Building Blocks for Redundancy-Free Vector Integer Multiplication

James You
james.you@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Christopher W. Schankula
schankuc@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

Bill O'Farrell
billo@ca.ibm.com
IBM Canada
Markham, Ontario, Canada

Christopher K. Anand
anandc@mcmaster.ca
McMaster University
Hamilton, Ontario, Canada

ABSTRACT

Commercial applications of cryptography require arithmetic in prime fields with primes larger than the sizes of architected registers, and over time there will be pressure to use even larger fields to keep up with the increasing resources available for brute-force attacks and the threat that quantum computers will reach the power required for unconventional attacks. Integer multiplication is the bottleneck for most computations, and most algorithm innovations revolve around strategic composition of efficient hardware multipliers for smaller integers into algorithms for larger integer multiplication. In this paper we present an novel vector instruction which would allow hardware multipliers to be used optimally for schoolbook multiplication by flexibly grouping multiplications to avoid empty slots in vector instructions resulting in unused hardware capacity. We give general conditions for optimality, consider latency/throughput tradeoffs and optionally pair the new instruction, mammma, with a novel shift-and-sum instruction.

CCS CONCEPTS

• **Computer systems organization → Reduced instruction set computing**; **Single instruction, multiple data**.

KEYWORDS

parallel processing, computer arithmetic, vector instruction sets, single instruction multiple data, cryptography

ACM Reference Format:
James You, Christopher W. Schankula, Bill O'Farrell, and Christopher K. Anand. 2021. Building Blocks for Redundancy-Free Vector Integer Multiplication. In *CASCON '21: Conference of the Centre for Advanced Studies on Collaborative Research, November 22-25, 2021, Toronto, Canada.* ACM, New York, NY, USA, 6 pages.

CASCON'21, November 22–25 2021, Toronto, Canada

1 INTRODUCTION

The importance of cryptographic operations in support of secure communication is only accelerating with the adoption of blockchains. Different cryptographic schemes are in use, but signing and key exchange always depend heavily on modulo arithmetic in prime fields, whether directly using Diffie-Hellman or indirectly using Elliptic Curve Diffie-Hellman (ECDH) for key exchange and Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signatures. When we say "arithmetic," we are really concerned with multiplication, because it is much more expensive than addition in terms of latency, circuit size and power consumption. This paper is an attempt to define an instruction set architecture (ISA) extension which can optimally take advantage of multipliers organized into vector/SIMD instructions.

1.1 Applications

Put simply, blockchain is a mechanism for storing immutable transactions on a shared ledger. Cryptography is heavily used in blockchains to sign transactions. One example is Hyperledger Fabric, an open-source[1] permissioned blockchain platform which delivers higher throughput and better scalability [3, 12] compared to permissionless blockchains such as Bitcoin [11] and Ethereum [13]. In order to guarantee that fraudulent transactions are not accepted by other participants, each Hyperledger Fabric transaction needs to be executed and signed by multiple endorsement peers. Before putting a transaction into a block, the committing peer nodes verify the authorship of these signed transactions using the corresponding public keys.

1.2 Schoolbook multiplication

Schoolbook multiplication is a method for multiplying integers larger than machine words, often many multiples of the size of a machine word. In school, we learn to multiply numbers with multiple decimal digits by arranging the products of the first number by each of the digits of the second number into a rhombus. Digital schoolbook multiplication varies based on the instructions provided by the architecture. Scalar instruction sets typically have separate integer multiplication instructions which produce upper and lower word outputs (where w represents the bit width of a general purpose register):

[1] https://github.com/hyperledger/fabric

```
MULLO r1, r2, r3
r1 = (r1 * r3) div (2^w)

MULHI r1, r2, r3
r1 = (r1 * r3) mod (2^w)
```

With such an instruction set, our "digits" would be register values, whether 32-bit or 64-bit, and we would need two instructions to compute the two digits of each product separately. We would then add the columns with add instructions which set a carry flag or register on carry out, and accept a carry-in, requiring careful sequencing of the instructions; see Figure 1.

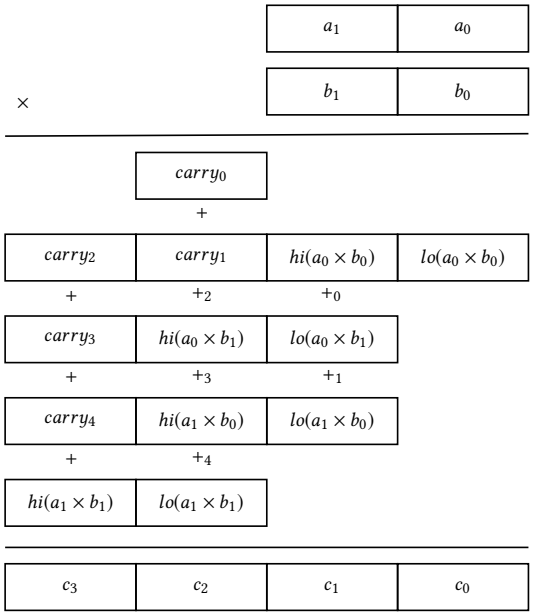

Figure 1: Diagram showing intermediate values involved in a typical schoolbook multiplication, with one rectangle for each register value computed. Each multiplication produces a low- and high-order result, and there are several carries involved.

Vