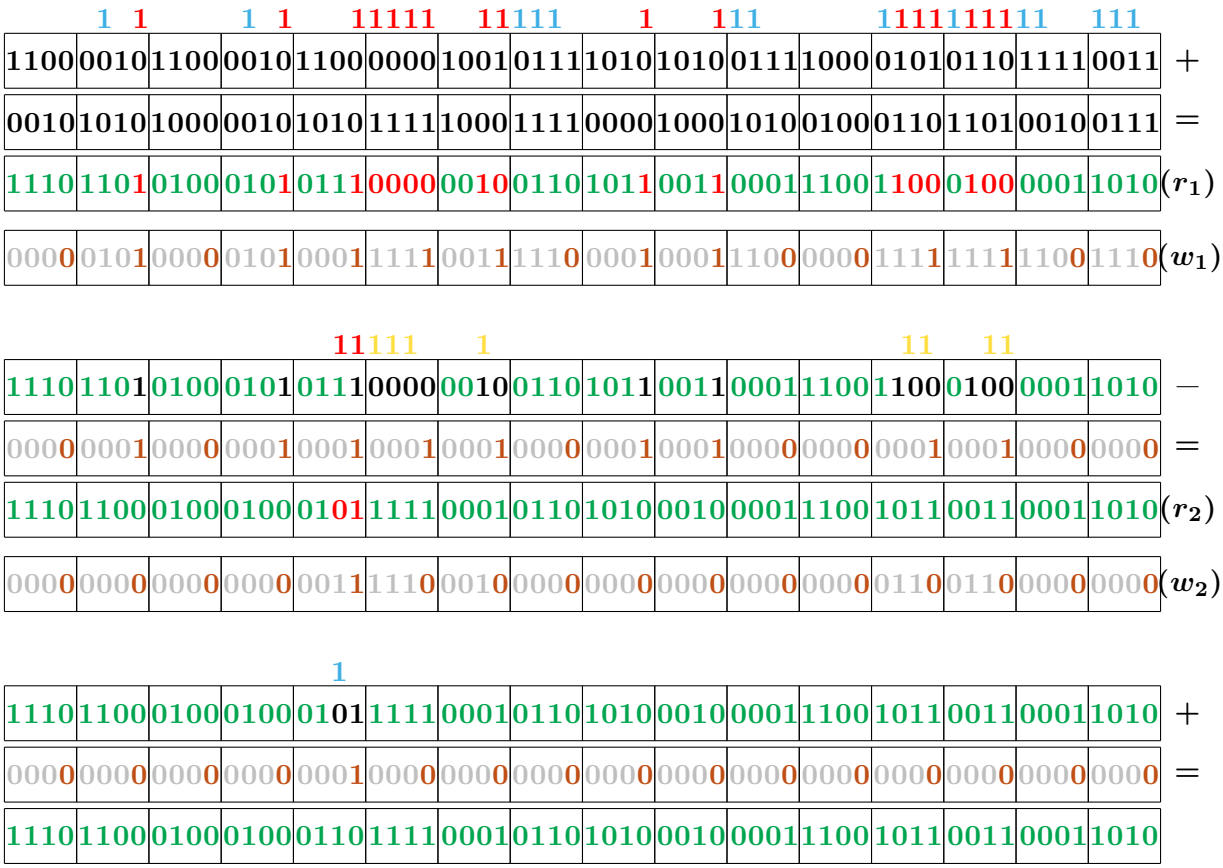

Figure 2.6: Demonstration of the double `XOR` algorithm for addition. This execution required three steps. Each operation is followed by $\boldsymbol{w_i}$, which is used to compute $\boldsymbol{x_i}$.

[4] $\boldsymbol{x_i}$ is the `XOR`-based indicator calculated at step $\boldsymbol{i}$, and used in step $\boldsymbol{i+1}$ to fix carry or borrow bits.
[5] $\boldsymbol{r_i}$ is the intermediate result after step $\boldsymbol{i}$.

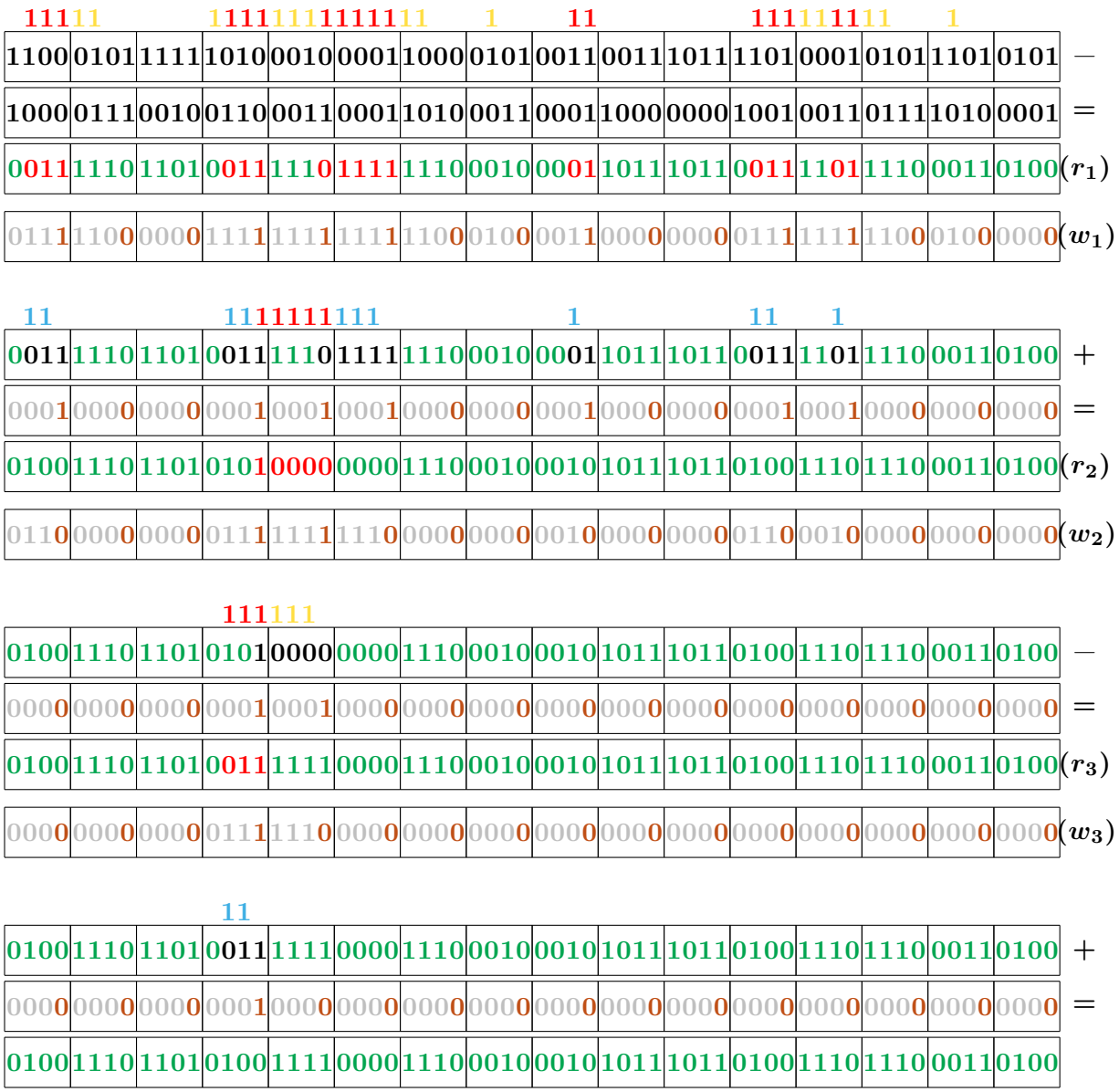

Figure 2.7: Demonstration of the double `XOR` algorithm for subtraction. This execution required four steps.

After running a billion simulations with random inputs for both addition and subtraction, we can see that, even though the algorithm could theoretically need up to 16 steps to finish, in practice almost two-thirds of the runs required just two, with an average of 2.375 steps needed. Fewer than 2 in a thousand runs required 5 or more steps, and none exceeded ten.[6]

[6]The first addition/subtraction between the input registers is counted as a step.

Steps	Addition	Subtraction
1	75,779	75,139
2	651,426,627	651,437,843
3	323,522,918	323,514,545
4	23,435,380	23,433,344
5	1,445,271	1,445,789
6	88,326	87,749
7	5,368	5,259
8	310	313
9	21	18
10	0	1

Table 2.3: Number of steps taken by the double `XOR` algorithm to compute the vectorized addition and subtraction of sixteen 4-bit unsigned integers, based on a billion simulations with random inputs.

Even though the recursive algorithm works, and the number of steps needed is usually small, the overhead from recursive calls renders it practically unusable if our goal is to speed up computations.

2.6 Iterative Double-`XOR`-Based Solution for Addition and Subtraction

A possible solution is to apply the same principles of the recursive approach, but write our code iteratively. To do that, we can write a loop where, in each iteration we compute either $\boldsymbol{r_i} = \boldsymbol{r_{i-1}} + \boldsymbol{x_{i-1}}$ or $\boldsymbol{r_i} = \boldsymbol{r_{i-1}} - \boldsymbol{x_{i-1}}$, depending on the step, to correct for carry-ins or borrow-ins from the previous iteration.[7]

Then, we compute $\boldsymbol{x_i} = (\boldsymbol{r_{i-1}} \oplus \boldsymbol{x_{i-1}} \oplus \boldsymbol{r_i}) \wedge \mathbf{0x1111111111111111}$, that detects potential borrow-ins or carry-ins induced by the correction itself. We keep doing this until $\boldsymbol{x_i}$ becomes zero.

Writing this in C++, we get the following functions. In this implementation, the first step is executed before the loop, and we are checking if $\boldsymbol{x_i}$ is zero every two steps, which reduces the overhead of frequent checks.

```
1 uint4x16_t vadd_u4(uint4x16_t _a, uint4x16_t _b)
```

[7]Except for the first step where $\boldsymbol{r_1} = \boldsymbol{a} \pm \boldsymbol{b}$ and $\boldsymbol{x_1} = (\boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{r_1}) \wedge \mathbf{0x1111111111111111}$.

```
{
    uint64_t r, a = _a.reg, b = _b.reg, x;
    r = a + b;
    x = (a ^ b ^ r) & 0x1111111111111111;
    while(x != 0)
    {
        a = r;
        r -= x;
        x = (x ^ a ^ r) & 0x1111111111111111;
        a = r;
        r += x;
        x = (x ^ a ^ r) & 0x1111111111111111;
    }
    return r;
}
```

Listing 2.1: Implementation of `vadd_u4` using the `XOR`-based approach iteratively.

```
uint4x16_t vsub_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t r, a = _a.reg, b = _b.reg, x;
    r = a - b;
    x = (a ^ b ^ r) & 0x1111111111111111;
    while(x != 0)
    {
        a = r;
        r += x;
        x = (x ^ a ^ r) & 0x1111111111111111;
        a = r;
        r -= x;
        x = (x ^ a ^ r) & 0x1111111111111111;
    }
    return r;
}
```

Listing 2.2: Implementation of `vsub_u4` using the `XOR`-based approach iteratively.

The following alternate implementation checks if $\boldsymbol{x_i}$ is zero after every step, avoiding unnecessary calculations, and the first step of the algorithm is computed in the loop.

```
uint4x16_t vadd_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a, r = _a.reg, x = _b.reg;
    while(x != 0)
    {
```

```
6           a = r;
7           r += x;
8           x = (a ^ r ^ x) & 0x1111111111111111;
9           if(x == 0)
10              break;
11          a = r;
12          r -= x;
13          x = (a ^ r ^ x) & 0x1111111111111111;
14      }
15      return r;
16  }
```

Listing 2.3: Alternate implementation of `vadd_u4` using the `XOR`-based approach iteratively.

```
1   uint4x16_t vsub_u4(uint4x16_t _a, uint4x16_t _b)
2   {
3       uint64_t a, r = _a.reg, x = _b.reg;
4       while(x != 0)
5       {
6           a = r;
7           r -= x;
8           x = (a ^ r ^ x) & 0x1111111111111111;
9           if(x == 0)
10              break;
11          a = r;
12          r += x;
13          x = (a ^ r ^ x) & 0x1111111111111111;
14      }
15      return r;
16  }
```

Listing 2.4: Alternate implementation of `vsub_u4` using the `XOR`-based approach iteratively.

While it is better than its recursive counterpart, the double `XOR` iterative algorithm still has a significant flaw: it uses conditional instructions. These can cause CPU pipeline stalls if the branch predictor misses, leading to slowdowns.

2.7 The High-Low Nibble Method for Addition

What we have been doing with the double `XOR` algorithm is correcting for unwanted carry-in bits. But what if, instead of correcting for them, we could prevent them from happening in the first place?

To do that, consider a setup where, in both our input registers $\boldsymbol{a}$ and $\boldsymbol{b}$, every other nibble is zero. Now, if we compute $\boldsymbol{a} + \boldsymbol{b}$, the non-zero nibbles can still overflow and produce a carry to their next nibble to the left—but, since those neighboring nibbles are all zero, the carry cannot propagate. That means the sums for those non-zero nibbles are all correct because they cannot receive a carry-in.

	1		11		1	1		11				11			1	
0000	0100	0000	1000	0000	0110	0000	1010	0000	0100	0000	1001	0000	1100	0000	0001	+
0000	0010	0000	1001	0000	0110	0000	1011	0000	1111	0000	0100	0000	1100	0000	1101	=
0000	0110	0001	0001	0000	1100	0001	0101	0001	0011	0000	1101	0001	1000	0000	1110	

Figure 2.8: Addition of two 64-bit registers storing sixteen independent 4-bit values, where every other nibble is zero. Carry bits are shown above the registers.

We can take advantage of this by splitting additions into two separate steps: first, we mask out the high nibbles[8] of both $\boldsymbol{a}$ and $\boldsymbol{b}$, which effectively zeroes out every other nibble. With that done, we can safely add them together to get the sum of the low nibbles, which we will call $\boldsymbol{r_{low}}$. Since some of the sums might overflow, we apply the same mask again to $\boldsymbol{r_{low}}$ to make sure every high nibble of it is zero, and get a first partial result that we are calling $\boldsymbol{s_{low}}$. We then repeat the process, but this time we mask out the low nibbles, add the registers together ($\boldsymbol{r_{high}}$), and mask out the low nibbles again to get $\boldsymbol{s_{high}}$. Finally, to get the intended result, we combine $\boldsymbol{s_{low}}$ and $\boldsymbol{s_{high}}$. Since we have ensured that each partial result has zeroes where the other has data, `OR`ing them together does the trick.

[8] High nibbles are the upper 4 bits of each byte, low nibbles are the lower 4.

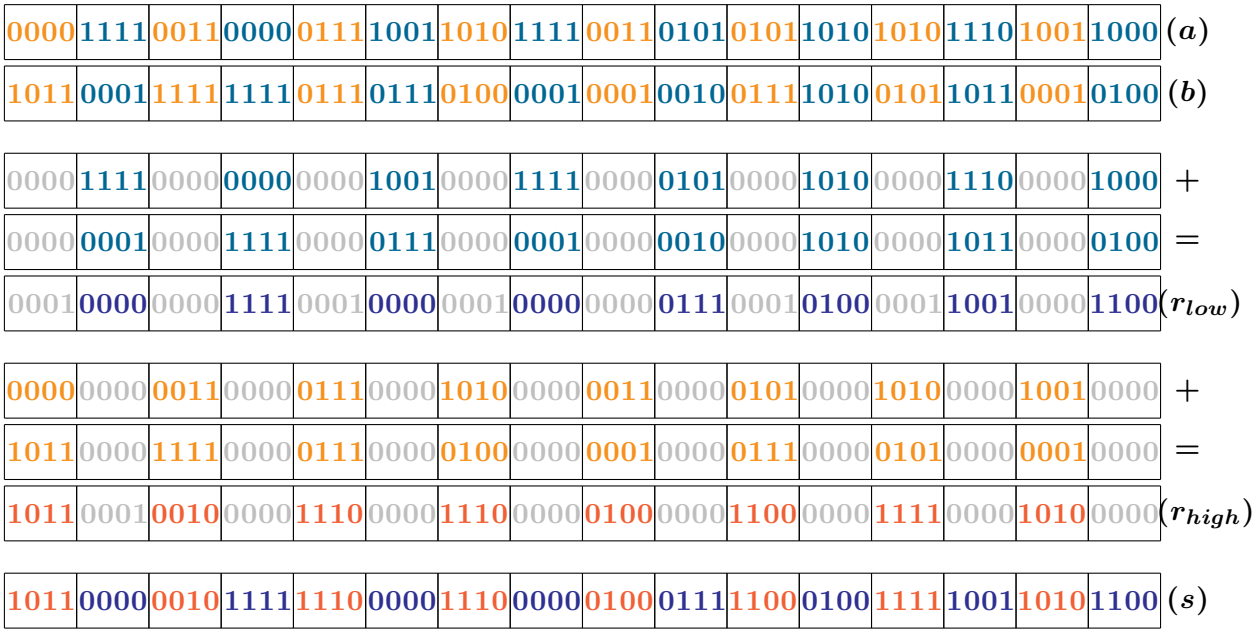

Figure 2.9: Nibble-wise addition of two 64-bit registers storing sixteen independent 4-bit values, using the high-low nibble method. Low nibbles are shown in blue, high nibble in orange. Gray bits in r_{low} and r_{high} must be masked out to get s_{low} and s_{high}. The last line shows the final result.

Implementing this in C++, we get the following function. As we can see, this function is completely branchless.

```
1 uint4x16_t vadd_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t a = _a.reg, b = _b.reg;
4     uint64_t s_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F)) & 0x0F0F0F0F0F0F0F0F;
5     uint64_t s_high = ((a & 0xF0F0F0F0F0F0F0F0) + (b &
    0xF0F0F0F0F0F0F0F0)) & 0xF0F0F0F0F0F0F0F0;
6     return s_low | s_high;
7 }
```

Listing 2.5: Branchless implementation of `vadd_u4`.

2.8 The High-Low Nibble Method for Subtraction

A similar approach can be applied to subtraction, with a key adjustment. The reason zeroing out first the high and then the low nibbles works for addition is that—even if

they receive a carry-in—the sum of the zeroed nibbles never overflows and cannot cause a carry-out. That means the sum of the *active* nibbles—the ones we did not zero out—ends up being correct.

For subtraction, though, this would not work because, if a borrow-in occurs in the zeroed-out nibbles, we would be subtracting one from zero, causing a borrow-out. Since our goal is to prevent borrow propagation, we need to make sure that *inactive* nibbles—the ones we are not trying to compute—never underflow. To do that, instead of zeroing them all out, we can simply set them to **1111**[9] in the first register (the minuend), and **0000** in the second one (the subtrahend). This guarantees that, even if there is a borrow-in, it will not propagate, and the differences in the active nibbles will remain unaffected. We can then proceed to combine the two partial results like we did for addition. The value left in the inactive nibbles does not matter because, like for addition, we are masking them out anyway.

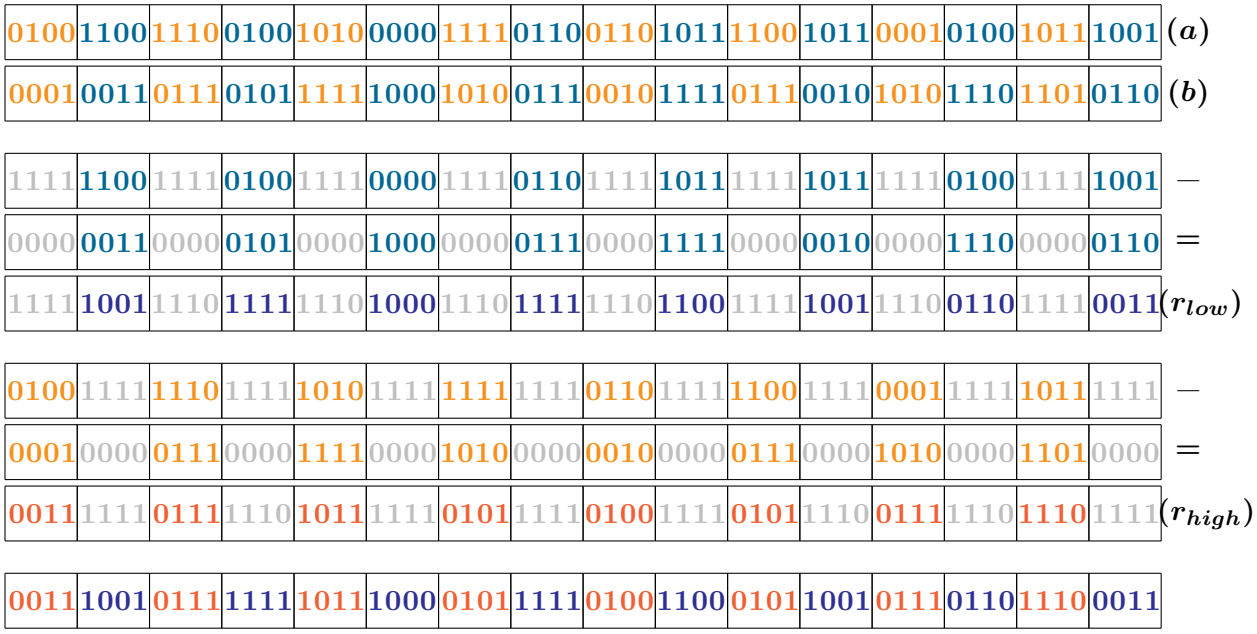

Figure 2.10: Parallel of 2.9 for subtraction.

This is how we can implement it in C++.

```
1 uint4x16_t vsub_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t a = _a.reg, b = _b.reg;
```

[9]To do that, we mask in all bits in those nibbles using a bitwise `OR`.

```
4     uint64_t d_low = ((a | 0xF0F0F0F0F0F0F0F0) - (b &
    0x0F0F0F0F0F0F0F0F)) & 0x0F0F0F0F0F0F0F0F;
5     uint64_t d_high = ((a | 0x0F0F0F0F0F0F0F0F) - (b &
    0xF0F0F0F0F0F0F0F0)) & 0xF0F0F0F0F0F0F0F0;
6     return d_low | d_high;
7 }
```

Listing 2.6: Branchless implementation of `vsub_u4`.

2.9 Addition with the High Bit `XOR` Method

Another method for performing nibble-wise addition without conditional instructions is what we call the high bit `XOR` method [2]. This technique also computes the result in two steps but reduces the number of operations compared to previous approaches.

First, we clear the high bit of each nibble of both input registers $\boldsymbol{a}$ and $\boldsymbol{b}$. Specifically, we do that by `AND`ing them with the bitmask **0x7777777777777777**. This prevents carry propagation between adjacent nibbles when we compute $\boldsymbol{l}$, the sum of these masked inputs. As a result, the lower three bits of each nibble in $\boldsymbol{l}$ contain the correct sum bits for those positions. To complete the sums, we have to determine the most significant bit of every nibble.

By looking at table 2.1, we can see that each output bit in an adder can be expressed as a `XOR` of the input bits and the carry-in. Observing $\boldsymbol{l}$, we notice that the most significant bit of each nibble is set if and only if it received a carry-in during the addition. This is because these high bits[10] were masked out—so both inputs were zero in those positions, and any nonzero result in those bits of $\boldsymbol{l}$ must be due to a carry-in. In other words, had we not cleared the high bits before adding, the values now sitting in the high bits of $\boldsymbol{l}$ would have instead been the carry-in signals to those positions during the addition.

This means that to correctly compute the high bits of the intended results, we need to `XOR` the most significant bit of each nibble in $\boldsymbol{l}$ (the carry-in) with the most significant bit of each nibble in $\boldsymbol{a}$ and $\boldsymbol{b}$ (the inputs), basically simulating single-bit adders in software. Since the `XOR` operation is associative, we can do this by first computing $\boldsymbol{z} = (\boldsymbol{a} \oplus \boldsymbol{b}) \wedge \mathbf{0x8888888888888888}$. The value $\boldsymbol{z}$ gives us the `XOR` of the high bits of $\boldsymbol{a}$ and $\boldsymbol{b}$, with all other bits masked out. Now we can compute the final result $\boldsymbol{s}$ as $\boldsymbol{s} = \boldsymbol{l} \oplus \boldsymbol{z}$. Since we cleared the lower three bits of each nibble when computing $\boldsymbol{z}$, we know those bits were not altered—`XOR`ing something with 0 leaves it unchanged.

Here is a possible implementation.

[10] High bits is used as a synonym for most significant bit of each nibble.

```
uint4x16_t vadd_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg;
    uint64_t l = (a & 0x7777777777777777) + (b & 0x7777777777777777);
    uint64_t z = (a ^ b) & 0x8888888888888888;
    return l ^ z;
}
```

Listing 2.7: Implementation of `vadd_u4` using the high bit `XOR` method.

2.10 Subtraction with the High Bit `XOR` Method

A similar approach can be applied to subtraction. To adapt the method, we still clear the most significant bit of each nibble in $\boldsymbol{b}$ when computing $\boldsymbol{l}$, but instead of doing the same for $\boldsymbol{a}$, we actually mask *in* those same bits there. This ensures that the lower three bits of each nibble in $\boldsymbol{l}$ still contain the correct difference bits for those positions.

Looking at table 2.2, we see that each output bit can still be computed as the `XOR` of the two inputs and the borrow-in bit. However, the high bits in $\boldsymbol{l}$ are now set if and only if they did *not* receive a borrow-in. To adapt the addition method, we would have to invert the most significant bit of each nibble in $\boldsymbol{l}$, but leave the other bits unchanged since they already contain the correct bits of the result.

Rather than doing that—which would require multiple instructions—we can take advantage of another property of `XOR`: inverting two inputs does not change the result. Since the high bits in $\boldsymbol{l}$ are inverted, we also invert high bits in $\boldsymbol{z}$. This is faster because we are masking out the other bits in $\boldsymbol{z}$ anyway. We can compute this as: $\boldsymbol{z} = (\neg(\boldsymbol{a} \oplus \boldsymbol{b})) \wedge \mathbf{0x8888888888888888}$.[11] Finally, we get our final result $\boldsymbol{d}$ by `XOR`ing $\boldsymbol{l}$ with $\boldsymbol{z}$, just like before.

This is a possible implementation.

```
uint4x16_t vsub_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg;
    uint64_t l = (a | 0x8888888888888888) - (b & 0x7777777777777777);
    uint64_t z = ~(a ^ b) & 0x8888888888888888;
    return l ^ z;
}
```

Listing 2.8: Implementation of `vsub_u4` using the high bit `XOR` method.

[11] $\neg$ here denotes the bitwise NOT operation.

2.11 Further Implementations

```
1 uint4x16_t vsub_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t r, a = _a.reg, b = _b.reg, x;
4     r = a - b;
5     x = (a ^ b ^ r) & 0x1111111111111111;
6     uint64_t s_low = ((r & 0x0F0F0F0F0F0F0F0F) + (x &
    0x0F0F0F0F0F0F0F0F)) & 0x0F0F0F0F0F0F0F0F;
7     uint64_t s_high = ((r & 0xF0F0F0F0F0F0F0F0) + (x &
    0xF0F0F0F0F0F0F0F0)) & 0xF0F0F0F0F0F0F0F0;
8     return s_low | s_high;
9 }
```

Listing 2.9: Additional implementation of `vsub_u4`. Here, we mixed the `XOR`-based approach with the high-low nibble method. This code computes $\boldsymbol{x}$ to detect borrow-ins, and corrects for them by computing a nibble-wise addition using the high-low bit method.

```
1 uint4x16_t vadd_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t r, a = _a.reg, b = _b.reg, x, y;
4     r = a + b;
5     x = (a ^ b ^ r) & 0x1111111111111111;
6     y = (r<<1) | (r<<2) | (r<<3) | (r<<4) | ~(x<<4);
7     x = x & y;
8     return r - x;
9 }
```

Listing 2.10: Additional implementation of `vadd_u4`. Here, we use the `XOR`-based approach, but we first clear the bits in $\boldsymbol{x}$ that would cause overcorrection. Specifically, if a nibble in $\boldsymbol{r}$ has a value of **0000**, and $\boldsymbol{x}$ detected a carry-in to both that nibble and the next nibble to the left, subtracting $\boldsymbol{x}$ from $\boldsymbol{r}$ would cause a borrow-in, leading to overcorrection. This code, before performing the subtraction, masks out unwanted bits in $\boldsymbol{x}$ by checking if, for each nibble, the previous nibble in $\boldsymbol{r}$ is zero and there is a carry in to that nibble, which is what we do in lines 6 and 7, where $\boldsymbol{y}$ is the mask we used to clear those unwanted bits in $\boldsymbol{x}$. Since those checks only use bitwise operations, this implementation effectively uses the double `XOR` method without conditional instructions.

Chapter 3

Saturating Addition and Subtraction

3.1 The Concept of Saturating Arithmetic

The difference between a saturating and a non-saturating operation comes down to how overflows are handled. Since we are working with 4-bit unsigned integers, we can only represent values from 0 to 15. Non-saturating operations—the ones we have been working with up until now—essentially ignore the existence of overflows altogether and let the values just wrap around, exactly like you would expect a regular integer variable to do when it overflows, leading to modulo-16 behavior. Sometimes, though, this behavior might not be ideal.

Think, for example, of increasing the brightness of an RGB image where each sub-pixel has sixteen brightness levels and is stored on 4 bits. To do that, we need to add a fixed value—say, one—to every sub-pixel. But what if a sub-pixel is already at its maximum value? We certainly do not want its brightness to wrap around and go back to zero. Ideally, we would increase its brightness, but since we have already hit the upper limit, the next best thing is to leave it at the max. That is what saturating addition does: instead of wrapping, it clamps any overflowed values to 15. Similarly, saturating subtraction clamps any results that would go below zero to zero.

3.2 Saturating Addition with the Double `XOR` Method

Let's quickly review the idea behind the double `XOR` method for regular (non-saturating) addition: first, we add the input registers together and get $\boldsymbol{r}$. Then, we compute $\boldsymbol{x}$, which tells us which nibbles were altered by a carry-in. Finally, we fix those nibbles through a nibble-wise subtraction between the intermediate result $\boldsymbol{r}$ and $\boldsymbol{x}$.

To implement saturating behavior, we now want to set the nibbles that overflowed to **1111**. And it turns out $\boldsymbol{x}$[1] can help us do that too. That is because if a nibble received a carry-in, it means the nibble before it—the one to its right—overflowed. So, since $\boldsymbol{x}$ can detect carry-ins, it can also detect overflows. Specifically, a one in the least significant bit of a nibble in $\boldsymbol{x}$ indicates that the nibble to its right experienced an overflow.[2] If we now shift $\boldsymbol{x}$ four bits to the right, we end up with an indicator that has a one in the least significant bit of each nibble that overflowed.[3] We can call this overflow indicator $\boldsymbol{o}$.

The easiest way to set specific nibbles in a register to **1111** is to bitwise `OR` the register with a bitmask that has **1111** in the positions corresponding to the nibbles we want to set, and zero elsewhere. And $\boldsymbol{o}$ gets us most of the way there—it correctly flags the overflowed nibbles, but only has their least significant bit set. To get a full nibble of ones in those positions, we just compute $\boldsymbol{f} = \boldsymbol{o} \cdot \mathbf{15}$.[4] So, if we call $\boldsymbol{s}$ the result of the non-saturating addition, which we can get using our iterative `XOR` algorithm, then we can *almost* compute our saturating addition as $\boldsymbol{s_s} = \boldsymbol{s} \vee \boldsymbol{f}$.[5]

Almost because we did not detect whether the leftmost nibble overflowed. That is because there is no nibble to its left that could indicate a carry-in. To catch this case, we can check if the full 64-bit sum of the input registers, $\boldsymbol{a} + \boldsymbol{b}$, overflows. If it does, that means the sum of the leftmost nibble also overflowed, and we have to manually set it to **1111**, either by setting it directly in the result, or setting it in the bitmask before we compute the bitwise `OR`.

Here is a possible implementation in C++.

```
uint4x16_t vqadd_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t r, a = _a.reg, b = _b.reg, x;
    r = a + b;
    x = (a ^ b ^ r) & 0x1111111111111111;
    uint64_t f = (x >> 4) * 15;
    if(r < a)
        f |= (0xFull << 60);
    while(x != 0)
```

[1]The iterative double `XOR` method computes a different $\boldsymbol{x_i}$ at each iteration, but to implement saturation we only care about the one computed at the first step, which we are referring to as just $\boldsymbol{x}$.

[2]It is possible that a nibble only overflows because of a carry in, and this method will still detect it. However, if a carry in makes a nibble overflow, that means its value, once we correct for the carry-ins, is **1111**. Since overflowed nibbles will be set to that same value, this does not cause any issues.

[3]Except for the leftmost nibble, but we will come back to that later.

[4]When enabling optimizations, this multiplication is compiled as a left shift and a subtraction, so we do not have to worry about it being slow.

[5]Where $\vee$ is the bitwise `OR` operator.

```
10      {
11          a = r;
12          r -= x;
13          x = (x ^ a ^ r) & 0x1111111111111111;
14          if(x == 0)
15              break;
16          a = r;
17          r += x;
18          x = (x ^ a ^ r) & 0x1111111111111111;
19      }
20      r |= f;
21      return r;
22  }
```

Listing 3.1: Implementation of `vqadd_u4` using the double `XOR` method. Lines 7 and 8 handle the leftmost nibble.

3.3 Saturating Subtraction with the Double `XOR` Method

We can apply the same ideas behind saturating addition to saturating subtraction, with some adjustments.

It is important to note that now, instead of setting overflowed nibbles to **1111**, we want to set *underflowed* nibbles to **0000**. As with addition, $\boldsymbol{x}$ has the last bit of each nibble set if the nibble to its right underflows.[6] That means that our idea of shifting it 4 bits to the right to detect the nibbles we want to set to **0000** can be applied here too. Just like before, we will call this underflow indicator $\boldsymbol{o}$.

When performing saturating addition, we took this indicator, multiplied it by 15 to get a value $\boldsymbol{f}$ that had **1111** in the nibbles that overflowed, and `OR`ed it with the result of the non-saturating addition, $\boldsymbol{s}$. That worked because we were saturating upward. But for subtraction, we want to saturate downward—i.e., force underflowed nibbles to zero. To do that, we simply bitwise `NOT` $\boldsymbol{f}$ and `AND` it with the non-saturated difference. So, if $\boldsymbol{d}$ is the non-saturated difference and $\boldsymbol{f}$ is the same mask as before, the saturated result can be calculated like this: $\boldsymbol{d_s} = \boldsymbol{d} \wedge \neg \boldsymbol{f}$. As with addition, this method does not handle the leftmost nibble, so we still need to treat it manually.

The code below implements saturating subtraction using the Double-`XOR` method and handles the leftmost nibble with a conditional check.

```
1 uint4x16_t vqsub_u4(uint4x16_t _a, uint4x16_t _b)
```

[6] We are still not considering the leftmost nibble for now.

```
2 {
3     uint64_t r, a = _a.reg, b = _b.reg, x;
4     r = a - b;
5     x = (a ^ b ^ r) & 0x1111111111111111;
6     uint64_t f = (x >> 4) * 15;
7     if(r > a)
8         f |= (0xFull << 60);
9     while(x != 0)
10    {
11        a = r;
12        r += x;
13        x = (x ^ a ^ r) & 0x1111111111111111;
14        if(x == 0)
15            break;
16        a = r;
17        r -= x;
18        x = (x ^ a ^ r) & 0x1111111111111111;
19    }
20    r &= ~f;
21    return r;
22 }
```

Listing 3.2: Implementation of `vqsub_u4` using the double `XOR` method. Lines 7 and 8 handle the leftmost nibble.

3.4 Saturating Addition with a Hybrid Solution

As we have mentioned before, the biggest drawback of the double `XOR` method is its reliance on conditional instructions. The algorithm we used to compute our saturating addition uses $\boldsymbol{x}$ to detect which sums overflowed, but that does not necessarily mean we have to use the double `XOR` method to compute $\boldsymbol{s}$ too. We can use $\boldsymbol{x}$ to detect overflows, but compute $\boldsymbol{s}$ using the high bit `XOR` method.[7] This leads to the following code.

```
1 uint4x16_t vqadd_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t a = _a.reg, b = _b.reg, s, r;
4     uint64_t l = (a & 0x7777777777777777) + (b & 0x7777777777777777);
5     uint64_t z = (a ^ b) & 0x8888888888888888;
6     s = l ^ z;
7     r = a + b;
```

[7] We can actually use whatever method we want.

```
8     uint64_t x = (a ^ b ^ r) & 0x1111111111111111;
9     uint64_t f = (x >> 4) * 15;
10    if(r < a)
11        f |= (0xFull << 60);
12    return s | f;
13 }
```

Listing 3.3: Implementation of `vqadd_u4` using the double `XOR` method to detect overflows and the high bit `XOR` method to compute the non-saturated sum. Lines 10 and 11 handle the leftmost nibble.

The problem with this implementation is that it is not completely free of conditional instructions. That is because it uses an `if` statement to check whether $\boldsymbol{a} + \boldsymbol{b}$ overflows—which also means the leftmost nibble overflows. But that is not really necessary. In the x86-64 architecture, when an `add` instruction overflows, the carry flag is set. Therefore, if we could access the flags, we would not need a conditional instruction to detect the overflow. Unfortunately, we cannot do that in C++: we need to drop down to Assembly.

Instead of writing the whole function in Assembly from scratch, we can remove lines 10 and 11 from the C++ code, compile it, and then patch the resulting Assembly to reintroduce the detection of whether the most significant nibble overflows. If it does, we will set it to **1111**. Compiling with GCC 11.4.0 on a Linux system with the `-O2` optimization flag, the compilation gives us the following Assembly output.

```
1  _Z8vqadd_u410uint4x16_tS_:
2      movabsq $8608480567731124087, %rdx
3      movq    %rdi, %rax
4      movq    %rdi, %rcx
5      addq    %rsi, %rdi
6      andq    %rdx, %rax
7      andq    %rsi, %rdx
8      xorq    %rsi, %rcx
9      addq    %rdx, %rax
10     xorq    %rcx, %rdi
11     movabsq $-8608480567731124088, %rdx
12     andq    %rcx, %rdx
13     shrq    $4, %rdi
14     xorq    %rdx, %rax
15     movabsq $76861433640456465, %rdx
16     andq    %rdx, %rdi
17     movq    %rdi, %rdx
18     salq    $4, %rdx
19     subq    %rdi, %rdx
```

```
20      orq     %rdx, %rax
21      ret
```

Before we attempt to edit this, we first have to understand what it is doing. According to the System V calling convention, $\boldsymbol{a}$ and $\boldsymbol{b}$ are passed in `%rdi` and `%rsi`, respectively. By line 5, `%rdi` and `%rsi` still hold $\boldsymbol{a}$ and $\boldsymbol{b}$, because neither of them has been written to yet. This means that line 5 performs $\boldsymbol{a} + \boldsymbol{b}$, which is very important to us, since that is where the overflow may occur. Lines 8 and 10 compute $\boldsymbol{w}$ and store it in `%rdi`. Instead of computing $\boldsymbol{x}$ from $\boldsymbol{w}$ and then shifting it to get $\boldsymbol{o}$, the compiler chose to shift $\boldsymbol{w}$ first (line 13) and then apply the mask (lines 15 and 16), never actually computing $\boldsymbol{x}$. The result, $\boldsymbol{o}$, is again in `%rdi`. Then, lines 17 to 19 multiply $\boldsymbol{o}$ by 15, storing the result, $\boldsymbol{f}$, in `%rdx`. Meanwhile, lines 2 to 4, 6 to 9,[8] 11, 12, and 14 compute $\boldsymbol{s}$ using the high bit `XOR` method and keep it in `%rax`. Finally, line 20 applies the mask $\boldsymbol{f}$ to $\boldsymbol{s}$ (which is still in `%rax`). Since, per the System V convention, `%rax` is the return register, we are good to go.

Now that we understand how the code works, we can edit it. If $\boldsymbol{a} + \boldsymbol{b}$ overflows, the carry flag will be set after line 5. We can capture it using the `adcq` instruction like this:

```
    xorq    %r9, %r9
    addq    %rsi, %rdi
    adcq    %r9, %r9
    salq    $60, %r9
```

Here, we first set `%r9`[9] to zero. After the addition, `adcq` adds `%r9`, which we set to zero, to `%r9`, plus the carry flag, so if there was an overflow, `%r9` becomes 1. Finally, we shift `%r9` sixty positions to the left, placing the carry bit into the leftmost nibble. Now we can just insert the line

```
    orq     %r9, %rdi
```

before we multiply $\boldsymbol{o}$ by fifteen: assuming the sum did overflow, this sets the least significant bit of the leftmost nibble to one. Consequently, when we compute $\boldsymbol{f}$, its leftmost nibble will be set to **1111**, which is exactly what we want.

Putting it all together, the modified Assembly looks like this:

```
1 _Z8vqadd_u410uint4x16_tS_:
2     movabsq $8608480567731124087, %rdx
3     movq    %rdi, %rax
4     movq    %rdi, %rcx
```

[8] Line 8 is used to compute both $\boldsymbol{f}$ and $\boldsymbol{s}$. This is because this line computes $\boldsymbol{a} \oplus \boldsymbol{b}$, used in both calculations, and the compiler optimized computing this intermediate result only once.

[9] This implementation uses `%r9`, but any other scrap register that is not already used in the original code would have worked.

```
    xorq    %r9, %r9
    addq    %rsi, %rdi
    adcq    %r9, %r9
    salq    $60, %r9
    andq    %rdx, %rax
    andq    %rsi, %rdx
    xorq    %rsi, %rcx
    addq    %rdx, %rax
    xorq    %rcx, %rdi
    movabsq $-8608480567731124088, %rdx
    andq    %rcx, %rdx
    shrq    $4, %rdi
    xorq    %rdx, %rax
    movabsq $76861433640456465, %rdx
    andq    %rdx, %rdi
    orq     %r9, %rdi
    movq    %rdi, %rdx
    salq    $4, %rdx
    subq    %rdi, %rdx
    orq     %rdx, %rax
    ret
```

Listing 3.4: Implementation of `vqadd_u4` in Assembly using the double `XOR` method to detect overflows and the high bit `XOR` method to compute the non-saturated sum, utilizing the `adcq` instruction to detect a potential overflow in the leftmost nibble.

3.5 Saturating Subtraction with a Hybrid Solution

Just like with addition, here we can detect underflows using $\boldsymbol{x}$, but compute $\boldsymbol{d}$ with the high bit `XOR` method. The following code does exactly that, though it still uses an `if` statement to handle the leftmost nibble.

```
uint4x16_t vqsub_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, r, d;
    uint64_t l = (a | 0x8888888888888888) - (b & 0x7777777777777777);
    uint64_t z = ~(a ^ b) & 0x8888888888888888;
    d = l ^ z;
    r = a - b;
    uint64_t x = (a ^ b ^ r) & 0x1111111111111111;
    uint64_t f = (x >> 4) * 15;
    if(r > a)
```

```
11          f |= (0xFull << 60);
12      return d & ~f;
13  }
```

Listing 3.5: Implementation of `vqsub_u4` using the double `XOR` method to detect underflows and the high bit `XOR` method to compute the non-saturated difference. Lines 10 and 11 handle the leftmost nibble.

To avoid the `if` and make the function completely conditionless, we can handle the leftmost nibble by accessing the carry flag in Assembly—just like we did for addition. Again, we are not writing the entire function in Assembly from scratch. Instead, we compiled[10] a slightly modified version of the function above (where we removed lines 10 and 11) and patched it to make it handle the leftmost nibble correctly.

After patching it, this is the resulting Assembly code. Like before, this code follows the System V calling convention, used on Linux.

```
1  _Z8vqsub_u410uint4x16_tS_:
2      movabsq $76861433640456465, %rax
3      movq    %rdi, %rcx
4      movq    %rdi, %rdx
5      movabsq $8608480567731124087, %r8
6      xorq    %rsi, %rcx
7      xorq    %r9, %r9
8      subq    %rsi, %rdx
9      adcq    %r9, %r9
10     salq    $60, %r9
11     andq    %r8, %rsi
12     xorq    %rcx, %rdx
13     notq    %rcx
14     shrq    $4, %rdx
15     andq    %rax, %rdx
16     orq     %r9, %rdx
17     movq    %rdx, %rax
18     salq    $4, %rax
19     subq    %rdx, %rax
20     movabsq $-8608480567731124088, %rdx
21     orq     %rdx, %rdi
22     andq    %rdx, %rcx
23     notq    %rax
24     subq    %rsi, %rdi
25     xorq    %rcx, %rdi
26     andq    %rdi, %rax
```

[10] Using GCC 11.4.0 on a Linux system using the `-O2` optimization flag.

```
27      ret
```

Listing 3.6: Implementation of `vqsub_u4` in Assembly using the double `XOR` method to detect underflows and the high bit `XOR` method to compute the non-saturated difference, utilizing the `adcq` instruction to detect a potential underflow in the leftmost nibble.

Since the saturating addition and subtraction functions are similar, we added the same lines here as we did when patching the Assembly for saturating addition, though changing the operands because the compiler did not use the same registers for our values. Specifically, we added lines 7, 9, 10, and 16. Line 8 is where the subtraction between $\boldsymbol{a}$ and $\boldsymbol{b}$ happens, which is very important—whether or not this subtraction sets the carry flag tells us if the leftmost nibble underflows.

3.6 Saturating Addition with the High-Low Nibble Method

We can also compute our saturating operations without using $\boldsymbol{x}$ at all. Just like when we first introduced the high-low nibble method, assume both input registers have zeros in all their high nibbles. We have already seen how—when adding those registers together—that prevents carry-ins from altering the sums of the low nibbles.

Now we have to turn our attention to what happens in the high nibbles—the ones we previously just masked out when computing non-saturating addition. Since both input registers have zeros in those nibbles, if there is no overflow from the nibble on the right, the corresponding high nibble in the result stays zero. But if a nibble does overflow, it causes a carry-out, which ends up setting the least significant bit of the high nibble immediately to its left.

0000	1100	0000	1101	0000	0001	0000	0110	0000	0100	0000	0010	0000	1110	0000	1100	+
0000	0100	0000	1100	0000	1010	0000	1110	0000	0001	0000	1000	0000	1011	0000	0000	=
0001	0000	0001	1001	0000	1011	0001	0100	0000	0101	0000	1010	0001	1001	0000	1100	

Figure 3.1: Nibble-wise addition of two 64-bit registers storing sixteen independent 4-bit values, where high nibbles are zero. The least significant bit of every high nibble in the sum, $\boldsymbol{r_{low}}$, is highlighted in purple.

The same applies if the zeroes in the input registers are in the low nibbles instead of

the high ones.[11]

We can take advantage of this behavior to detect overflows. Start by computing the non-saturated sum $\boldsymbol{s}$ using the high-low nibble method, but now, when we calculate the sum of the low nibbles, $\boldsymbol{s_{low}}$, we also create an overflow indicator $\boldsymbol{o_{low}}$ by masking out low nibbles in $\boldsymbol{r_{low}}$—the ones holding our sums. We do the opposite when computing $\boldsymbol{s_{high}}$, and get another overflow indicator $\boldsymbol{o_{high}}$. If we now `OR` $\boldsymbol{o_{low}}$ and $\boldsymbol{o_{high}}$ together, we end up with a value where the least significant bit of each nibble is set if the nibble to its right overflowed. Shifting this result 4 bits to the right, we basically get the same kind of overflow indicator $\boldsymbol{o}$ we got by shifting $\boldsymbol{x}$.[12]

Using this indicator, we can now *almost* compute our saturating addition using the same formula: $\boldsymbol{s_s = s \vee (o \cdot 15)}$. Again, *almost*, because just like before, we still have to handle the special case of the leftmost nibble. One way to do that without conditional instructions is to retrieve the carry flag in Assembly after summing the high nibbles—similar to what we did earlier. But this time, there is another trick we can use that requires no Assembly.

The leftmost nibble is tricky because it does not have another nibble to its left to catch the carry-out when it overflows. But we can fix that by shifting our inputs 4 bits to the right before computing $\boldsymbol{r_{high}}$. This moves the original high nibble values into the low positions, so now all the resulting carry-outs can be caught. We obviously have to make sure to shift $\boldsymbol{r_{high}}$ back 4 bits to the left when we mask out its inactive nibbles and compute $\boldsymbol{s_{high}}$. Also, since $\boldsymbol{r_{high}}$ is already shifted, $\boldsymbol{o_{high}}$ is also already shifted. This means that now, instead of `OR`ing $\boldsymbol{o_{high}}$ and $\boldsymbol{o_{low}}$ and then shifting the result to the right, we just shift $\boldsymbol{o_{low}}$ to the right and `OR` it with this already shifted version of $\boldsymbol{o_{high}}$.

[11] With the same annoying exception of the leftmost nibble.

[12] Except unlike the $\boldsymbol{o}$ we got by shifting $\boldsymbol{x}$, here we are only flagging actual overflows, not ones caused only by carry-ins. This makes no practical difference, though.

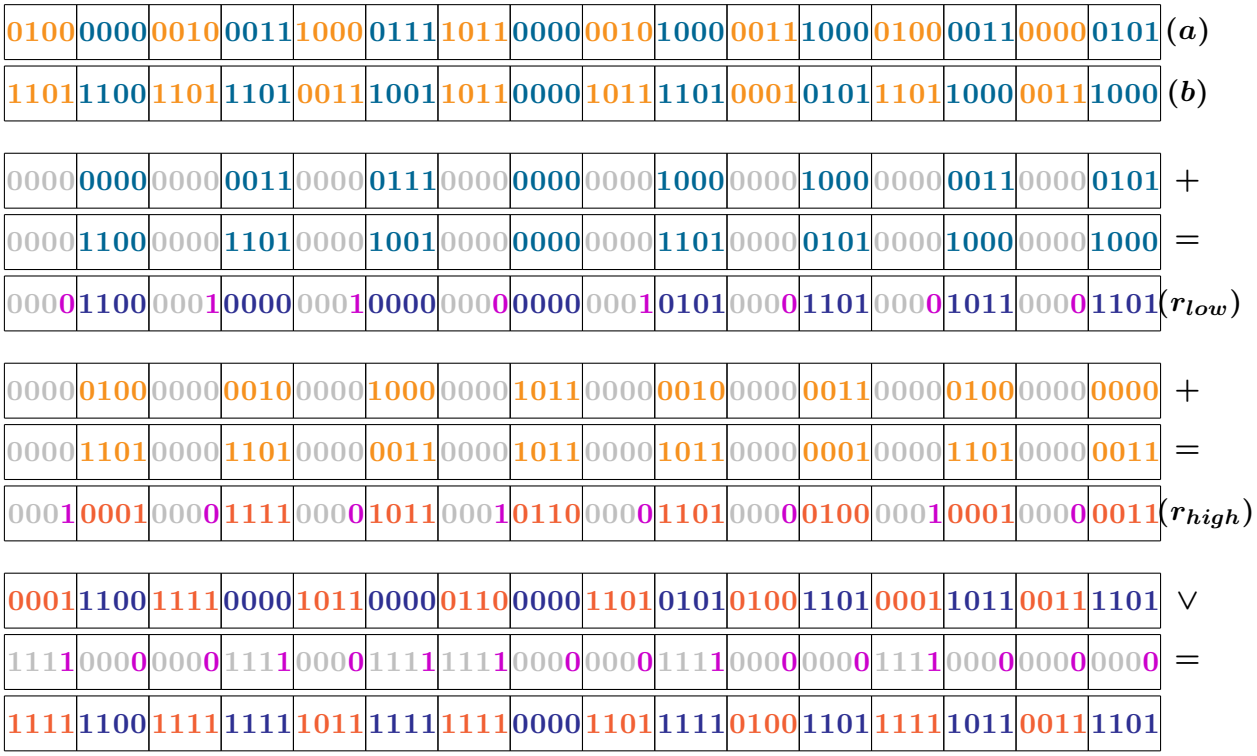

Figure 3.2: Nibble-wise saturating addition of two 64-bit register holding sixteen 4-bit values, using the high-low nibble method. The least significant bit of every inactive nibble of $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$, that detects overflows, is highlighted in purple. Here, before we compute $\boldsymbol{r_{high}}$, we shift the registers to the right. The last three lines show how we get $\boldsymbol{s_s}$ by `OR`ing together $\boldsymbol{s}$ (first operand) and $\boldsymbol{f}$ (second operand), where $\boldsymbol{f = o \cdot 15}$, like before.

The following code is a possible implementation of this algorithm in C++.

```
1 uint4x16_t vqadd_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t a = _a.reg, b = _b.reg;
4     uint64_t r_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F));
5     uint64_t r_high = (((a >> 4) & 0x0F0F0F0F0F0F0F0F) + ((b >> 4) &
    0x0F0F0F0F0F0F0F0F));
6     uint64_t f = ((r_high & 0xF0F0F0F0F0F0F0F0) | ((r_low &
    0xF0F0F0F0F0F0F0F0) >> 4)) * 15;
7     return ((r_high & 0x0F0F0F0F0F0F0F0F) << 4)| (r_low &
    0x0F0F0F0F0F0F0F0F) | f;
8 }
```

Listing 3.7: Implementation of `vqadd_u4` using the high-low nibble method.

3.7 Saturating Subtraction with the High-Low Nibble Method

The key difference between the high-low nibble method for addition and its variation for subtraction is that here we preset the inactive nibbles in $\boldsymbol{a}$ to **1111**. This means that when computing $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$, these inactive nibbles will either stay **1111** or turn into **1110** if a borrow-out occurs.

We now want to add saturating behavior to the high-low nibble method for subtraction. To avoid having to deal with the leftmost nibble, we can shift the inputs before we compute $\boldsymbol{r_{high}}$, and then shift the pieces back afterward, exactly like we just did for addition.

Since having a zero in the least significant bit of an inactive nibble in $\boldsymbol{r_{low}}$ or $\boldsymbol{r_{high}}$ means that the nibble immediately to the right had a borrow-out, this also means it underflowed. To compute $\boldsymbol{o_{low}}$ and $\boldsymbol{o_{high}}$, we cannot just mask out the active nibbles of $\boldsymbol{r_{low}}$ or $\boldsymbol{r_{high}}$ like before, though. We also have to invert the inactive ones. We can do that by first setting the active nibbles in $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$ to **1111**, and then inverting the whole results using a bitwise `NOT`.[13]

Now that we have our partial underflow indicators $\boldsymbol{o_{low}}$ and $\boldsymbol{o_{high}}$, we can compute $\boldsymbol{o}$ the same way we did for saturating addition with the high-low nibble method, and then compute $\boldsymbol{f = o \cdot 15}$. Since now we want to saturate downward, instead of `OR`ing the non-saturated result with $\boldsymbol{f}$, we `AND` it with $\neg\boldsymbol{f}$.

The following is a possible implementation of this algorithm in C++. Line 6 computes $\boldsymbol{o}$ applying De Morgan's laws: instead of computing $\boldsymbol{o_{high}}$ and $\boldsymbol{o_{low}}$ and `OR`ing them, we `AND` their inverses and bitwise `NOT` the result.

```
1 uint4x16_t vqsub_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t a = _a.reg, b = _b.reg;
4     uint64_t r_low = ((a | 0xF0F0F0F0F0F0F0F0) - (b &
    0x0F0F0F0F0F0F0F0F));
5     uint64_t r_high = (((a >> 4) | 0xF0F0F0F0F0F0F0F0) - ((b >> 4) &
    0x0F0F0F0F0F0F0F0F));
6     uint64_t o = ~((r_high | 0x0F0F0F0F0F0F0F0F) & ((r_low >> 4) |
    0xF0F0F0F0F0F0F0F0));
7     uint64_t f = o * 15;
8     return (((r_high & 0x0F0F0F0F0F0F0F0F) << 4) | (r_low &
    0x0F0F0F0F0F0F0F0F)) & ~f;
```

[13] Inverting inactive nibbles is necessary, because this way we can compute $\boldsymbol{f}$ as $\boldsymbol{o \cdot 15}$.

```
9 }
```

Listing 3.8: Implementation of `vqsub_u4` using the high-low nibble method.

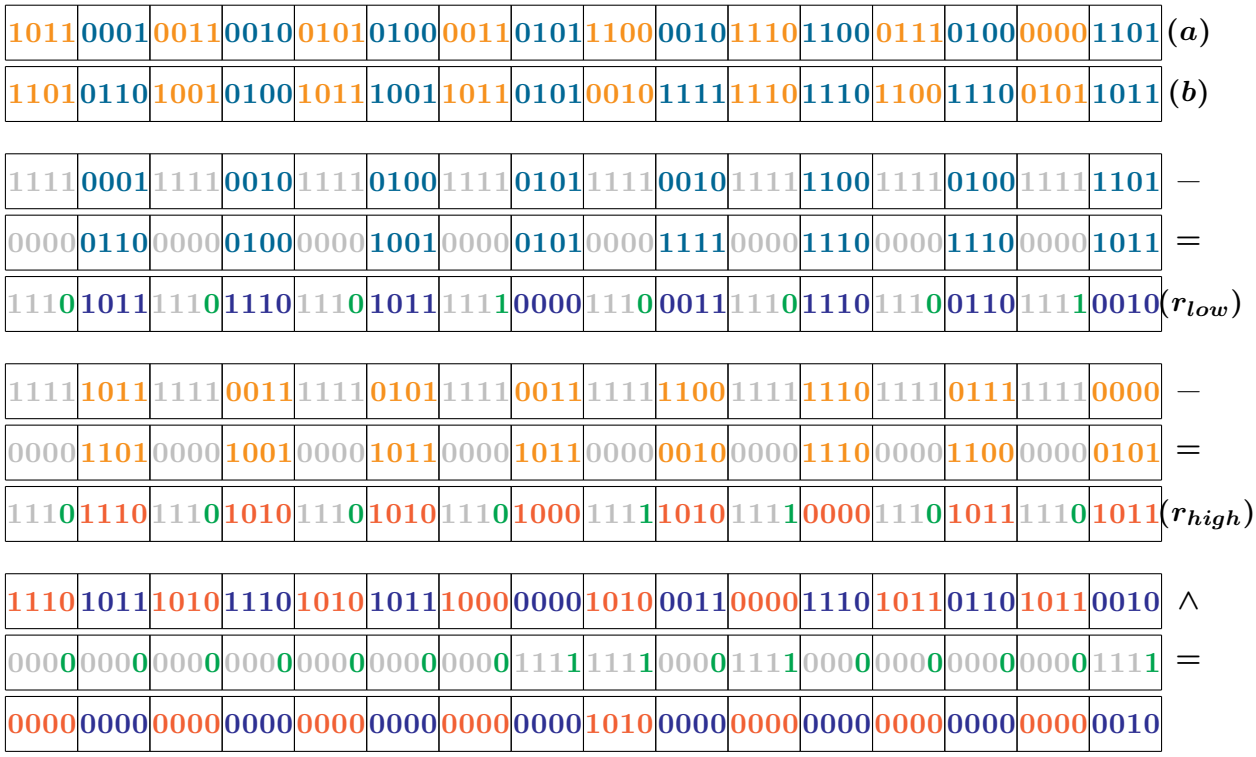

Figure 3.3: Nibble-wise saturating subtraction of two 64-bit register holding sixteen 4-bit values, using the high-low nibble method. The least significant bit of every inactive nibble of $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$, that detects underflows, is highlighted in green. Here, before we compute $\boldsymbol{r_{high}}$, we shift the registers to the right, to prevent having to deal with the leftmost nibble. The last three lines show how we get $\boldsymbol{d_s}$ by `AND`ing together $\boldsymbol{d}$ (first operand) and $\neg \boldsymbol{f}$ (second operand), where $\boldsymbol{f} = \boldsymbol{o} \cdot \mathbf{15}$.

3.8 Saturating Addition with the High Bit `XOR` Method

This method for saturating addition begins exactly like the regular high bit `XOR` method. We first add the inputs while masking out each nibble's high bit to prevent cross-nibble carry propagation. In doing so, the lower three bits of each nibble in the intermediate result $\boldsymbol{l}$ are the correct sum bits, while the high bit is the carry-in signal to that position during the addition. We then compute the high bit of the sums by simulating single-bit adders in software.

To achieve saturating behavior, we take this idea a step further by also simulating the carry-out signal of each nibble's high bit. Specifically, for a single-bit full adder with inputs $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C_{in}}$, the carry-out signal $\boldsymbol{C_{out}}$ can be computed as:

$$\boldsymbol{C_{out}} = (\boldsymbol{A} \wedge \boldsymbol{B}) \vee (\boldsymbol{A} \wedge \boldsymbol{C_{in}}) \vee (\boldsymbol{C_{in}} \wedge \boldsymbol{B})$$

Applying this formula bitwise to $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{l}$, we compute a value where the most significant bit of each nibble is set if and only if that nibble produced a carry-out—indicating an overflow. The remaining bits are irrelevant, so we mask them out, resulting in a value $\boldsymbol{y}$. Shifting $\boldsymbol{y}$ right by three positions and multiplying it by 15, we obtain a mask $\boldsymbol{f}$ that contains all ones (**1111**) in every nibble that overflowed. This mask functions exactly like the ones used in other saturating addition methods, and we can `OR` it with the non-saturated result $\boldsymbol{s}$[14] to produce the saturated output.

Below is an implementation of this method in C++.

```
uint4x16_t vqadd_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg;
    uint64_t l = (a & 0x7777777777777777) + (b & 0x7777777777777777);
    uint64_t z = (a ^ b) & 0x8888888888888888;
    uint64_t y = ((a & b) | (a & l) | (l & b)) & 0x8888888888888888;
    uint64_t f = (y >> 3) * 15;
    return (l ^ z) | f;
}
```

Listing 3.9: Implementation of `vqadd_u4` using the high bit `XOR` method.

3.9 Saturating Subtraction with the High Bit `XOR` Method

We can also add saturation to the high bit `XOR` method for subtraction. In the regular (non-saturating) subtraction, the lower three bits of each nibble in the intermediate difference $\boldsymbol{l}$ are the correct difference bits, while the high one is the *negated* borrow-in signal for that position during the subtraction. To compute the high bits, we simulated the behavior of single-bit subtractors in software.

To achieve saturating behavior, we also simulate the borrow-out signal ($\boldsymbol{B_{out}}$) of each nibble's high bit. For a single-bit full subtractor with inputs $\boldsymbol{A}$, $\boldsymbol{C}$, and $\boldsymbol{B_{in}}$, $\boldsymbol{B_{out}}$ can be

[14] $\boldsymbol{s} = \boldsymbol{l} \oplus \boldsymbol{z}$, as per the non-saturating high bit `XOR` method.

computed as:

$$\boldsymbol{B_{out}} = (\neg \boldsymbol{A} \wedge (\boldsymbol{C} \vee \boldsymbol{B_{in}})) \vee (\boldsymbol{A} \wedge \boldsymbol{C} \wedge \boldsymbol{B_{in}})$$

Applying this formula bitwise to $\boldsymbol{a}$, $\boldsymbol{b}$, and $\neg\boldsymbol{l}$,[15] we compute a value where, just like for addition, the high bit of each nibble is set if and only if that nibble produced a borrow-out (i.e., underflowed).

Again, we mask out the irrelevant bits, shift right by three positions, and multiply by 15 to obtain a mask $\boldsymbol{f}$ with all ones in every nibble that underflowed. Since we now want to saturate downward, we `AND` $\neg\boldsymbol{f}$ with $\boldsymbol{d}$[16] to produce the saturated difference.

The following is a possible implementation.

```
1 uint4x16_t vqsub_u4(uint4x16_t _a, uint4x16_t _b)
2 {
3     uint64_t a = _a.reg, b = _b.reg;
4     uint64_t l = (a | 0x8888888888888888) - (b & 0x7777777777777777);
5     uint64_t z = ~(a ^ b) & 0x8888888888888888;
6     uint64_t y = ((~(a) & (b | ~l)) | (a & b & ~l)) &
    0x8888888888888888;
7     uint64_t f = (y >> 3) * 15;
8     return (l ^ z) & (~f);
9 }
```

Listing 3.10: Implementation of `vsub_u4` using the high bit `XOR` method.

[15] Where the bits in $\boldsymbol{a}$ are $\boldsymbol{A}$, the bits in $\boldsymbol{b}$ are $\boldsymbol{C}$, and the bits in $\neg\boldsymbol{l}$ are $\boldsymbol{B_{in}}$. We negate $\boldsymbol{l}$ because its high bits encode the negated borrow-in signals, so negating it recovers the actual values.

[16] $\boldsymbol{d} = \boldsymbol{l} \oplus \boldsymbol{z}$, as per the the non-saturating high bit `XOR` method.

Chapter 4

Multiplication and Saturating Multiplication

4.1 Multiplication

The key to understanding the algorithms we will present in this chapter is to view unsigned 4-bit multiplication as the sum of four partial products. This way, we can adapt our addition algorithms and reuse them to perform the multiplication.

Let $\boldsymbol{n_1}$ and $\boldsymbol{n_2}$ be our 4-bit operands. To compute their product, we proceed just as we would when multiplying decimal numbers:

1. Multiply $\boldsymbol{n_1}$ by the least significant bit in $\boldsymbol{n_2}$, call this partial result $\boldsymbol{p}$.

2. Multiply $\boldsymbol{n_1}$ by the next bit in $\boldsymbol{n_2}$, shift the result 1 bit to the left, and add it to $\boldsymbol{p}$.

3. Multiply $\boldsymbol{n_1}$ by the second most significant bit in $\boldsymbol{n_2}$, shift the result 2 bits to the left, and add it to $\boldsymbol{p}$.

4. Multiply $\boldsymbol{n_1}$ by the most significant bit in $\boldsymbol{n_2}$, shift the result 3 positions to the left, and add it to $\boldsymbol{p}$.

After these four steps, $\boldsymbol{p}$ holds the final product. Since we are working with binary numbers, multiplying $\boldsymbol{n_1}$ by a single bit value $\boldsymbol{B}$ is trivial: if $\boldsymbol{B}$ is one, we do nothing; if $\boldsymbol{B}$ is zero, we zero out $\boldsymbol{n_1}$. So, just `AND`ing every bit in $\boldsymbol{n_1}$ with $\boldsymbol{B}$ does the trick. This means that for each step $\boldsymbol{i}$ of the algorithm, we have to `AND` the whole value $\boldsymbol{n_1}$ with the $\boldsymbol{i}^{\text{th}}$ bit in $\boldsymbol{n_2}$,[1] counting from the right.

[1] Assuming $\boldsymbol{i}$ is zero based (i.e. the first iteration is the zeroeth one).

An x86-64 processor cannot just `AND` a whole 4-bit value with a lone bit, though. We need to build a bitmask that, for each iteration, has copies of that $\boldsymbol{i}^{\text{th}}$ bit of $\boldsymbol{n_2}$ in all four positions. This way, we can now compute each of the four partial single-bit multiplications by just bitwise `AND`ing the mask we calculated in the $\boldsymbol{i}^{\text{th}}$ iteration with $\boldsymbol{n_1}$.

To compute this mask, we can, at iteration $\boldsymbol{i}$, shift $\boldsymbol{n_2}$ by $\boldsymbol{i}$ bits to the right, mask out all bits but the least significant one (which, since we shifted $\boldsymbol{n_2}$, is now the $\boldsymbol{i}^{\text{th}}$ bit of the original $\boldsymbol{n_2}$), and multiply it by 15.

We can now apply this idea to compute the vectorized nibble-wise multiplication of 16 different 4-bit values.

4.2 Multiplication with the High-Low Nibble Method

Our goal is to multiply each nibble in $\boldsymbol{a}$ by the corresponding nibble in $\boldsymbol{b}$. We can adapt the 4-bit multiplication routine from above, with the difference that now, for each step $\boldsymbol{i}$, instead of multiplying a single 4-bit number by the $\boldsymbol{i}^{\text{th}}$ bit of the other, we now want to multiply *every* nibble in $\boldsymbol{a}$ by the $\boldsymbol{i}^{\text{th}}$ bit of the corresponding nibble in $\boldsymbol{b}$. We then have to shift the partial product in each nibble left by $\boldsymbol{i}$ bits and add them to the previous partial results, just like before, but we now have to prevent bits and carry-ins from crossing nibble boundaries.

We will start our implementation with adapting the four single-bit multiplications to make them work with sixteen nibbles packed in a 64-bit register, as opposed to a single nibble. It turns out, we can compute them almost exactly the same way as before: for each iteration, we shift $\boldsymbol{b}$ right by $\boldsymbol{i}$ bits, keep only the least significant bit of each nibble (which corresponds to the original $\boldsymbol{i}^{\text{th}}$ bit of each nibble), and multiply the result by 15 to replicate that single bit across all four positions in each nibble. We will call this mask $\boldsymbol{m_i}$. `AND`ing $\boldsymbol{m_i}$ with $\boldsymbol{a}$ yields an intermediate value $\boldsymbol{k_i}$ that has, in each nibble, either zero or the original nibble from $\boldsymbol{a}$.[2] This is exactly what we want for vectorized single-bit multiplication.

The problem is that here we cannot just shift $\boldsymbol{k_i}$ by $\boldsymbol{i}$ bits to the left and add it to the running total, because shifting would make bits spill across nibble boundaries, and also the additions can alter the result due to carry bits.

To solve this, the trick we are using is very similar to the one in the high-low nibble method for addition—hence the shared name. If we mask out every other nibble in $\boldsymbol{k_i}$, the remaining *active* nibbles get four bits to their left for potential overflows,[3] without

[2] The subscript $\boldsymbol{i}$ here just indicates that we compute a different $\boldsymbol{k}$ and $\boldsymbol{m}$ at each step.

[3] Except for the leftmost nibble—though it does not matter since there is no nibble to its left.

affecting the other results. Since the largest possible product of two 4-bit unsigned integers is $\mathbf{15 \times 15 = 225 \leq 255}$, 8 bits are enough to safely hold our intermediate values.

We can take advantage of this by computing the sums of the partial multiplications for low and high nibbles separately, with no risk of overflows altering our results. We store the intermediate sums of low nibbles in $\boldsymbol{p_{low}}$, and the high ones in $\boldsymbol{p_{high}}$. In each iteration, we shift the low nibbles of $\boldsymbol{k_i}$ by $\boldsymbol{i}$ positions to the left and add them to $\boldsymbol{p_{low}}$. We also shift the high nibbles and add them to $\boldsymbol{p_{high}}$. After four iterations, $\boldsymbol{p_{low}}$ holds the final products for the low nibbles, and $\boldsymbol{p_{high}}$ for the high nibbles. These products are stored on 8 bits, but we want our results to be contained in a single nibble, so we discard the higher 4 bits of each product, essentially computing modulo-16 multiplications. To do that, we combine $\boldsymbol{p_{low}}$ and $\boldsymbol{p_{high}}$ together, masking out the high nibbles of $\boldsymbol{p_{low}}$ and the low nibbles of $\boldsymbol{p_{high}}$ and finally `OR`ing the two halves together.

Here is a direct C++ implementation.

```
uint4x16_t vmul_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, p_low = 0, p_high = 0, k;
    for (int i = 0; i < 4; i++)
    {
        k = a & (((b >> i) & 0x1111111111111111) * 15);
        p_low += (k & 0x0F0F0F0F0F0F0F0F) << i;
        p_high += (k & 0xF0F0F0F0F0F0F0F0) << i;
    }
    return (p_low & 0x0F0F0F0F0F0F0F0F) | (p_high & 0xF0F0F0F0F0F0F0F0);
}
```

Listing 4.1: Implementation of `vmul_u4` using the high-low nibble method, with a loop to iterate through the four partial products.

This implementation uses a for loop to iterate through the four partial products. While functional, the loop introduces conditional instructions and the overhead of managing the loop variable $\boldsymbol{i}$. We can avoid that by unrolling the loop and replacing the iterative process with explicitly written instructions for each iteration. This results in the following code.

```
uint4x16_t vmul_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, p_low, p_high, k;

    k = a & ((b & 0x1111111111111111) * 15);
    p_low = k & 0x0F0F0F0F0F0F0F0F;
    p_high = k & 0xF0F0F0F0F0F0F0F0;

```

```
9     k = a & (((b >> 1) & 0x1111111111111111) * 15);
10    p_low += (k & 0x0F0F0F0F0F0F0F0F) << 1;
11    p_high += (k & 0xF0F0F0F0F0F0F0F0) << 1;
12
13    k = a & (((b >> 2) & 0x1111111111111111) * 15);
14    p_low += (k & 0x0F0F0F0F0F0F0F0F) << 2;
15    p_high += (k & 0xF0F0F0F0F0F0F0F0) << 2;
16
17    k = a & (((b >> 3) & 0x1111111111111111) * 15);
18    p_low += (k & 0x0F0F0F0F0F0F0F0F) << 3;
19    p_high += (k & 0xF0F0F0F0F0F0F0F0) << 3;
20
21    return (p_low & 0x0F0F0F0F0F0F0F0F) | (p_high & 0xF0F0F0F0F0F0F0F0);
22 }
```

Listing 4.2: Implementation of `vmul_u4` using the high-low nibble method, after unrolling the loop.

We can actually push the optimization a bit further. In the code above, we shift $\boldsymbol{b}$ to the right by $\boldsymbol{i}$ bits at each step to compute each mask $\boldsymbol{m_i}$, we use this mask to compute $\boldsymbol{k_i}$, and then later shift $\boldsymbol{k_i}$ back to the left when adding its high and low nibbles to $\boldsymbol{p_{low}}$ and $\boldsymbol{p_{high}}$, respectively.

We can save some shifts by computing an already shifted version of $\boldsymbol{k_i}$. Instead of shifting $\boldsymbol{b}$ to the right and isolating the least significant bit of each nibble, we directly isolate the $\boldsymbol{i}^{\text{th}}$ bit of each nibble.[4] After that, when we multiply this result by 15, the resulting nibble mask $\boldsymbol{m_i}$ will no longer be aligned with the original nibbles in $\boldsymbol{a}$. To fix this, we shift $\boldsymbol{a}$ to the left, aligning its nibbles with the new mask. Finally, computing $\boldsymbol{a} \wedge \boldsymbol{m_i}$ gives us a slightly altered $\boldsymbol{k_i}$ that is already shifted correctly.

We can do even better, though. As we shift $\boldsymbol{a}$, the higher bits of each nibble spill into the next nibble to the left. These spilling bits are useless because they cannot alter the lower 4 bits (the only ones we are considering).

Instead of masking them out later, we avoid replicating them into $\boldsymbol{k_i}$ in the first place.

- When isolating bit 0, no spill happens, so we copy the bit into all 4 positions of each nibble to compute $\boldsymbol{m_0}$ by multiplying by 15 (**1111** in binary).

- When isolating bit 1, since we are now shifting $\boldsymbol{a}$ to the left, the top bit of each nibble would cross a nibble boundary. To prevent that, we multiply by 7 (**111** in binary) instead. The masks stay contained inside each nibble—either **1110** if the

[4] We can just `AND` $\boldsymbol{b}$ with a different mask at each step that has only the $\boldsymbol{i}^{\text{th}}$ bit of each nibble set.

isolated bit is set, or **0000** otherwise—and the bits of $\boldsymbol{a}$ that cross a nibble boundary will not be replicated in $\boldsymbol{k_1}$.

- Similarly, for bit 2, we multiply by 3 (**11** in binary).
- For bit 3, we do not need to multiply at all.

This is important for two reasons. First, it saves an unnecessary multiplication in the last step. Second, it allows us to use the same two masks at every step when splitting the low and high nibbles of $\boldsymbol{k_i}$—which are now already shifted—to add them to $\boldsymbol{p_{low}}$ and $\boldsymbol{p_{high}}$. Without this optimization, we would have needed a different mask at every step because the nibbles in $\boldsymbol{k_i}$ are no longer aligned with the *physical* nibbles, and so we would have had to mask out different portions of it at each step to separate the high and low nibbles. This could be bad for performance because every new mask would have to be loaded into a register with a `movabsq` instruction.

With this optimization our partial multiplications—even though already shifted and not aligned with the physical nibbles—do not spill into the nibble to their left. That is because we did not replicate the bits in $\boldsymbol{a}$ that spilled out of their original nibble. This means that we can reuse the same masks at each step, and the compiler can keep them in a register.

The following code implements these optimizations.

```
1  uint4x16_t vmul_u4(uint4x16_t _a, uint4x16_t _b)
2  {
3      uint64_t a = _a.reg, b = _b.reg, p_low, p_high, k;
4
5      k = a & ((b & 0x1111111111111111) * 15);
6      p_low = k & 0x0F0F0F0F0F0F0F0F;
7      p_high = k & 0xF0F0F0F0F0F0F0F0;
8
9      a <<= 1;
10     k = a & ((b & 0x2222222222222222) * 7);
11     p_low += k & 0x0F0F0F0F0F0F0F0F;
12     p_high += k & 0xF0F0F0F0F0F0F0F0;
13
14     a <<= 1;
15     k = a & ((b & 0x4444444444444444) * 3);
16     p_low += k & 0x0F0F0F0F0F0F0F0F;
17     p_high += k & 0xF0F0F0F0F0F0F0F0;
18
19     a <<= 1;
```

```
    k = a & (b & 0x8888888888888888);
    p_low += k & 0x0F0F0F0F0F0F0F0F;
    p_high += k & 0xF0F0F0F0F0F0F0F0;

    return (p_low & 0x0F0F0F0F0F0F0F0F) | (p_high & 0xF0F0F0F0F0F0F0F0);
}
```

Listing 4.3: Implementation of `vmul_u4` using the high-low nibble method after unrolling the loop and implementing optimizations.

4.3 Multiplication with the High Bit `XOR` Method

Instead of computing the sums of our partial products by splitting high and low nibbles with the high-low nibble method, we can add them together using the faster high bit `XOR` method. This is possible thanks to the last optimizations we made: the nibbles in the partial products, $\boldsymbol{k_i}$, now cleanly hold the partial modulo-16 single-bit product for each nibble, already in the right position and without any unwanted bits spilling in from the nibble to the right.

So now, we just need to maintain a nibble-wise running total $\boldsymbol{p}$, and at each step, we add $\boldsymbol{k_i}$ to $\boldsymbol{p}$ nibble-wise using the high bit `XOR` method.

Here is the C++ code.

```
uint4x16_t vmul_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, p, k, l, z;

    p = a & ((b & 0x1111111111111111) * 15);

    a <<= 1;
    k = a & ((b & 0x2222222222222222) * 7);
    l = (k & 0x7777777777777777) + (p & 0x7777777777777777);
    z = (k ^ p) & 0x8888888888888888;
    p = l ^ z;

    a <<= 1;
    k = a & ((b & 0x4444444444444444) * 3);
    l = (k & 0x7777777777777777) + (p & 0x7777777777777777);
    z = (k ^ p) & 0x8888888888888888;
    p = l ^ z;

    a <<= 1;
```

```
20      k = a & (b & 0x8888888888888888);
21      p = p ^ k;
22
23      return p;
24  }
```

Listing 4.4: Implementation of `vmul_u4` using the high bit `XOR` method.

Notice that in the last step, when we compute $\boldsymbol{k_3}$, we just `XOR` it directly with $\boldsymbol{p}$ to get our final result. That is because $\boldsymbol{k_3}$ only has nonzero bits in the highest bit of each nibble.

4.4 Saturating Multiplication with the High-Low Nibble Method

To implement saturating behavior, we can start with the unoptimized version of regular multiplication using the high-low nibble method. In that algorithm, before we mask $\boldsymbol{p_{low}}$ and $\boldsymbol{p_{high}}$ to combine them, they basically hold the full 8-bit products—except for the leftmost product in $\boldsymbol{p_{high}}$, where its higher four bits are lost.

We can use those higher four bits, which we previously just masked out, to implement saturating behavior. After computing the sums of the four partial products, for each lower (active) nibble in $\boldsymbol{p_{low}}$, if the nibble to its left has a value of **0000**, it means that the product does not overflow. But if it is set to any other value, it means that the product cannot fit in just four bits, and so we have to saturate the nibble to **1111**.

The same applies to $\boldsymbol{p_{high}}$, but here, to avoid losing information about the leftmost nibble, we compute a modified $\boldsymbol{p_{high}}$ that is shifted four bits to the right. To do that, at each step, we simply shift $\boldsymbol{k_i}$ to the right before we mask out the inactive nibbles and add the active ones to $\boldsymbol{p_{high}}$, essentially using a similar trick to the one we used when computing saturating addition with the high-low nibble method.

After the four partial single-bit multiplication steps, we end up with:

- A value $\boldsymbol{p_{low}}$ that has, in the low nibbles, the lower 4 bits of the products of the nibbles that were in those low nibbles, and in the high nibbles the higher four bits of those same products. The products in the low nibbles are aligned with where we want them to end up, while the high nibbles—which we now want to use for saturation—are in the nibble to their left.

- A value $\boldsymbol{p_{high}}$ that has, again in the *low* nibbles, the lower 4 bits of the products of the nibbles that were originally in the high nibbles, and in the high nibbles the

higher four bits of those same products. The products in the low nibbles are now shifted 4 bits to the right, while the high nibbles are where we want the products to end up.

This means that now, if we mask out the low nibbles of $\boldsymbol{p_{high}}$, mask out the low nibbles of $\boldsymbol{p_{low}}$, shift the masked $\boldsymbol{p_{low}}$ four bits to the right, and `OR` the two halves together, we get an indicator, which we can call $\boldsymbol{o}$, that has, in each nibble, either **0000** if the product in that nibble can fit within 4 bits, or any other nonzero value if it cannot.

To turn this indicator into a mask that can be `OR`ed with the non-saturated product $\boldsymbol{p}$ to saturate it, we can use the following piece of C++ code.

```
o |= (o >> 1);
o |= (o >> 2);
o &= 0x1111111111111111;
o *= 15;
```

This code alters each nibble in $\boldsymbol{o}$ so that if it contains at least one bit set to 1, the whole nibble becomes **1111**. If all the bits are zero, the nibble stays zero. Putting it all together, we can implement saturating multiplication using the following C++ function.

```
uint4x16_t vqmul_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, p_low, p_high, k, o, f;

    k = a & ((b & 0x1111111111111111) * 15);
    p_low = k & 0x0F0F0F0F0F0F0F0F;
    p_high = (k >> 4) & 0x0F0F0F0F0F0F0F0F;

    k = a & (((b >> 1) & 0x1111111111111111) * 15);
    p_low += (k & 0x0F0F0F0F0F0F0F0F) << 1;
    p_high += ((k >> 4) & 0x0F0F0F0F0F0F0F0F) << 1;

    k = a & (((b >> 2) & 0x1111111111111111) * 15);
    p_low += (k & 0x0F0F0F0F0F0F0F0F) << 2;
    p_high += ((k >> 4) & 0x0F0F0F0F0F0F0F0F) << 2;

    k = a & (((b >> 3) & 0x1111111111111111) * 15);
    p_low += (k & 0x0F0F0F0F0F0F0F0F) << 3;
    p_high += ((k >> 4) & 0x0F0F0F0F0F0F0F0F) << 3;

    o = (p_high & 0xF0F0F0F0F0F0F0F0) | ((p_low & 0xF0F0F0F0F0F0F0F0)
   >> 4);
    o |= (o >> 1);
```

```
23      o |= (o >> 2);
24      o &= 0x1111111111111111;
25      f = o * 15;
26
27      return ((p_low & 0x0F0F0F0F0F0F0F0F) | ((p_high &
        0x0F0F0F0F0F0F0F0F) << 4)) | f;
28  }
```

Listing 4.5: Implementation of `vqmul_u4` using the high-low nibble method.

We can actually optimize this code and save some operations by computing two additional values $\boldsymbol{a_{low}}$ (by masking out the high nibbles in $\boldsymbol{a}$) and $\boldsymbol{a_{high}}$ (by masking out the low nibbles and shifting the result four bits to the right) *before* we start the actual multiplication.

This way, at each step, instead of computing $\boldsymbol{k_i}$ (the original, non-shifted $\boldsymbol{k_i}$) and then masking out alternately the low and high nibbles, we compute our mask $\boldsymbol{m_i}$, `AND` it with $\boldsymbol{a_{low}}$, and add the result—which we can call $\boldsymbol{k_{low}}$, since it is exactly what we would get by masking out the high nibbles in $\boldsymbol{k_i}$—to $\boldsymbol{p_{low}}$. We then shift $\boldsymbol{m_i}$ four bits to the left, `AND` it with $\boldsymbol{a_{high}}$, and add that result—$\boldsymbol{k_{high}}$, which is what we would get by masking out the low bits of $\boldsymbol{k_i}$ and shifting right by four bits—to $\boldsymbol{p_{high}}$.

Basically, instead of masking out low and high nibbles in $\boldsymbol{k_i}$ at every step, we achieve the same effect by `AND`ing $\boldsymbol{m_i}$ with both pre-masked (and shifted in the case of $\boldsymbol{a_{high}}$) versions of $\boldsymbol{a}$. We then add the results to $\boldsymbol{p_{high}}$ and $\boldsymbol{p_{low}}$ and proceed exactly as before to saturate the result. The following is a possible implementation of this optimized algorithm.

```
1  uint4x16_t vqmul_u4(uint4x16_t _a, uint4x16_t _b)
2  {
3      uint64_t a = _a.reg, b = _b.reg, p_low, p_high, o, f, a_low,
       a_high, m;
4      a_low = a & 0x0F0F0F0F0F0F0F0F;
5      a_high = (a >> 4) & 0x0F0F0F0F0F0F0F0F;
6
7      m = (b & 0x1111111111111111) * 15;
8      p_low = a_low & m;
9      p_high = a_high & (m >> 4);
10
11     m = ((b >> 1) & 0x1111111111111111) * 15;
12     p_low += (a_low & m) << 1;
13     p_high += (a_high & (m >> 4)) << 1;
14
15     m = ((b >> 2) & 0x1111111111111111) * 15;
16     p_low += (a_low & m) << 2;
```

```
17      p_high += (a_high & (m >> 4)) << 2;
18
19      m = ((b >> 3) & 0x1111111111111111) * 15;
20      p_low += (a_low & m) << 3;
21      p_high += (a_high & (m >> 4)) << 3;
22
23      o = (p_high & 0xF0F0F0F0F0F0F0F0) | ((p_low & 0xF0F0F0F0F0F0F0F0)
    >> 4);
24      o |= (o >> 1);
25      o |= (o >> 2);
26      o &= 0x1111111111111111;
27      f = o * 15;
28
29      return ((p_low & 0x0F0F0F0F0F0F0F0F) | ((p_high &
    0x0F0F0F0F0F0F0F0F) << 4)) | f;
30  }
```

Listing 4.6: Implementation of `vqmul_u4` using the high-low nibble method, after implementing the optimization.

Chapter 5

Vectorized Operations with a Lookup Table

5.1 The Lookup Table Approach

An interesting alternative to the vectorized algorithms discussed in the previous chapters is to use a lookup table that stores the results of the operation we want to perform. Since we are working with 4-bit data types, we can precompute the results for every possible input combination—$\mathbf{2^4 \times 2^4 = 256}$ total combinations. We can then just access the precomputed values to get our results. A different lookup table is needed for each operation.

In practice, we have to build a 256-entry table, with each entry holding the 4-bit result for a combination of inputs. Assuming we are storing the table in memory, accessing the right entry simply involves reading from a different memory location determined by the two 4-bit inputs. Since each entry fits in a nibble, the table is tiny—just 128 bytes total—easily fitting into the smallest (and fastest) L1 cache. This approach, however, still requires a separate read for every 4-bit result we compute.

To reduce the number of memory accesses, we can optimize the lookup table so that each entry stores two 4-bit results packed into a single byte. Essentially, instead of computing each nibble independently, we take two full 8-bit values (each holding two 4-bit inputs) and precompute the corresponding 8-bit result (that is basically composed of the two 4-bit results) for every possible pair. This effectively lets us cut the number of memory accesses in half, but it also makes the table significantly larger at $\mathbf{2^8 \times 2^8 = 65{,}536}$ entries, each one a byte, for a total of 64 KiB.

The processor we are using to run our tests, an Intel i7-6700HQ, has a 32 KiB L1 data

cache for each physical core. This means only half of this larger table can reside in L1 at a time, with the remainder spilling into L2. However, many modern CPUs come with larger L1 caches, making this approach potentially even more beneficial on newer hardware.

Building the table with even bigger inputs, say 12 bits (three nibbles), is not feasible—that would require a 24 MiB table, too large to fit even in the largest L3 cache.

5.2 Lookup Table Implementation with 8-bit Entries

We can start with implementing the lookup table approach where each entry in the table stores two 4-bit results. This version is simpler because we can operate directly on full bytes rather than individual nibbles. We will structure the function to take two `uint4x16_t` inputs and return another `uint4x16_t`—just like our vectorized algorithms described in the previous chapters. This means that this function computes sixteen 4-bit results, one for each pair of corresponding nibbles in our inputs $\boldsymbol{a}$ and $\boldsymbol{b}$. We assume that `table` is a global `uint8_t` array that has already been initialized with the correct values for whatever operation we are computing.

```
uint4x16_t table_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint8_t* a = (uint8_t*)&_a;
    uint8_t* b = (uint8_t*)&_b;
    uint64_t _r;
    uint8_t* r = (uint8_t*)&_r;
    r[0] = table[a[0] * 256 + b[0]];
    r[1] = table[a[1] * 256 + b[1]];
    r[2] = table[a[2] * 256 + b[2]];
    r[3] = table[a[3] * 256 + b[3]];
    r[4] = table[a[4] * 256 + b[4]];
    r[5] = table[a[5] * 256 + b[5]];
    r[6] = table[a[6] * 256 + b[6]];
    r[7] = table[a[7] * 256 + b[7]];
    return _r;
}
```

Listing 5.1: Implementation of the lookup table-based approach, where each entry in the table stores two results.

The downside of this implementation is that we are reading from and writing to memory for every byte of the input and output, one byte at a time. This is exactly what we have been trying to avoid with our vectorized functions. To prevent it, we can extract each byte directly from the packed 64-bit integers using bitmasks and shifts, and combine results

using bitwise ORs. This lets the compiler keep the inputs, $\boldsymbol{a}$ and $\boldsymbol{b}$, and the output, $\boldsymbol{r}$, within registers, with memory access limited to table lookups. The result is the following optimized implementation.

```
uint4x16_t table_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, r;
    r = (uint64_t)table[((a & 0xFFull) << 8) | (b & 0xFFull)];
    r |= (uint64_t)table[(((a & 0xFF00ull) << 8) | (b & 0xFF00ull)) >>
    8] << 8;
    r |= (uint64_t)table[(((a & 0xFF0000ull) << 8) | (b & 0xFF0000ull))
    >> 16] << 16;
    r |= (uint64_t)table[(((a & 0xFF000000ull) << 8) | (b &
    0xFF000000ull)) >> 24] << 24;
    r |= (uint64_t)table[(((a & 0xFF00000000ull) << 8) | (b &
    0xFF00000000ull)) >> 32] << 32;
    r |= (uint64_t)table[(((a & 0xFF0000000000ull) << 8) | (b &
    0xFF0000000000ull)) >> 40] << 40;
    r |= (uint64_t)table[(((a & 0xFF000000000000ull) << 8) | (b &
    0xFF000000000000ull)) >> 48] << 48;
    r |= (uint64_t)table[(((a & 0xFF00000000000000ull)) | ((b &
    0xFF00000000000000ull) >> 8)) >> 48] << 56;
    return r;
}
```

Listing 5.2: Optimized implementation of the lookup table-based approach, where each entry in the table stores two results.

5.3 Lookup Table Implementation with 4-bit Entries

As discussed earlier, using 8-bit entries is great because it lets us reduce the number of memory accesses. On older or lower-end CPUs, though, the larger table cannot fully fit in the L1 cache.

To take full advantage of this faster cache—even at the cost of more accesses—we can adapt our previous lookup table implementation to use a smaller table where each entry stores just one result. This means each entry is now only 4 bits in size, and we could theoretically fit two of them in a single byte. However, to make accessing the entries simpler, instead of packing two entries in the same byte, we store each one in the low nibble of a different byte and set the high nibble to zero. As a result, the table occupies 256 bytes rather than the theoretical 128 we mentioned earlier, but it can still easily fit in

the L1 cache.

Here is an implementation that uses this approach.

```
uint4x16_t table_u4(uint4x16_t _a, uint4x16_t _b)
{
    uint64_t a = _a.reg, b = _b.reg, r;
    r = (uint64_t)table[((a & 0xFull) << 4) | (b & 0xFull)];
    r |= (uint64_t)table[(a & 0xF0ull) | ((b & 0xF0ull) >> 4)] << 4;
    r |= (uint64_t)table[((a & 0xF00ull) >> 4) | ((b & 0xF00ull) >> 8)]
   << 8;
    r |= (uint64_t)table[((a & 0xF000ull) >> 8) | ((b & 0xF000ull) >>
   12)] << 12;
    r |= (uint64_t)table[((a & 0xF0000ull) >> 12) | ((b & 0xF0000ull)
   >> 16)] << 16;
    r |= (uint64_t)table[((a & 0xF00000ull) >> 16) | ((b & 0xF00000ull)
   >> 20)] << 20;
    r |= (uint64_t)table[((a & 0xF000000ull) >> 20) | ((b &
   0xF000000ull) >> 24)] << 24;
    r |= (uint64_t)table[((a & 0xF0000000ull) >> 24) | ((b &
   0xF0000000ull) >> 28)] << 28;
    r |= (uint64_t)table[((a & 0xF00000000ull) >> 28) | ((b &
   0xF00000000ull) >> 32)] << 32;
    r |= (uint64_t)table[((a & 0xF000000000ull) >> 32) | ((b &
   0xF000000000ull) >> 36)] << 36;
    r |= (uint64_t)table[((a & 0xF0000000000ull) >> 36) | ((b &
   0xF0000000000ull) >> 40)] << 40;
    r |= (uint64_t)table[((a & 0xF00000000000ull) >> 40) | ((b &
   0xF00000000000ull) >> 44)] << 44;
    r |= (uint64_t)table[((a & 0xF000000000000ull) >> 44) | ((b &
   0xF000000000000ull) >> 48)] << 48;
    r |= (uint64_t)table[((a & 0xF0000000000000ull) >> 48) | ((b &
   0xF0000000000000ull) >> 52)] << 52;
    r |= (uint64_t)table[((a & 0xF00000000000000ull) >> 52) | ((b &
   0xF00000000000000ull) >> 56)] << 56;
    r |= (uint64_t)table[((a & 0xF000000000000000ull) >> 56) | ((b &
   0xF000000000000000ull) >> 60)] << 60;
    return r;
}
```

Listing 5.3: Implementation of the lookup table-based approach, where each entry in the table stores one results.

Chapter 6

Performance Analysis and Comparisons

6.1 Performance of Vectorized Algorithms

We will start our performance analysis by comparing different versions of each operation. To do this, we wrote a benchmark function that initializes two 32 MiB data blocks—roughly 65 million nibbles per data block—with random values. It then performs the operation on each pair of corresponding nibbles in those data blocks, processing 16 nibble pairs at a time by calling the function we want to evaluate.

We measure the execution time using the class `std::chrono::high_resolution_clock`, as shown in the code snippet below.

```
auto start0 = high_resolution_clock::now();
for(int i = 0; i < (1 << 22); i++)
{
    ((uint4x16_t*)data0)[i] =
    op(((uint4x16_t*)i1)[i], ((uint4x16_t*)i2)[i]);
}
auto end0 = high_resolution_clock::now();
```

In this code:

- The variables `i1` and `i2` are void pointers to the initialized random data.
- The variable `data0` is a void pointer to a 32 MiB allocated memory region that will store our results.
- The function `op` is the vectorized algorithm being evaluated.

The recorded execution time includes loop and function call overhead. To ensure more reliable measurements, we pinned the process to a specific CPU core, reducing the impact of context switches. The full benchmark function is reported in the appendix.

We ran our benchmark function 1,001 times for each algorithm and took the median result. The system used to perform these tests has an Intel i7-6700HQ processor, 16 GiB of DDR4 RAM operating at 1,200 MHz, and runs Ubuntu 22.04.5. The C++ code was compiled using GCC 11.4.0 with the optimization flag `-O2`. Here is what we found in our performance analysis.

6.1.1 Addition

For addition, we compared the two implementations of the double `XOR` algorithms: one that checks the exit condition every two iterations (V1) and one that checks it at every iteration (V2), the branchless double `XOR` implementation (reported in the final section), the high-low nibble method, and the high bit `XOR` method.

Algorithm	Median Time (µs)
Double `XOR` V1	11,715
Double `XOR` V2	27,747
Branchless Double `XOR`	11,251
High-Low Nibble	9,721
High Bit `XOR`	8,669

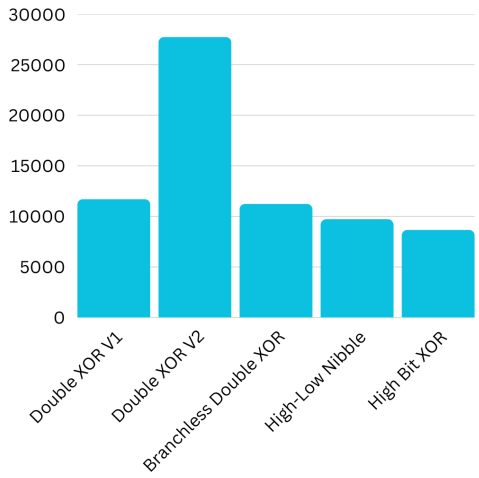

Figure 6.1: Performance of addition algorithms.

6.1.2 Saturating Addition

For saturating addition, we compared our implementation of the double `XOR` method, the two variants of the hybrid double `XOR`/high bit `XOR` method (where the branchless version is implemented in Assembly), and the high-low nibble method.

Algorithm	Median Time (µs)
Double `XOR`	45,421
Hybrid	32,297
Branchless Hybrid	13,524
High-Low Nibble	12,852
High Bit `XOR`	14,146

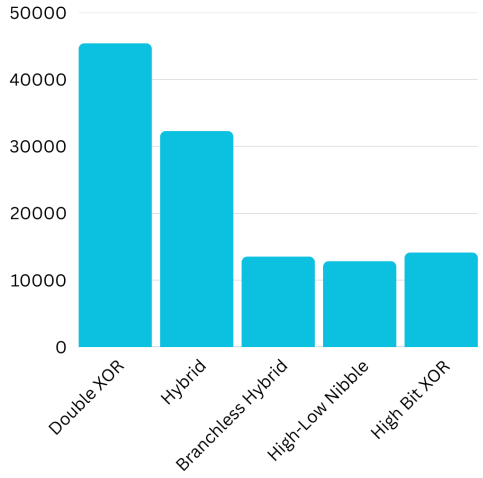

Figure 6.2: Performance of saturating addition algorithms.

6.1.3 Subtraction

For subtraction, we used the same algorithm variants as in the addition section. Here, the branchless double `XOR` method refers to the function that computes $\boldsymbol{x}$ and then adds it to $\boldsymbol{r}$ using the high-low nibble approach.

Algorithm	Median Time (µs)
Double `XOR` V1	11,993
Double `XOR` V2	27,060
Branchless Double `XOR`	14,529
High-Low Nibble	11,157
High Bit `XOR`	9,699

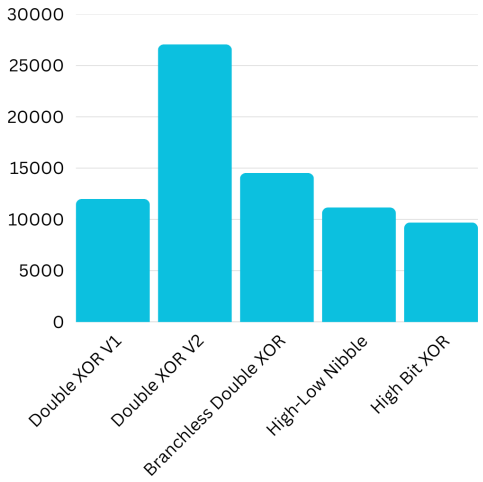

Figure 6.3: Performance of subtraction algorithms.

6.1.4 Saturating Subtraction

For saturating subtraction, we used the same algorithm variants as for saturating addition.

Algorithm	Median Time (µs)
Double `XOR`	44,636
Hybrid	33,070
Branchless Hybrid	14,672
High-Low Nibble	13,501
High Bit `XOR`	14,212

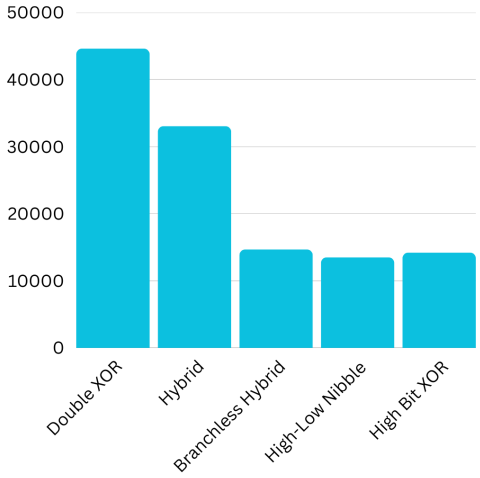

Figure 6.4: Performance of saturating subtraction algorithms.

6.1.5 Multiplication

For multiplication, we compared the three versions of the high-low nibble method and the high-bit `XOR` method.

Algorithm	Median Time (µs)
High-Low Nibble	28,495
Loopless High-Low Nibble	26,207
Optimized High-Low Nibble	19,765
High Bit `XOR`	18,369

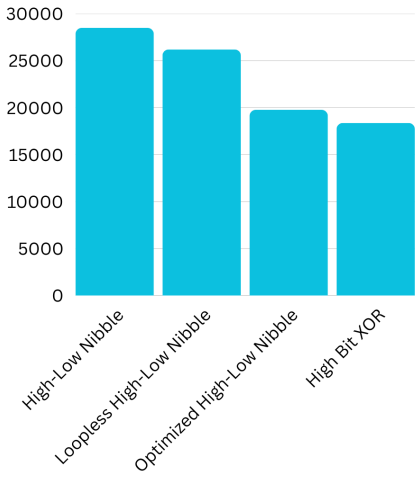

Figure 6.5: Performance of multiplication algorithms.

6.1.6 Saturating Multiplication

Finally, for saturating multiplication, we compared the two versions of the high-low nibble method.

Algorithm	Median Time (µs)
High-Low Nibble	31,590
Optimized High-Low Nibble	29,947

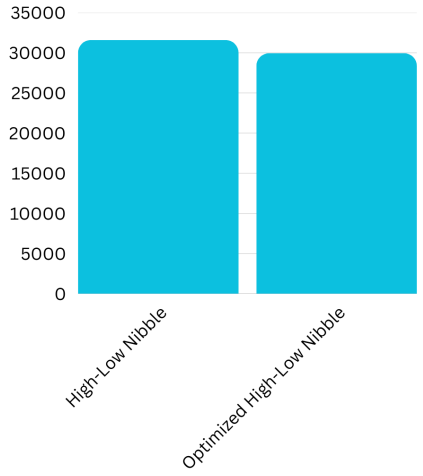

Figure 6.6: Performance of saturating multiplication algorithms.

6.2 Comparisons with the Lookup Table Approach

We also ran the same test to evaluate the performance of our lookup table functions. We compared implementations of both the 8-bit and 4-bit lookup table approaches. The results are independent of the operation we are performing since we just copy whatever precomputed values are stored in the table.

Algorithm	Median Time (µs)
Unoptimized 8-bit	34,495
Optimized 8-bit	27,228
Optimized 4-bit	53,478

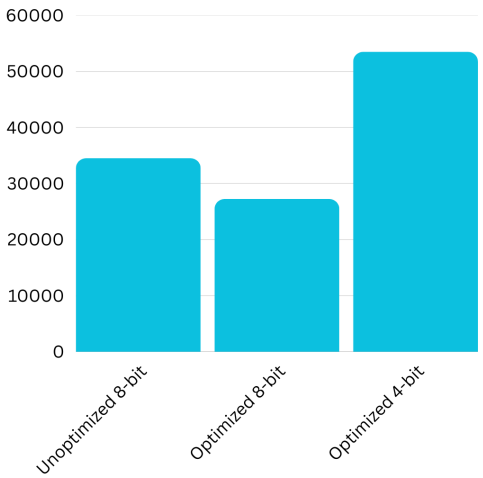

Figure 6.7: Performance of lookup table implementations.

From these results, we can see that on this system, the 8-bit approach is more performant, even though its larger table cannot fully fit in the L1 cache. We also notice that the lookup table approach is slower than our vectorized operation algorithms, except

for saturating multiplication.[1]

6.3 Comparisons with Non-Vectorized Operations

Now we compare our vectorized algorithms with a plain non-vectorized approach that performs the same operations. The non-vectorized version processes each byte independently.[2] Since each byte contains two nibbles, we isolate the high and low nibbles, compute the two results, and combine them together. This approach handles one byte at a time using a simple loop.

To ensure a fair comparison, we inlined the fastest versions of our vectorized algorithms—based on the previous benchmark results—directly into the benchmarking function. This avoids any function call overhead that could otherwise skew the results, since the non-vectorized approach does not involve any function calls. This full modified benchmarking function is also provided in the appendix.

Compiling with `-O2` and running the tests yields the following results. As before, we reported the median execution time across 1,001 runs for each operation, with each run processing two 32 MiB data blocks initialized with random values.

Operation	**Vectorized**	**Non-Vectorized**
Addition	6,854	31,963
Saturating Addition	10,395	51,970
Subtraction	7,753	31,589
Saturating Subtraction	11,360	62,011
Multiplication	14,547	44,137
Saturating Multiplication	25,562	72,299

Table 6.1: Median execution time for vectorized and non-vectorized benchmarks, measured in microseconds, compiling with the optimization flag `-O2`. For reference, the median execution time for the inlined lookup table approach is 24,803 microseconds.

If we compile with `-O3` and run the same tests again, though, something interesting happens.

As we can see from table 6.2, the non-vectorized algorithms received an incredible speedup—enough to outperform our explicitly vectorized versions in most cases. This

[1] Comparing the fastest version of each algorithm to the fastest implementation of the table approach.

[2] We cannot process each nibble separately because x86-64 processors do not have the instructions to work with 4-bit data types.

Operation	Vectorized	Non-Vectorized
Addition	5,981	5,865
Saturating Addition	6,653	8,130
Subtraction	5,989	5,729
Saturating Subtraction	6,624	6,172
Multiplication	8,707	7,633
Saturating Multiplication	14,364	9,745

Table 6.2: Median execution time for vectorized and non-vectorized benchmarks, measured in microseconds, compiling with the optimization flag `-O3`. For reference, the median execution time for the inlined lookup table approach is 24,806 microseconds.

happens because, with the `-O3` optimization flag, the compiler automatically vectorizes the originally scalar code, leveraging actual SIMD instructions and registers.

Just as an example, this was the original scalar algorithm for addition.

```
for(int i = 0; i < (1 << 25); i++)
{
    ((uint8_t*)data2)[i] =
    ((((uint8_t*)i1)[i] + ((uint8_t*)i2)[i]) & 0x0F) |
    ((((uint8_t*)i1)[i] + (((uint8_t*)i2)[i] & 0xF0)) & 0xF0);
}
```

Now, let's compile it with `-O3` and analyze the resulting Assembly code.

```
1     movdqa  .LC17(%rip), %xmm3
2     movdqa  .LC18(%rip), %xmm2
3     xorl    %eax, %eax
4     movq    8(%rsp), %rdx
5 .L255:
6     movdqu  0(%r13,%rax), %xmm0
7     movdqu  (%r12,%rax), %xmm7
8     movdqu  0(%r13,%rax), %xmm1
9     pand    %xmm3, %xmm0
10    paddb   %xmm7, %xmm0
11    paddb   %xmm7, %xmm1
12    pand    %xmm3, %xmm0
13    pand    %xmm2, %xmm1
14    por     %xmm1, %xmm0
15    movups  %xmm0, (%rdx,%rax)
16    addq    $16, %rax
17    cmpq    $33554432, %rax
18    jne     .L255
```

Here, lines 1 and 2 load 128-bit constants into the SIMD registers `%xmm3` and `%xmm2`. These are bitmasks that closely resemble the ones used in the high-low nibble method, except they are twice as long. Line 3 initializes `%rax` to zero,[3] which serves as the loop index variable. Line 4 sets `%rdx` to point to the output data block (`data2` in the C++ code). Before this snippet, `%r12` and `%r13` have been initialized to point to the input arrays (`i1` and `i2` in C++).

The actual loop begins on line 6. Lines 6 to 8 load 128 bits from `i1`[4] to `%xmm0` and `%xmm1`, and 128 bits from `i2`[5] to `%xmm7`. Lines 9 to 14 perform the actual additions between each nibble pair across the 128-bit registers. These lines effectively compute the thirty-two 4-bit additions, using a strategy similar to the high-low nibble method. The key detail here is that `paddb` performs bytewise addition and does not allow carry bits to propagate across bytes, so there is no need to mask out the high nibbles in advance in the input registers. After computing the partial results, lines 12 and 13 apply masks, and line 14 combines the two masked values using a bitwise `OR`.

Line 15 copies the result from `%xmm0` to memory. Line 16 increments the index `%rax` by 16 (since each iteration processes 16 bytes), and finally, lines 17 and 18 check whether we have finished processing all data, jumping back to line 6 if not.

The other *originally scalar* operations, although more complicated, are compiled in similar ways. Using SIMD instructions is what allows these implementations to be so fast.

[3]Even though it explicitly zeroes only the lower half of `%rax`, writing to `%eax` automatically zeros the upper 32 bits as well.

[4]Actually, `i1` plus `%rax`, where `%rax` is the index.

[5]Actually, `i2` plus `%rax`, where `%rax` is the index.

Chapter 7

Vectorized Composite Linear Algebra Operations

7.1 Dot Product

We can take our vectorized approach a step further by implementing more complex linear operations using the same principles we have used for basic arithmetic so far.

We will start with the dot product of two vectors, where each vector has 16 components and is stored in a 64-bit general-purpose register—one component per nibble, just like we have been doing up until now. Here, though, instead of producing a 16-component vector, the output is a single scalar value.

We defined our function to take two `uint4x16_t` inputs, but rather than returning another `uint4x16_t`, it returns a scalar `uint16_t`. We chose this type because it is the smallest standard unsigned integer that is guaranteed to be big enough to store the dot product without overflowing.

To implement this, we break the dot product into two steps: first, compute the 16 element-wise products, then sum them all together. Conveniently, our existing vectorized multiplication algorithm already computes element-wise products, which is exactly what we need for the first step. In particular, we turn our attention to our optimized saturating multiplication function. We opt for the saturating version because our optimized regular multiplication never computes the full products but just the lower 4 bits of them.[1] The saturating variant, on the other hand, computes the full products and then masks out the high nibbles before performing saturation. This means that we can reuse part of that

[1] And that was okay because we had to fit our products in a nibble by computing modulo-16 results. Here, though, we want to add the full sixteen products together and return the dot product on 16 bits.

function for the first step of our dot product algorithm, removing the final lines, where we computed and applied the overflow mask $\boldsymbol{f}$ and masked out the unwanted high bits in the products.

In fact, the following implementation is identical to the optimized saturating multiplication function up until line 21.[2]

```
1  uint16_t vdot_u16(uint4x16_t _a, uint4x16_t _b)
2  {
3      uint64_t a = _a.reg, b = _b.reg, p_low, p_high, o, f, a_low,
       a_high, m, dot;
4      a_low = a & 0x0F0F0F0F0F0F0F0F;
5      a_high = (a >> 4) & 0x0F0F0F0F0F0F0F0F;
6
7      m = (b & 0x1111111111111111) * 15;
8      p_low = a_low & m;
9      p_high = a_high & (m >> 4);
10
11     m = ((b >> 1) & 0x1111111111111111) * 15;
12     p_low += (a_low & m) << 1;
13     p_high += (a_high & (m >> 4)) << 1;
14
15     m = ((b >> 2) & 0x1111111111111111) * 15;
16     p_low += (a_low & m) << 2;
17     p_high += (a_high & (m >> 4)) << 2;
18
19     m = ((b >> 3) & 0x1111111111111111) * 15;
20     p_low += (a_low & m) << 3;
21     p_high += (a_high & (m >> 4)) << 3;
22
23     dot = (p_low & 0x00FF00FF00FF00FF) + (p_high & 0x00FF00FF00FF00FF)
       + ((p_low & 0xFF00FF00FF00FF00) >> 8) + ((p_high &
       0xFF00FF00FF00FF00) >> 8);
24     dot = dot + (dot >> 16);
25     dot = dot + (dot >> 32);
26
27     return (uint16_t)dot;
28 }
```

Listing 7.1: Implementation of `vdot_u16`.

After line 21, $\boldsymbol{p_{low}}$ and $\boldsymbol{p_{high}}$ hold the full 8-bit products of the original 4-bit vector components—$\boldsymbol{p_{low}}$ contains the results for the lower nibbles, and $\boldsymbol{p_{high}}$ for the upper

[2]Except here we also define the variable `dot`.

nibbles.[3] To compute the dot product, we need to add all these 16 products together.

Line 23 adds the even bytes of $\boldsymbol{p_{low}}$ and the even bytes of $\boldsymbol{p_{high}}$ together, using masks to avoid carry-ins between adjacent bytes. It does the same for the odd bytes, but first shifts them right to align the odd-byte sums with the even-byte sums.[4] Then, it adds these sums together, resulting in four values, each stored in a 16-bit portion of the variable `dot`. Masking is not necessary anymore because we know 16 bits are enough to hold the results.

Line 24 adds the upper two 16-bit sums into the lower two, collapsing from four 16-bit values to two. Finally, line 25 shifts the remaining upper 32 bits into the lower 32 and adds. The result in `dot`'s low 16 bits is thus the sum of all sixteen 8-bit products—our scalar dot product that we return.

7.2 Multiply-Accumulate by Scalar

Another useful operation is the multiply-accumulate by scalar, abbreviated `vmla_lane` in the NEON SIMD nomenclature we are following. Here, the term *lane* refers to selecting a specific component from a vector. This operation takes three vectors $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$, and a scalar index `lane`. It selects the component at position `lane` in $\boldsymbol{c}$, multiplies every component in $\boldsymbol{b}$ by this scalar, and adds every component in the result to the corresponding component in $\boldsymbol{a}$. Basically, we are computing $\boldsymbol{a} + \boldsymbol{b} \cdot \boldsymbol{c}$[lane].

This pattern is common in linear algebra kernels such as matrix multiplication. In our case, we implement it for vectors of sixteen 4-bit components packed into a 64-bit register.

The following function performs this operation using the same high-low nibble strategy we have used for other vector operations.

```
1 uint4x16_t vmla_lane_u4(uint4x16_t _a, uint4x16_t _b, uint4x16_t _c,
    int lane)
2 {
3     uint64_t a = _a.reg, b = _b.reg, c_lane = (_c.reg >> (lane << 2)) &
    0xFull;
4     uint64_t m_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F) * c_lane) & 0x0F0F0F0F0F0F0F0F;
5     uint64_t m_high = ((a & 0xF0F0F0F0F0F0F0F0) + (b &
    0xF0F0F0F0F0F0F0F0) * c_lane) & 0xF0F0F0F0F0F0F0F0;
6     return m_low | m_high;
7 }
```

Listing 7.2: Implementation of `vmla_lane_u4`.

[3] Also, the products in $\boldsymbol{p_{high}}$ are shifted to the right compared to the original nibbles.

[4] This also prevents losing the carry bit of the most significant byte.

The scalar value is extracted from $\boldsymbol{c}$ by shifting right by `lane` times four bits and isolating the least significant nibble using a bitmask. This gives us the nibble at the specified position, which we call $\boldsymbol{c_{lane}}$. This value is effectively the `lane`$^{\text{th}}$ component in $\boldsymbol{c}$.

The multiply-accumulate is then carried out separately for the low and high nibbles. To do that, we first isolate the low and high nibbles of $\boldsymbol{a}$ and $\boldsymbol{b}$ using bitmasks. Each masked portion of $\boldsymbol{b}$ is multiplied by the scalar $\boldsymbol{c_{lane}}$[5] and added to the corresponding portion of $\boldsymbol{a}$. This produces two results that we call $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$.[6] We are sure having a nibble of buffer between two values is enough because the maximum accumulated result in each component is $\mathbf{15 \times 15 + 15 = 240 \leq 255}$ and thus can fit in a byte.

Each component in $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$ is then masked again to confine it within the nibble bounds, discarding the *inactive* nibbles (the ones holding the high 4 bits of each result) and essentially computing modulo-16 results. We call these masked values $\boldsymbol{m_{low}}$ and $\boldsymbol{m_{high}}$. Finally, the two halves are recombined using a bitwise `OR`, producing the final result vector.

7.3 Saturating Multiply-Accumulate by Scalar

As usual, we can extend the multiply-accumulate operation to support saturation. Once again, we can do that by applying a similar trick to the one we used in the other saturating operations based on the high-low nibble method, which consists of looking at the *inactive* nibbles—if an inactive nibble is zero, the *active* nibble to its right did not overflow; otherwise, it did.

To implement this, we modify the code of the regular multiply-accumulate function to add saturating behavior using the strategy we just mentioned. Since we now need to check the inactive nibbles, we cannot just mask them out straight away after we compute $\boldsymbol{r_{low}}$ and $\boldsymbol{r_{high}}$ like we did before. Instead, we combine those inactive nibbles to build an overflow detector $\boldsymbol{o}$, and then we use $\boldsymbol{o}$ to saturate our result.

Essentially, we are following the exact same steps we used to saturate our multiplication function, and just like with saturating multiplication, the $\boldsymbol{r_{high}}$ we are computing here is shifted 4 bits to the right, to avoid losing the leftmost inactive nibble, which tells us whether the leftmost *active* nibble overflowed. We obviously need to shift $\boldsymbol{r_{high}}$ it back to the left when we use it to combine the high nibble results with the low nibble ones.

This leads to the following implementation.

[5] This is compiled as two actual `imulq` instructions—one for the high nibbles and one for the low nibbles—between the register holding $\boldsymbol{b_{high}}/\boldsymbol{b_{low}}$ and the one holding $\boldsymbol{c_{lane}}$.

[6] In the code above, these are only computed as temporary values and do not appear as variables.

```
1  uint4x16_t vqmla_lane_u4(uint4x16_t _a, uint4x16_t _b, uint4x16_t _c,
       int lane)
2  {
3      uint64_t a = _a.reg, b = _b.reg, c_lane = (_c.reg >> (lane << 2)) &
       0xFull, f;
4      uint64_t r_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
       0x0F0F0F0F0F0F0F0F) * c_lane);
5      uint64_t r_high = (((a >> 4) & 0x0F0F0F0F0F0F0F0F) + ((b >> 4) &
       0x0F0F0F0F0F0F0F0F) * c_lane);
6      uint64_t o = ((r_high) & 0xF0F0F0F0F0F0F0F0) | (((r_low) &
       0xF0F0F0F0F0F0F0F0) >> 4);
7      o |= (o >> 1);
8      o |= (o >> 2);
9      o &= 0x1111111111111111;
10     f = o * 15;
11     return ((r_high & 0x0F0F0F0F0F0F0F0F) << 4) | (r_low &
       0x0F0F0F0F0F0F0F0F) | f;
12 }
```

Listing 7.3: Implementation of `vqmla_lane_u4`.

In this code, lines 7 to 10 turn the value $\boldsymbol{o}$, which contains a non-zero value in each nibble where an overflow occurred (and zero otherwise), into the proper saturation mask $\boldsymbol{f}$: **0000** for no overflow and **1111** for overflow. We used these exact same lines in the saturating multiplication function.

7.4 Matrix Multiplication and Saturating Matrix Multiplication

As we know from linear algebra, matrix multiplication takes two input matrices, $\boldsymbol{m_0}$ (of size `rows` × `inner`) and $\boldsymbol{m_1}$ (of size `inner` × `columns`), and outputs another matrix, $\boldsymbol{m_r}$ (of size `rows` × `columns`), where each element in $\boldsymbol{m_r}$ is defined as:

$$\boldsymbol{m_r}[r][c] = \sum_{i=0}^{\texttt{inner}-1} \boldsymbol{m_0}[r][i] \cdot \boldsymbol{m_1}[i][c]$$

The summation can be thought of as a series of multiply-accumulate operations, as the following code snippet shows. Here, we are using a hypothetical function `mla(a, b, c)` that returns $\boldsymbol{a} + \boldsymbol{b} \cdot \boldsymbol{c}$, essentially a non-vectorized variant of the multiply-accumulate operation described earlier.

```
mr[r][c] = 0;
for(int i = 0; i < inner; i++)
{
    mr[r][c] = mla(mr[r][c], m0[r][i], m1[i][c]);
}
```

We used the `mla` function because our goal is to take advantage of the multiply-accumulate-by-scalar operation from earlier to build a vectorized matrix multiplication over 4-bit values. We assume our matrices are stored by rows in a single data block[7] and the elements are held in contiguous nibbles. For simplicity, we also assume all dimensions are divisible by 16. This is important because our vectorized `mla` functions operate on 8 bytes at a time, so we know each row is stored in a whole number of 8-byte sections.[8]

Vectorization enables the simultaneous computation of 16 adjacent elements within the same row of $\boldsymbol{m_r}$. Essentially, instead of computing a single element $\boldsymbol{m_r}[\boldsymbol{r}][\boldsymbol{c}]$ at a time, we leverage vectorized operations to process an entire set of elements $\boldsymbol{m_r}[\boldsymbol{r}][\boldsymbol{c_0} \ldots \boldsymbol{c_{15}}]$ as a single block. Mathematically, this can be expressed as:

$$\boldsymbol{m_r}[\boldsymbol{r}][\boldsymbol{c_0} \ldots \boldsymbol{c_{15}}] = \sum_{\boldsymbol{i=0}}^{\mathtt{inner}-\boldsymbol{1}} \boldsymbol{m_0}[\boldsymbol{r}][\boldsymbol{i}] \cdot \boldsymbol{m_1}[\boldsymbol{i}][\boldsymbol{c_0} \ldots \boldsymbol{c_{15}}]$$

where $\boldsymbol{c_0} \ldots \boldsymbol{c_{15}}$ represents the indices of the 16 adjacent elements being computed together, and the multiplication means that we are multiplying each element in $\boldsymbol{m_1}[\boldsymbol{i}][\boldsymbol{c_0} \ldots \boldsymbol{c_{15}}]$ by the same scalar value $\boldsymbol{m_0}[\boldsymbol{r}][\boldsymbol{i}]$.

Again the summation can be thought of as a series of multiply-accumulate operations, but this time, vectorized, leading to the following code snippet.

```
mr[r * parts + p] = 0;
for(int i = 0; i < inner; i++)
{
    mr[r * parts + p] = vmla_lane_u4(mr[r * parts + p], m1[i * parts +
   p], m0[r * parts_input + (i >> 4)], i & 15);
}
```

Here, `parts` is basically the number of 8-byte sections each row of $\boldsymbol{m_1}$ and $\boldsymbol{m_r}$ is divided into, and $\boldsymbol{p}$ is the index of the section we are computing (essentially representing $\boldsymbol{c_0} \ldots \boldsymbol{c_{15}}$). The variable `parts_input` is the number of sections in the rows of $\boldsymbol{m_0}$ (which is different, because the first operand has a different number of columns than the second operand and the result). Our matrices $\boldsymbol{m_r}$, $\boldsymbol{m_1}$ and $\boldsymbol{m_0}$ are `uint4x16_t` pointers. The

[7] Essentially linearized as a single vector.

[8] Actually, `rows` can be any positive integer and it does not need to be divisible by 16.

last two parameters we pass to the function specify the 16-element block where $\boldsymbol{m_0}[\boldsymbol{r}][\boldsymbol{i}]$ is (`m0[r * parts_input + (i >> 4)]`) and the index in the block (`i & 15`).

Now, to compute the whole result matrix, we can write the following function.

```
void vmm_u4(const uint4x16_t* m0, const uint4x16_t* m1, uint4x16_t* mr,
    int rows, int inner, int columns)
{
    int parts = columns >> 4;
    int parts_input = inner >> 4;
    for(int r = 0; r < rows; r++)
    {
        for(int p = 0; p < parts; p++)
        {
            mr[r * parts + p] = 0;
            for(int i = 0; i < inner; i++)
            {
                mr[r * parts + p] = vmla_lane_u4(mr[r * parts + p],
    m1[i * parts + p], m0[r * parts_input + (i >> 4)], i & 15);
            }
        }
    }
}
```

Listing 7.4: Implementation of `vmm_u4`.

To compute saturating matrix multiplication, we just substitute `vmla_lane_u4` with its saturating variant `vqmla_lane_u4`.

```
void vqmm_u4(const uint4x16_t* m0, const uint4x16_t* m1, uint4x16_t*
    mr, int rows, int inner, int columns)
{
    int parts = columns >> 4;
    int parts_input = inner >> 4;
    for(int r = 0; r < rows; r++)
    {
        for(int p = 0; p < parts; p++)
        {
            mr[r * parts + p] = 0;
            for(int i = 0; i < inner; i++)
            {
                mr[r * parts + p] = vqmla_lane_u4(mr[r * parts + p],
    m1[i * parts + p], m0[r * parts_input + (i >> 4)], i & 15);
            }
        }
    }
```

```
}
```

Listing 7.5: Implementation of vqmm_u4.

We can finally inline the function calls to try and gain a bit of performance, leading to the following implementations.

```
void vmm_u4(const uint4x16_t* m0, const uint4x16_t* m1, uint4x16_t* mr,
    int rows, int inner, int columns)
{
    int parts = columns >> 4;
    int parts_input = inner >> 4;
    for(int r = 0; r < rows; r++)
    {
        for(int p = 0; p < parts; p++)
        {
            mr[r * parts + p] = 0;
            for(int i = 0; i < inner; i++)
            {
                uint64_t a = mr[r * parts + p].reg, b = m1[i * parts +
    p].reg, c_lane = (m0[r * parts_input + (i >> 4)].reg >> ((i & 15) <<
    2)) & 0xFull;
                uint64_t m_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F) * c_lane) & 0x0F0F0F0F0F0F0F0F;
                uint64_t m_high = ((a & 0xF0F0F0F0F0F0F0F0) + (b &
    0xF0F0F0F0F0F0F0F0) * c_lane) & 0xF0F0F0F0F0F0F0F0;
                mr[r * parts + p] = m_low | m_high;
            }
        }
    }
}
```

Listing 7.6: Implementation of vmm_u4 after inlining the function call.

```
void vqmm_u4(const uint4x16_t* m0, const uint4x16_t* m1, uint4x16_t*
    mr, int rows, int inner, int columns)
{
    int parts = columns >> 4;
    int parts_input = inner >> 4;
    for(int r = 0; r < rows; r++)
    {
        for(int p = 0; p < parts; p++)
        {
            mr[r * parts + p] = 0;
            for(int i = 0; i < inner; i++)
```

```
11              {
12                  uint64_t a = mr[r * parts + p].reg, b = m1[i * parts +
    p].reg, c_lane = (m0[r * parts_input + (i >> 4)].reg >> ((i & 15) <<
    2)) & 0xFull, f;
13                  uint64_t r_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F) * c_lane);
14                  uint64_t r_high = (((a >> 4) & 0x0F0F0F0F0F0F0F0F) +
    ((b >> 4) & 0x0F0F0F0F0F0F0F0F) * c_lane);
15                  uint64_t o = ((r_high) & 0xF0F0F0F0F0F0F0F0) |
    (((r_low) & 0xF0F0F0F0F0F0F0F0) >> 4);
16                  o |= (o >> 1);
17                  o |= (o >> 2);
18                  o &= 0x1111111111111111;
19                  f = o * 15;
20                  mr[r * parts + p] = ((r_high & 0x0F0F0F0F0F0F0F0F) <<
    4) | (r_low & 0x0F0F0F0F0F0F0F0F) | f;
21              }
22          }
23      }
24  }
```

Listing 7.7: Implementation of `vqmm_u4` after inlining the function call.

Chapter 8

Performance of Linear Algebra Operations

8.1 Dot Product

As with basic operations, we will analyze the performance of vectorized linear algebra operations and see how they compare to a non-vectorized approach, starting with the dot product.

A lookup table approach is not feasible here because, unlike before, where each nibble result depended just on the two corresponding nibbles in the inputs, we have a single result that depends on 32 different 4-bit values (the components of our input vectors). This would require a table with $\mathbf{2^{64} \times 2^{64} = 2^{128}}$ entries, making it of astronomical size (exceeding 4.75×10^{29} GiB, considering that each entry requires at least 12 bits[1]).

To analyze the performance of our dot product algorithm, we used a very similar test function to the one we used before to benchmark our basic operations: we computed the dot products for 2^{22} pairs of 16-component vectors initialized with random values (where each component is stored on 4 bits), meaning our input data consists of two 32 MiB data blocks, just like when we tested basic operations. This time, though, the 2^{22} `uint16_t` results are stored in an 8 MiB array.

We tested both our vectorized dot product algorithm (inlined) and the following non-vectorized implementation (where `i1` and `i2` point to the input data blocks, `data2` points to the output vector).

```
for(int i = 0; i < (1 << 22); i++)
```

[1]The maximum possible dot product of two sixteen component vectors of 4-bit unsigned integers is $\mathbf{15 \times 15 \times 16 = 3{,}600}$.

```
{
    uint16_t dot = 0;
    for(int j = 0; j < 8; j++)
    {
        dot += (((uint8_t*)i1)[(i<<3) + j] & 0x0F)
        * (((uint8_t*)i2)[(i<<3) + j] & 0x0F);
        dot += (((uint8_t*)i1)[(i<<3) + j] >> 4)
        * (((uint8_t*)i2)[(i<<3) + j] >> 4);
    }
    ((uint16_t*)data2)[i] = (uint16_t)dot;
}
```

Below are the measured computation times (in microseconds). We took the median out of 1,001 runs.

Algorithm	`-O2` **Optimization**	`-O3` **Optimization**
Non-Vectorized	44,862	13,074
Vectorized	22,996	13,973

Table 8.1: Median execution time for vectorized and non-vectorized dot products measured in microseconds.

Again, compiling with the `-O3` optimization flag massively improved performance for the non-vectorized algorithm, as the compiler automatically vectorized it using SIMD instructions. The full benchmark function is provided in the appendix.

8.2 Multiply-Accumulate and Saturating Multiply-Accumulate

We conducted similar tests for our vectorized multiply-accumulate functions (both saturating and non-saturating). This benchmark involved three randomized 32 MiB input data blocks and another data block of the same size as output. The third input data block provided the scalar c_{lane} for the multiplication, and only one out of sixteen nibbles of it was actually used for the operation.[2] The full benchmark function is reported in the appendix.

Like before, we ran the tests on both our (inlined) vectorized algorithms and the non-vectorized alternatives, taking median execution times from 1,001 runs.

[2]The one at position `lane` of each 64-bit section. The value of `lane` was computed as a function of the index.

Algorithm	`-O2` Optimization	`-O3` Optimization
Non-Vectorized `mla_lane`	45,411	39,215
Vectorized `mla_lane`	10,993	10,055
Non-Vectorized `qmla_lane`	70,290	74,295
Vectorized `qmla_lane`	16,224	16,165

Table 8.2: Median execution time for vectorized and non-vectorized multiply-accumulate and saturating multiply-accumulate operations measured in microseconds.

A lookup table approach could be possible here, but the table would require significantly more space due to each result being dependent on three inputs rather than the two of our basic operations. This means we would not be able to couple nibbles together to compute two nibbles at a time (otherwise the table would not fit in the cache), and we would need to go with the 4-bit table approach. Since our previous tests showed that 4-bit lookup tables do not offer substantial performance benefits, and larger tables would be even slower,[3] we determined that this approach was not worthwhile.

8.3 Matrix Multiplication

To test matrix multiplication—and its saturating variant—we computed the product of two matrices of sizes 512 × 1,024 and 1,024 × 2,048, stored in 256 KiB and 1 MiB data blocks, respectively. These matrices were initialized with random values.[4] We ran the benchmark using both our vectorized algorithm and a non-vectorized version for comparison. As before, we measured execution times and took the median of 1,001 runs. The full benchmark function is provided in the appendix.

Algorithm	`-O2` Optimization	`-O3` Optimization
Non-Vectorized `mm`	1,682,446	1,655,617
Vectorized `mm`	251,607	249,316
Non-Vectorized `qmm`	2,890,124	2,885,002
Vectorized `qmm`	418,900	418,285

Table 8.3: Median execution time for vectorized and non-vectorized matrix multiplication and saturating matrix multiplication measured in microseconds.

[3] Also because lookup computation becomes more complex since we would have to calculate the correct entry based on three inputs rather than two.

[4] Instead of uniformly distributed random numbers, we used a sparse random function that only sets a few random bits. This prevents the result of the saturating multiplication from being all 1s.

Chapter 9

Conclusions

9.1 Key Takeaways

In this thesis, we have presented a comprehensive study on the vectorization of 4-bit arithmetic operations using 64-bit registers on the x86-64 architecture. We designed and analyzed multiple algorithms for addition, subtraction, multiplication, and their saturating counterparts—as well as more complex linear algebra operations such as matrix multiplication—and we have proved that software vectorization can offer measurable performance benefits even for data sizes not directly supported by SIMD instructions.

Performance benchmarks highlight the effectiveness of our different SWAR algorithms, which provide significant speedups relative to both non-vectorized algorithms and lookup table approaches. However, results also suggest that compiler-vectorized scalar implementations—as observed with the `-O3` optimization flag—can rival and, in some cases, outperform explicitly vectorized solutions, indicating further opportunities for exploration.

9.2 Future Works

Further work should reimplement the vectorized algorithms presented in this thesis using actual SIMD registers and instructions, mimicking what the compiler automatically did when we compiled with the optimization flag `-O3`. This would enable parallel computation of 32 nibbles instead of 16. Future research could also extend these techniques to 4-bit computations with signed two's complement arithmetic, fixed-point formats, or even floating-point representations. Additionally, these algorithms could be adapted for different hardware environments—including older or newer CPUs, systems with smaller caches, and architectures beyond x86-64, such as ARM or RISC-V. Exploring such variations may reveal new trade-offs and lead to alternative optimization strategies.

Appendix A

Helper and Compute Functions

A.1 Helper Functions

```
1 uint64_t my_random()
2 {
3     return (((((((uint64_t)(rand() & 0x7FFF) << 15) |
    (uint64_t)(rand()) & 0x7FFF) << 15) | (uint64_t)(rand()) & 0x7FFF)
    << 15) | (uint64_t)(rand()) & 0x7FFF) << 15) | (uint64_t)(rand() &
    0x7FFF);
4 }
```

Listing A.1: 64-bit random number generator.

```
1 uint64_t sparse_random()
2 {
3     uint64_t v = 0;
4     while(rand() % 4 != 0)
5         v += (1 << ((rand() % 16) * 4));
6     return v;
7 }
```

Listing A.2: 64-bit sparse random number generator.

```
1 void pin_to_core(int core_id)
2 {
3     cpu_set_t cpuset;
4     CPU_ZERO(&cpuset);
5     CPU_SET(core_id, &cpuset);
6     if(sched_setaffinity(0, sizeof(cpu_set_t), &cpuset) != 0)
7     {
8         perror("sched_setaffinity");
```

```
    }
}
```

Listing A.3: Function to pin our process to a specific core. We used this function for benchmarks.

A.2 Compute Functions to Check the Results

The following functions were used to check the correctness of our vectorized algorithms.

```
uint8_t vadd_u4_compute(uint8_t a, uint8_t b)
{
    return (a + b) % 16;
}

uint8_t vqadd_u4_compute(uint8_t a, uint8_t b)
{
    return (a + b) < 16 ? a + b : 15;
}

uint8_t vsub_u4_compute(uint8_t a, uint8_t b)
{
    return (a >= b) ? a - b : 16 + a - b;
}

uint8_t vqsub_u4_compute(uint8_t a, uint8_t b)
{
    return (a >= b) ? a - b : 0;
}

uint8_t vmul_u4_compute(uint8_t a, uint8_t b)
{
    return (a * b) % 16;
}

uint8_t vqmul_u4_compute(uint8_t a, uint8_t b)
{
    return (a * b) < 16 ? a * b : 15;
}
```

Listing A.4: Compute functions for basic operations.

```
uint16_t vdot_u16_compute(uint4x16_t a, uint4x16_t b)
```

```
{
    uint16_t r = 0;
    for(int i = 0; i < 16; i++)
    {
        r += a[i] * b[i];
    }
    return r;
}

uint8_t vmla_u4_compute(uint8_t a, uint8_t b, uint8_t c)
{
    return (a + b * c) % 16;
}

uint8_t vqmla_u4_compute(uint8_t a, uint8_t b, uint8_t c)
{
    return (a + b * c) < 16 ? (a + b * c) : 15;
}
```

Listing A.5: Compute functions for composite operations.

For matrix multiplication, we relied on the multiply-accumulate compute functions to check the results, and used the following function to access given elements.

```
uint8_t i_j(uint8_t* p, int i, int j, int columns)
{
    uint8_t v = p[i * (columns/2) + j/2];
    if(j % 2)
        return v >> 4;
    return v & 0xF;
}
```

Listing A.6: Function to access a given element in a matrix of 4-bit values stored by rows. The variable `p` points to the matrix, `columns` is the total number of columns in the matrix, `i` and `j` are the indices.

Appendix B

Benchmark Functions

B.1 Benchmark Functions for Basic Operations

Here is the code we used to benchmark the different algorithms for each operation, including the lookup table approach and the non-vectorized reference implementations. The type `OperationType` is an enumerator. Function pointers are used to select the intended function for performing the operation and checking the results.

```
uint64_t manipulate_data(OperationType operation)
{
    OpFunc op;
    ComputeFunc compute;
    const char* name;
    switch(operation)
    {
        case ADD:
            op = vadd_u4;
            compute = vadd_u4_compute;
            name = "vadd_u4";
            break;
        case QADD:
            op = vqadd_u4;
            compute = vqadd_u4_compute;
            name = "vqadd_u4";
            break;
        case SUB:
            op = vsub_u4;
            compute = vsub_u4_compute;
            name = "vsub_u4";
            break;
```

```
23          case QSUB:
24              op = vqsub_u4;
25              compute = vqsub_u4_compute;
26              name = "vqsub_u4";
27              break;
28          case MUL:
29              op = vmul_u4;
30              compute = vmul_u4_compute;
31              name = "vmul_u4";
32              break;
33          case QMUL:
34              op = vqmul_u4;
35              compute = vqmul_u4_compute;
36              name = "vqmul_u4";
37              break;
38      }
39      void* i1 = new uint64_t[1 << 22];
40      void* i2 = new uint64_t[1 << 22];
41      void* data0 = new uint64_t[1 << 22];
42      void* data1 = new uint64_t[1 << 22];
43      void* data2 = new uint64_t[1 << 22];
44      for(int i = 0; i < (1 << 22); i++)
45      {
46          ((uint64_t*)i1)[i] = my_random();
47          ((uint64_t*)i2)[i] = my_random();
48          ((uint64_t*)data0)[i] = 0;
49          ((uint64_t*)data1)[i] = 0;
50          ((uint64_t*)data2)[i] = 0;
51      }
52      build_table(compute);
53      cout << name << " \t";
54
55      auto start0 = high_resolution_clock::now();
56      for(int i = 0; i < (1 << 22); i++)
57      {
58          ((uint4x16_t*)data0)[i] = table_u4(((uint4x16_t*)i1)[i],
    ((uint4x16_t*)i2)[i]);
59      }
60      auto end0 = high_resolution_clock::now();
61      cout << duration_cast<microseconds>(end0 - start0).count() << " us
    (vectorized)\t ";
62
63      auto start1 = high_resolution_clock::now();
```

```
64      for(int i = 0; i < (1 << 22); i++)
65      {
66          ((uint4x16_t*)data1)[i] = table_u4(((uint4x16_t*)i1)[i],
    ((uint4x16_t*)i2)[i]);
67      }
68      auto end1 = high_resolution_clock::now();
69      cout << duration_cast<microseconds>(end1 - start1).count() << " us
    (table)\t";
70
71      auto start2 = high_resolution_clock::now();
72      switch(operation)
73      {
74          case ADD:
75              for(int i = 0; i < (1 << 25); i++)
76              {
77                  ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] +
    ((uint8_t*)i2)[i]) & 0x0F) | ((((uint8_t*)i1)[i] +
    (((uint8_t*)i2)[i] & 0xF0)) & 0xF0);
78              }
79              break;
80          case QADD:
81              for(int i = 0; i < (1 << 25); i++)
82              {
83                  ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] & 0x0F) +
    (((uint8_t*)i2)[i] & 0x0F) > 0xF ? 0x0F : (((uint8_t*)i1)[i] & 0x0F)
    + (((uint8_t*)i2)[i] & 0x0F)) | ((((uint8_t*)i1)[i] & 0xF0) +
    (((uint8_t*)i2)[i] & 0xF0) > 0xF0 ? 0xF0 : (((uint8_t*)i1)[i] &
    0xF0) + (((uint8_t*)i2)[i] & 0xF0));
84              }
85              break;
86          case SUB:
87              for(int i = 0; i < (1 << 25); i++)
88              {
89                  ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] -
    ((uint8_t*)i2)[i]) & 0x0F) | ((((uint8_t*)i1)[i] -
    (((uint8_t*)i2)[i] & 0xF0)) & 0xF0);
90              }
91              break;
92          case QSUB:
93              for(int i = 0; i < (1 << 25); i++)
94              {
95                  ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] & 0x0F) <
    (((uint8_t*)i2)[i] & 0x0F) ? 0x00 : (((uint8_t*)i1)[i] & 0x0F) -
```

```
    (((uint8_t*)i2)[i] & 0x0F)) | ((((uint8_t*)i1)[i] & 0xF0) <
    (((uint8_t*)i2)[i] & 0xF0) ? 0x00 : (((uint8_t*)i1)[i] & 0xF0) -
    (((uint8_t*)i2)[i] & 0xF0));
96              }
97              break;
98          case MUL:
99              for(int i = 0; i < (1 << 25); i++)
100             {
101                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] *
    ((uint8_t*)i2)[i]) & 0x0F) | (((((uint8_t*)i1)[i] & 0xF0) *
    (((uint8_t*)i2)[i] >> 4)) & 0xF0);
102             }
103             break;
104         case QMUL:
105             for(int i = 0; i < (1 << 25); i++)
106             {
107                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] & 0x0F) *
    (((uint8_t*)i2)[i] & 0x0F) > 0xF ? 0x0F : (((uint8_t*)i1)[i] & 0x0F)
    * (((uint8_t*)i2)[i] & 0x0F)) | ((((uint8_t*)i1)[i] & 0xF0) *
    ((((uint8_t*)i2)[i] & 0xF0) >> 4) > 0xF0 ? 0xF0 : (((uint8_t*)i1)[i]
    & 0xF0) * ((((uint8_t*)i2)[i] & 0xF0) >> 4));
108             }
109             break;
110     }
111     auto end2 = high_resolution_clock::now();
112     cout << duration_cast<microseconds>(end2 - start2).count() << " us
    (reference)\t" << endl;
113
114     for(int i = 0; i < (1 << 22); i++)
115     {
116         uint4x16_t control_a = *((uint4x16_t*)(i1) + i);
117         uint4x16_t control_b = *((uint4x16_t*)(i2) + i);
118         uint4x16_t d0 = *((uint4x16_t*)(data0) + i);
119         uint4x16_t d1 = *((uint4x16_t*)(data1) + i);
120         uint4x16_t d2 = *((uint4x16_t*)(data2) + i);
121         for(int i = 0; i < 16; i++)
122         {
123             if(d0[i] != compute(control_a[i], control_b[i]))
124             {
125                 cout << "ERROR D0" << endl;
126                 return -1;
127             }
128             if(d1[i] != compute(control_a[i], control_b[i]))
```

```
129             {
130                 cout << "ERROR D1" << endl;
131                 return -1;
132             }
133 
134             if(d2[i] != compute(control_a[i], control_b[i]))
135             {
136                 cout << "ERROR D2" << endl;
137                 return -1;
138             }
139         }
140     }
141     delete[] (uint64_t*)data0;
142     delete[] (uint64_t*)data1;
143     delete[] (uint64_t*)data2;
144     delete[] (uint64_t*)i1;
145     delete[] (uint64_t*)i2;
146     return (duration_cast<microseconds>(end0 - start0).count()) |
    (duration_cast<microseconds>(end1 - start1).count() << 20) |
    (duration_cast<microseconds>(end2 - start2).count() << 40);
147 }
```

Listing B.1: Benchmark function for basic operations.

The following procedure is very similar, but we inlined the code for the operations (and for the lookup table) instead of calling functions to avoid the overhead of function calls. For each operation, we used the fastest algorithm based on the data obtained from the function above.

```
1  uint64_t manipulate_data(OperationType operation)
2  {
3      OpFunc op;
4      ComputeFunc compute;
5      const char* name;
6      switch(operation)
7      {
8          case ADD:
9              op = vadd_u4;
10             compute = vadd_u4_compute;
11             name = "vadd_u4";
12             break;
13         case QADD:
14             op = vqadd_u4;
15             compute = vqadd_u4_compute;
```

```
            name = "vqadd_u4";
            break;
        case SUB:
            op = vsub_u4;
            compute = vsub_u4_compute;
            name = "vsub_u4";
            break;
        case QSUB:
            op = vqsub_u4;
            compute = vqsub_u4_compute;
            name = "vqsub_u4";
            break;
        case MUL:
            op = vmul_u4;
            compute = vmul_u4_compute;
            name = "vmul_u4";
            break;
        case QMUL:
            op = vqmul_u4;
            compute = vqmul_u4_compute;
            name = "vqmul_u4";
            break;
    }
    void* i1 = new uint64_t[1 << 22];
    void* i2 = new uint64_t[1 << 22];
    void* data0 = new uint64_t[1 << 22];
    void* data1 = new uint64_t[1 << 22];
    void* data2 = new uint64_t[1 << 22];
    for(int i = 0; i < (1 << 22); i++)
    {
        ((uint64_t*)i1)[i] = my_random();
        ((uint64_t*)i2)[i] = my_random();
        ((uint64_t*)data0)[i] = 0;
        ((uint64_t*)data1)[i] = 0;
        ((uint64_t*)data2)[i] = 0;
    }
    build_table(compute);
    cout << name << " \t";

    auto start0 = high_resolution_clock::now();
    switch(operation)
    {
        case ADD:
```

```
59            for(int i = 0; i < (1 << 22); i++)
60            {
61                uint64_t a = ((uint64_t*)i1)[i], b = ((uint64_t*)i2)[i];
62                uint64_t l = (a & 0x7777777777777777) + (b &
    0x7777777777777777);
63                uint64_t z = (a ^ b) & 0x8888888888888888;
64                ((uint64_t*)data0)[i] = l ^ z;
65            }
66            break;
67        case QADD:
68            for(int i = 0; i < (1 << 22); i++)
69            {
70                uint64_t a = ((uint64_t*)i1)[i], b = ((uint64_t*)i2)[i];
71                uint64_t r_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F));
72                uint64_t r_high = (((a >> 4) & 0x0F0F0F0F0F0F0F0F) +
    ((b >> 4) & 0x0F0F0F0F0F0F0F0F));
73                uint64_t f = ((r_high & 0xF0F0F0F0F0F0F0F0) | ((r_low &
    0xF0F0F0F0F0F0F0F0) >> 4)) * 15;
74                ((uint64_t*)data0)[i] = ((r_high & 0x0F0F0F0F0F0F0F0F)
    << 4)| (r_low & 0x0F0F0F0F0F0F0F0F) | f;
75            }
76            break;
77        case SUB:
78            for(int i = 0; i < (1 << 22); i++)
79            {
80                uint64_t a = ((uint64_t*)i1)[i], b = ((uint64_t*)i2)[i];
81                uint64_t l = (a | 0x8888888888888888) - (b &
    0x7777777777777777);
82                uint64_t z = ~(a ^ b) & 0x8888888888888888;
83                ((uint64_t*)data0)[i] = l ^ z;
84            }
85            break;
86        case QSUB:
87            for(int i = 0; i < (1 << 22); i++)
88            {
89                uint64_t a = ((uint64_t*)i1)[i], b = ((uint64_t*)i2)[i];
90                uint64_t r_low = ((a | 0xF0F0F0F0F0F0F0F0) - (b &
    0x0F0F0F0F0F0F0F0F));
91                uint64_t r_high = (((a >> 4) | 0xF0F0F0F0F0F0F0F0) -
    ((b >> 4) & 0x0F0F0F0F0F0F0F0F));
92                uint64_t o = ~((r_high | 0x0F0F0F0F0F0F0F0F) & ((r_low
    >> 4) | 0xF0F0F0F0F0F0F0F0));
```

```
93                  uint64_t f = o * 15;
94                  ((uint64_t*)data0)[i] = (((r_high & 0x0F0F0F0F0F0F0F0F)
    << 4) | (r_low & 0x0F0F0F0F0F0F0F0F)) & ~f;
95              }
96              break;
97          case MUL:
98              for(int i = 0; i < (1 << 22); i++)
99              {
100                 uint64_t a = ((uint64_t*)i1)[i], b =
    ((uint64_t*)i2)[i], p, k, l, z;
101
102                 p = a & ((b & 0x1111111111111111) * 15);
103
104                 a <<= 1;
105                 k = a & ((b & 0x2222222222222222) * 7);
106                 l = (k & 0x7777777777777777) + (p & 0x7777777777777777);
107                 z = (k ^ p) & 0x8888888888888888;
108                 p = l ^ z;
109
110                 a <<= 1;
111                 k = a & ((b & 0x4444444444444444) * 3);
112                 l = (k & 0x7777777777777777) + (p & 0x7777777777777777);
113                 z = (k ^ p) & 0x8888888888888888;
114                 p = l ^ z;
115
116                 a <<= 1;
117                 k = a & (b & 0x8888888888888888);
118                 p = p ^ k;
119
120                 ((uint64_t*)data0)[i] = p;
121             }
122             break;
123         case QMUL:
124             for(int i = 0; i < (1 << 22); i++)
125             {
126                 uint64_t a = ((uint64_t*)i1)[i], b =
    ((uint64_t*)i2)[i], p_low, p_high, o, f, a_low, a_high, m;
127                 a_low = a & 0x0F0F0F0F0F0F0F0F;
128                 a_high = (a >> 4) & 0x0F0F0F0F0F0F0F0F;
129
130                 m = (b & 0x1111111111111111) * 15;
131                 p_low = a_low & m;
132                 p_high = a_high & (m >> 4);
```

```
133
134                 m = ((b >> 1) & 0x1111111111111111) * 15;
135                 p_low += (a_low & m) << 1;
136                 p_high += (a_high & (m >> 4)) << 1;
137
138                 m = ((b >> 2) & 0x1111111111111111) * 15;
139                 p_low += (a_low & m) << 2;
140                 p_high += (a_high & (m >> 4)) << 2;
141
142                 m = ((b >> 3) & 0x1111111111111111) * 15;
143                 p_low += (a_low & m) << 3;
144                 p_high += (a_high & (m >> 4)) << 3;
145
146                 o = (p_high & 0xF0F0F0F0F0F0F0F0) | ((p_low &
    0xF0F0F0F0F0F0F0F0) >> 4);
147                 o |= (o >> 1);
148                 o |= (o >> 2);
149                 o &= 0x1111111111111111;
150                 f = o * 15;
151
152                 ((uint64_t*)data0)[i] = ((p_low & 0x0F0F0F0F0F0F0F0F) |
    ((p_high & 0x0F0F0F0F0F0F0F0F) << 4)) | f;
153             }
154             break;
155     }
156     auto end0 = high_resolution_clock::now();
157     cout << duration_cast<microseconds>(end0 - start0).count() << " us
    (vectorized)\t ";
158
159     auto start1 = high_resolution_clock::now();
160     for(int i = 0; i < (1 << 22); i++)
161     {
162         ((uint64_t*)data1)[i]  = (uint64_t)table[((((uint64_t*)i1)[i] &
    0xFFull) << 8) | (((uint64_t*)i2)[i] & 0xFFull)];
163         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
    & 0xFF00ull) << 8) | (((uint64_t*)i2)[i] & 0xFF00ull)) >> 8] << 8;
164         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
    & 0xFF0000ull) << 8) | (((uint64_t*)i2)[i] & 0xFF0000ull)) >> 16] <<
    16;
165         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
    & 0xFF000000ull) << 8) | (((uint64_t*)i2)[i] & 0xFF000000ull)) >>
    24] << 24;
166         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
```

```
& 0xFF00000000ull) << 8) | (((uint64_t*)i2)[i] & 0xFF00000000ull))
>> 32] << 32;
167         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
& 0xFF0000000000ull) << 8) | (((uint64_t*)i2)[i] &
0xFF0000000000ull)) >> 40] << 40;
168         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
& 0xFF000000000000ull) << 8) | (((uint64_t*)i2)[i] &
0xFF000000000000ull)) >> 48] << 48;
169         ((uint64_t*)data1)[i] |= (uint64_t)table[(((((uint64_t*)i1)[i]
& 0xFF00000000000000ull)) | ((((uint64_t*)i2)[i] &
0xFF00000000000000ull) >> 8)) >> 48] << 56;
170     }
171     auto end1 = high_resolution_clock::now();
172     cout << duration_cast<microseconds>(end1 - start1).count() << " us
(table)\t";
173
174     auto start2 = high_resolution_clock::now();
175     switch(operation)
176     {
177         case ADD:
178             for(int i = 0; i < (1 << 25); i++)
179             {
180                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] +
((uint8_t*)i2)[i]) & 0x0F) | ((((uint8_t*)i1)[i] +
(((uint8_t*)i2)[i] & 0xF0)) & 0xF0);
181             }
182             break;
183         case QADD:
184             for(int i = 0; i < (1 << 25); i++)
185             {
186                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] & 0x0F) +
(((uint8_t*)i2)[i] & 0x0F) > 0xF ? 0x0F : (((uint8_t*)i1)[i] & 0x0F)
+ (((uint8_t*)i2)[i] & 0x0F)) | ((((uint8_t*)i1)[i] & 0xF0) +
(((uint8_t*)i2)[i] & 0xF0) > 0xF0 ? 0xF0 : (((uint8_t*)i1)[i] &
0xF0) + (((uint8_t*)i2)[i] & 0xF0));
187             }
188             break;
189         case SUB:
190             for(int i = 0; i < (1 << 25); i++)
191             {
192                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] -
((uint8_t*)i2)[i]) & 0x0F) | ((((uint8_t*)i1)[i] -
(((uint8_t*)i2)[i] & 0xF0)) & 0xF0);
```

```
193             }
194             break;
195         case QSUB:
196             for(int i = 0; i < (1 << 25); i++)
197             {
198                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] & 0x0F) <
    (((uint8_t*)i2)[i] & 0x0F) ? 0x00 : (((uint8_t*)i1)[i] & 0x0F) -
    (((uint8_t*)i2)[i] & 0x0F)) | ((((uint8_t*)i1)[i] & 0xF0) <
    (((uint8_t*)i2)[i] & 0xF0) ? 0x00 : (((uint8_t*)i1)[i] & 0xF0) -
    (((uint8_t*)i2)[i] & 0xF0));
199             }
200             break;
201         case MUL:
202             for(int i = 0; i < (1 << 25); i++)
203             {
204                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] *
    ((uint8_t*)i2)[i]) & 0x0F) | (((((uint8_t*)i1)[i] & 0xF0) *
    (((uint8_t*)i2)[i] >> 4)) & 0xF0);
205             }
206             break;
207         case QMUL:
208             for(int i = 0; i < (1 << 25); i++)
209             {
210                 ((uint8_t*)data2)[i] = ((((uint8_t*)i1)[i] & 0x0F) *
    (((uint8_t*)i2)[i] & 0x0F) > 0xF ? 0x0F : (((uint8_t*)i1)[i] & 0x0F)
    * (((uint8_t*)i2)[i] & 0x0F)) | ((((uint8_t*)i1)[i] & 0xF0) *
    ((((uint8_t*)i2)[i] & 0xF0) >> 4) > 0xF0 ? 0xF0 : (((uint8_t*)i1)[i]
    & 0xF0) * ((((uint8_t*)i2)[i] & 0xF0) >> 4));
211             }
212             break;
213     }
214     auto end2 = high_resolution_clock::now();
215     cout << duration_cast<microseconds>(end2 - start2).count() << " us
    (reference)\t" << endl;
216 
217     for(int i = 0; i < (1 << 22); i++)
218     {
219         uint4x16_t control_a = *((uint4x16_t*)(i1) + i);
220         uint4x16_t control_b = *((uint4x16_t*)(i2) + i);
221         uint4x16_t d0 = *((uint4x16_t*)(data0) + i);
222         uint4x16_t d1 = *((uint4x16_t*)(data1) + i);
223         uint4x16_t d2 = *((uint4x16_t*)(data2) + i);
224         for(int i = 0; i < 16; i++)
```

```
225         {
226             if(d0[i] != compute(control_a[i], control_b[i]))
227             {
228                 cout << "ERROR D0" << endl;
229                 return -1;
230             }
231 
232             if(d1[i] != compute(control_a[i], control_b[i]))
233             {
234                 cout << "ERROR D1" << endl;
235                 return -1;
236             }
237 
238             if(d2[i] != compute(control_a[i], control_b[i]))
239             {
240                 cout << "ERROR D2" << endl;
241                 return -1;
242             }
243         }
244     }
245     delete[] (uint64_t*)data0;
246     delete[] (uint64_t*)data1;
247     delete[] (uint64_t*)data2;
248     delete[] (uint64_t*)i1;
249     delete[] (uint64_t*)i2;
250     return (duration_cast<microseconds>(end0 - start0).count()) |
    (duration_cast<microseconds>(end1 - start1).count() << 20) |
    (duration_cast<microseconds>(end2 - start2).count() << 40);
251 }
```

Listing B.2: Benchmark function for basic operations, inlining the functions.

B.2 Benchmark Functions for Composite Operations

This is the benchmark function we used to test the dot product operation.

```
1 uint64_t manipulate_data(OperationType operation)
2 {
3     OpFunc op;
4     ComputeFunc compute;
5     const char* name;
6     switch(operation)
7     {
```

```
8         case DOT:
9             op = vdot_u16;
10            compute = vdot_u16_compute;
11            name = "vdot_u16";
12            break;
13    }
14    void* i1 = new uint64_t[1 << 22];
15    void* i2 = new uint64_t[1 << 22];
16    void* data0 = new uint16_t[1 << 22];
17    void* data2 = new uint16_t[1 << 22];
18    for(int i = 0; i < (1 << 22); i++)
19    {
20        ((uint64_t*)i1)[i] = my_random();
21        ((uint64_t*)i2)[i] = my_random();
22        ((uint16_t*)data0)[i] = 0;
23        ((uint16_t*)data2)[i] = 0;
24    }
25    cout << name << " \t";
26
27    auto start0 = high_resolution_clock::now();
28    switch(operation)
29    {
30        case DOT:
31            for(int i = 0; i < (1 << 22); i++)
32            {
33                uint64_t a = ((uint64_t*)i1)[i], b =
    ((uint64_t*)i2)[i], p_low, p_high, o, f, a_low, a_high, m, dot;
34                a_low = a & 0x0F0F0F0F0F0F0F0F;
35                a_high = (a >> 4) & 0x0F0F0F0F0F0F0F0F;
36
37                m = (b & 0x1111111111111111) * 15;
38                p_low = a_low & m;
39                p_high = a_high & (m >> 4);
40
41                m = ((b >> 1) & 0x1111111111111111) * 15;
42                p_low += (a_low & m) << 1;
43                p_high += (a_high & (m >> 4)) << 1;
44
45                m = ((b >> 2) & 0x1111111111111111) * 15;
46                p_low += (a_low & m) << 2;
47                p_high += (a_high & (m >> 4)) << 2;
48
49                m = ((b >> 3) & 0x1111111111111111) * 15;
```

```
50                p_low += (a_low & m) << 3;
51                p_high += (a_high & (m >> 4)) << 3;
52
53                dot = (p_low & 0x00FF00FF00FF00FF) + (p_high &
    0x00FF00FF00FF00FF) + ((p_low & 0xFF00FF00FF00FF00) >> 8) + ((p_high
    & 0xFF00FF00FF00FF00) >> 8);
54                dot = dot + (dot >> 16);
55                dot = dot + (dot >> 32);
56
57                ((uint16_t*)data0)[i] = (uint16_t)dot;
58            }
59            break;
60     }
61     auto end0 = high_resolution_clock::now();
62     cout << duration_cast<microseconds>(end0 - start0).count() << " us
    (vectorized)\t ";
63
64     auto start2 = high_resolution_clock::now();
65     switch(operation)
66     {
67         case DOT:
68             for(int i = 0; i < (1 << 22); i++)
69             {
70                 uint16_t dot = 0;
71                 for(int j = 0; j < 8; j++)
72                 {
73                     dot += (((uint8_t*)i1)[(i<<3) + j] & 0x0F) *
    (((uint8_t*)i2)[(i<<3) + j] & 0x0F);
74                     dot += (((uint8_t*)i1)[(i<<3) + j] >> 4) *
    (((uint8_t*)i2)[(i<<3) + j] >> 4);
75                 }
76                 ((uint16_t*)data2)[i] = (uint16_t)dot;
77             }
78             break;
79     }
80     auto end2 = high_resolution_clock::now();
81     cout << duration_cast<microseconds>(end2 - start2).count() << " us
    (reference)\t" << endl;
82
83     for(int i = 0; i < (1 << 22); i++)
84     {
85         uint4x16_t control_a = *((uint4x16_t*)(i1) + i);
86         uint4x16_t control_b = *((uint4x16_t*)(i2) + i);
```

```
87          uint16_t d0 = *((uint16_t*)(data0) + i);
88          uint16_t d2 = *((uint16_t*)(data2) + i);
89
90          if(d0 != compute(control_a, control_b))
91          {
92              cout << "ERROR D0" << endl;
93              return -1;
94          }
95
96          if(d2 != compute(control_a, control_b))
97          {
98              cout << "ERROR D2" << endl;
99              return -1;
100         }
101     }
102     delete[] (uint16_t*)data0;
103     delete[] (uint16_t*)data2;
104     delete[] (uint64_t*)i1;
105     delete[] (uint64_t*)i2;
106     return (duration_cast<microseconds>(end0 - start0).count()) |
    (duration_cast<microseconds>(end2 - start2).count() << 32);
107 }
```

Listing B.3: Benchmark function for vdot_u4.

Here is the benchmark function for the multiply-accumulate and saturating multiply-accumulate.

```
1  uint64_t manipulate_data(OperationType operation)
2  {
3      OpFunc op;
4      ComputeFunc compute;
5      const char* name;
6      switch(operation)
7      {
8          case MLALANE:
9              op = vmla_lane_u4;
10             compute = vmla_u4_compute;
11             name = "vmla_lane_u4";
12             break;
13         case QMLALANE:
14             op = vqmla_lane_u4;
15             compute = vqmla_u4_compute;
16             name = "vqmla_lane_u4";
17             break;
```

```
18      }
19      void* i1 = new uint64_t[1 << 22];
20      void* i2 = new uint64_t[1 << 22];
21      void* i3 = new uint64_t[1 << 22];
22      void* data0 = new uint64_t[1 << 22];
23      void* data2 = new uint64_t[1 << 22];
24      for(int i = 0; i < (1 << 22); i++)
25      {
26          ((uint64_t*)i1)[i] = my_random();
27          ((uint64_t*)i2)[i] = my_random();
28          ((uint64_t*)i3)[i] = my_random();
29          ((uint64_t*)data0)[i] = 0;
30          ((uint64_t*)data2)[i] = 0;
31      }
32      cout << name << " \t";
33
34      auto start0 = high_resolution_clock::now();
35      switch(operation)
36      {
37          case MLALANE:
38              for(int i = 0; i < (1 << 22); i++)
39              {
40                  int lane = i & 0xF;
41                  uint64_t a = ((uint64_t*)i1)[i], b =
    ((uint64_t*)i2)[i], c_lane = (((uint64_t*)i3)[i] >> (lane << 2)) &
    0xFull;
42                  uint64_t m_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F) * c_lane) & 0x0F0F0F0F0F0F0F0F;
43                  uint64_t m_high = ((a & 0xF0F0F0F0F0F0F0F0) + (b &
    0xF0F0F0F0F0F0F0F0) * c_lane) & 0xF0F0F0F0F0F0F0F0;
44                  ((uint64_t*)data0)[i] = m_low | m_high;
45              }
46              break;
47          case QMLALANE:
48              for(int i = 0; i < (1 << 22); i++)
49              {
50                  int lane = i & 0xF;
51                  uint64_t a = ((uint64_t*)i1)[i], b =
    ((uint64_t*)i2)[i], c_lane = (((uint64_t*)i3)[i] >> (lane << 2)) &
    0xFull, f;
52                  uint64_t r_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b &
    0x0F0F0F0F0F0F0F0F) * c_lane);
53                  uint64_t r_high = (((a >> 4) & 0x0F0F0F0F0F0F0F0F) +
```

```
    ((b >> 4) & 0x0F0F0F0F0F0F0F0F) * c_lane);
54                  uint64_t o = ((r_high) & 0xF0F0F0F0F0F0F0F0) |
    (((r_low) & 0xF0F0F0F0F0F0F0F0) >> 4);
55                  o |= (o >> 1);
56                  o |= (o >> 2);
57                  o &= 0x1111111111111111;
58                  f = o * 15;
59                  ((uint64_t*)data0)[i] = ((r_high & 0x0F0F0F0F0F0F0F0F)
    << 4) | (r_low & 0x0F0F0F0F0F0F0F0F) | f;
60              }
61              break;
62      }
63      auto end0 = high_resolution_clock::now();
64      cout << duration_cast<microseconds>(end0 - start0).count() << " us
    (vectorized)\t ";
65
66      auto start2 = high_resolution_clock::now();
67      switch(operation)
68      {
69          case MLALANE:
70              for(int i = 0; i < (1 << 22); i++)
71              {
72                  uint64_t c_lane = ((uint8_t*)i3)[(i << 3) + ((i & 0xF)
    >> 1)];
73                  c_lane = (c_lane >> ((i & 1) << 2)) & 0x0F;
74                  for(int j = 0; j < 8; j++)
75                  {
76                      ((uint8_t*)data2)[(i<<3) + j] =
    ((((uint8_t*)i1)[(i<<3) + j] + ((uint8_t*)i2)[(i<<3) + j] * c_lane)
    & 0x0F) | ((((uint8_t*)i1)[(i<<3) + j] + ((((uint8_t*)i2)[(i<<3) +
    j] & 0xF0) * c_lane)) & 0xF0);
77                  }
78              }
79              break;
80          case QMLALANE:
81              for(int i = 0; i < (1 << 22); i++)
82              {
83                  uint64_t c_lane = ((uint8_t*)i3)[(i << 3) + ((i & 0xF)
    >> 1)];
84                  c_lane = (c_lane >> ((i & 1) << 2)) & 0x0F;
85                  for(int j = 0; j < 8; j++)
86                  {
87                      ((uint8_t*)data2)[(i<<3) + j] =
```

```
    ((((uint8_t*)i1)[(i<<3) + j] & 0x0F) + (((uint8_t*)i2)[(i<<3) + j] &
    0x0F) * c_lane > 0x0F ? 0x0F : (((uint8_t*)i1)[(i<<3) + j] & 0x0F) +
    (((uint8_t*)i2)[(i<<3) + j] & 0x0F) * c_lane) |
    ((((uint8_t*)i1)[(i<<3) + j] & 0xF0) + ((((uint8_t*)i2)[(i<<3) + j]
    & 0xF0) * c_lane) > 0xF0 ? 0xF0 : (((uint8_t*)i1)[(i<<3) + j] &
    0xF0) + ((((uint8_t*)i2)[(i<<3) + j] & 0xF0) * c_lane));
88                  }
89              }
90      }
91      auto end2 = high_resolution_clock::now();
92      cout << duration_cast<microseconds>(end2 - start2).count() << " us
    (reference)\t" << endl;
93
94      for(int i = 0; i < (1 << 22); i++)
95      {
96          uint4x16_t control_a = *((uint4x16_t*)(i1) + i);
97          uint4x16_t control_b = *((uint4x16_t*)(i2) + i);
98          uint4x16_t control_c = *((uint4x16_t*)(i3) + i);
99          uint4x16_t d0 = *((uint4x16_t*)(data0) + i);
100         uint4x16_t d2 = *((uint4x16_t*)(data2) + i);
101         for(int j = 0; j < 16; j++)
102         {
103             if(d0[j] != compute(control_a[j], control_b[j], control_c[i
    & 0xF]))
104             {
105                 cout << "ERROR D0" << endl;
106                 return -1;
107             }
108
109             if(d2[j] != compute(control_a[j], control_b[j], control_c[i
    & 0xF]))
110             {
111                 cout << "ERROR D2" << endl;
112                 return -1;
113             }
114         }
115     }
116     delete[] (uint64_t*)data0;
117     delete[] (uint64_t*)data2;
118     delete[] (uint64_t*)i1;
119     delete[] (uint64_t*)i2;
120     delete[] (uint64_t*)i3;
121     return (duration_cast<microseconds>(end0 - start0).count()) |
```

```
    (duration_cast<microseconds>(end2 - start2).count() << 32);
}
```

Listing B.4: Benchmark function for `vmla_lane_u4` and `qvmla_lane_u4`.

Finally, this is the function we used to test matrix multiplication and saturating matrix multiplication.

```
uint64_t manipulate_data(OperationType operation)
{
    OpFunc op;
    ComputeFunc compute;
    const char* name;
    int rows = 512, columns = 2048, inner = 1024;
    switch(operation)
    {
        case MM:
            op = vmm_u4;
            compute = vmla_u4_compute;
            name = "vmm_u4 ";
            break;
        case QMM:
            op = vqmm_u4;
            compute = vqmla_u4_compute;
            name = "vqmm_u4 ";
            break;
    }
    void* i1 = new uint64_t[1 << 15];
    void* i2 = new uint64_t[1 << 17];
    void* data0 = new uint64_t[1 << 16];
    void* data2 = new uint64_t[1 << 16];
    for(int i = 0; i < (1 << 15); i++)
    {
        ((uint64_t*)i1)[i] = sparse_random();
    }
    for(int i = 0; i < (1 << 17); i++)
    {
        ((uint64_t*)i2)[i] = sparse_random();
    }
    for(int i = 0; i < (1 << 16); i++)
    {
        ((uint64_t*)data0)[i] = 0;
        ((uint64_t*)data2)[i] = 0;
    }
    cout << name << " \t";
```

```
38
39      auto start0 = high_resolution_clock::now();
40      const uint4x16_t* m0 = (uint4x16_t*)i1;
41      const uint4x16_t* m1 = (uint4x16_t*)i2;
42      uint4x16_t* mr = (uint4x16_t*)data0;
43      int parts = columns >> 4;
44      int parts_input = inner >> 4;
45      switch(operation)
46      {
47          case MM:
48              for(int r = 0; r < rows; r++)
49              {
50                  for(int p = 0; p < parts; p++)
51                  {
52                      mr[r * parts + p] = 0;
53                      for(int i = 0; i < inner; i++)
54                      {
55                          uint64_t a = mr[r * parts + p].reg, b = m1[i *
    parts + p].reg, c_lane = (m0[r * parts_input + (i >> 4)].reg >> ((i
    & 15) << 2)) & 0xFull;
56                          uint64_t m_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b
    & 0x0F0F0F0F0F0F0F0F) * c_lane) & 0x0F0F0F0F0F0F0F0F;
57                          uint64_t m_high = ((a & 0xF0F0F0F0F0F0F0F0) +
    (b & 0xF0F0F0F0F0F0F0F0) * c_lane) & 0xF0F0F0F0F0F0F0F0;
58                          mr[r * parts + p] = m_low | m_high;
59                      }
60                  }
61              }
62              break;
63          case QMM:
64              for(int r = 0; r < rows; r++)
65              {
66                  for(int p = 0; p < parts; p++)
67                  {
68                      mr[r * parts + p] = 0;
69                      for(int i = 0; i < inner; i++)
70                      {
71                          uint64_t a = mr[r * parts + p].reg, b = m1[i *
    parts + p].reg, c_lane = (m0[r * parts_input + (i >> 4)].reg >> ((i
    & 15) << 2)) & 0xFull, f;
72                          uint64_t r_low = ((a & 0x0F0F0F0F0F0F0F0F) + (b
    & 0x0F0F0F0F0F0F0F0F) * c_lane);
73                          uint64_t r_high = (((a >> 4) &
```

```
    0x0F0F0F0F0F0F0F0F) + ((b >> 4) & 0x0F0F0F0F0F0F0F0F) * c_lane);
74                      uint64_t o = ((r_high) & 0xF0F0F0F0F0F0F0F0) |
    (((r_low) & 0xF0F0F0F0F0F0F0F0) >> 4);
75                      o |= (o >> 1);
76                      o |= (o >> 2);
77                      o &= 0x1111111111111111;
78                      f = o * 15;
79                      mr[r * parts + p] = ((r_high &
    0x0F0F0F0F0F0F0F0F) << 4) | (r_low & 0x0F0F0F0F0F0F0F0F) | f;
80                  }
81              }
82          }
83          break;
84  }
85  auto end0 = high_resolution_clock::now();
86  cout << duration_cast<microseconds>(end0 - start0).count() << " us
    (vectorized)\t ";
87
88  auto start2 = high_resolution_clock::now();
89  switch(operation)
90  {
91      case MM:
92          for(int r = 0; r < rows; r++)
93          {
94              for(int p = 0; p < (columns >> 1); p++)
95              {
96                  ((uint8_t*)data2)[r * (columns >> 1) + p] = 0;
97                  for(int i = 0; i < inner; i++)
98                  {
99                      uint8_t v = ((uint8_t*)i1)[r * (inner >> 1) +
    (i >> 1)];
100                     v = (v >> ((i & 1) << 2));
101                     uint8_t w = ((uint8_t*)i2)[i * (columns >> 1) +
    p];
102                     w = ((w * v) & 0x0F) | (((w & 0xF0) * v) &
    0xF0);
103                     ((uint8_t*)data2)[r * (columns >> 1) + p] =
    ((((uint8_t*)data2)[r * (columns >> 1) + p] + w) & 0x0F) |
    ((((uint8_t*)data2)[r * (columns >> 1) + p] + (w & 0xF0)) & 0xF0);
104                 }
105             }
106         }
107         break;
```

```
108         case QMM:
109             for(int r = 0; r < rows; r++)
110             {
111                 for(int p = 0; p < (columns >> 1); p++)
112                 {
113                     ((uint8_t*)data2)[r * (columns >> 1) + p] = 0;
114                     for(int i = 0; i < inner; i++)
115                     {
116                         uint8_t v = ((uint8_t*)i1)[r * (inner >> 1) +
    (i >> 1)];
117                         v = (v >> ((i & 1) << 2)) & 0xF;
118                         uint8_t w = ((uint8_t*)i2)[i * (columns >> 1) +
    p];
119                         w = (((w & 0xF) * v) > 0xF ? 0xF : ((w & 0xF) *
    v)) | (((w & 0xF0) * v) > 0xF0 ? 0xF0 : ((w & 0xF0) * v));
120                         ((uint8_t*)data2)[r * (columns >> 1) + p] =
    ((((uint8_t*)data2)[r * (columns >> 1) + p] & 0x0F) + (w & 0x0F) >
    0xF ? 0x0F : (((uint8_t*)data2)[r * (columns >> 1) + p] & 0x0F) + (w
    & 0x0F)) | ((((uint8_t*)data2)[r * (columns >> 1) + p] & 0xF0) + (w
    & 0xF0) > 0xF0 ? 0xF0 : (((uint8_t*)data2)[r * (columns >> 1) + p] &
    0xF0) + (w & 0xF0));
121                     }
122                 }
123             }
124             break;
125     }
126     auto end2 = high_resolution_clock::now();
127     cout << duration_cast<microseconds>(end2 - start2).count() << " us
    (reference)\t" << endl;
128 
129     for(int i = 0; i < rows; i++)
130     {
131         for(int j = 0; j < columns; j++)
132         {
133             uint8_t r = 0;
134             for(int k = 0; k < inner; k++)
135             {
136                 r = compute(r, i_j((uint8_t*)i1, i, k, inner),
    i_j((uint8_t*)i2, k, j, columns));
137             }
138             if (r != i_j((uint8_t*)data0, i, j, columns))
139             {
140                 cout << "ERROR D0" << endl;
```

```
141                 return -1;
142             }
143             if (r != i_j((uint8_t*)data2, i, j, columns))
144             {
145                 cout << "ERROR D2" << endl;
146                 return -1;
147             }
148         }
149     }
150     delete[] (uint64_t*)data0;
151     delete[] (uint64_t*)data2;
152     delete[] (uint64_t*)i1;
153     delete[] (uint64_t*)i2;
154     return (duration_cast<microseconds>(end0 - start0).count()) |
    (duration_cast<microseconds>(end2 - start2).count() << 32);
155 }
```

Listing B.5: Benchmark function for `vmm_lane_u4` and `qvmm_lane_u4`.

Acknowledgments

First, I would like to thank my advisor, Professor Marco Cococcioni, who guided me throughout the entire project and helped me with his invaluable feedback. His expertise played a crucial role in shaping this thesis.

I also want to thank all of my friends and colleagues who supported and motivated me—not just while writing this thesis, but throughout my (probably too long) bachelor's program. Special thanks to Alessio Meini and Federico Nardi for their constant encouragement, and to Andrea Covelli, whose contributions were particularly helpful in the completion of this work.

Lastly, I am grateful to my family for always being there for me. A special thanks to my father, whose expertise as an electronic engineer and remarkable patience have been priceless to me on numerous occasions.

Bibliography

[1] Wikipedia. *Adder (electronics)*. URL: https://en.wikipedia.org/wiki/Adder_(electronics).

[2] Chessprogramming. *SIMD and SWAR Techniques*. URL: https://www.chessprogramming.org/SIMD_and_SWAR_Techniques.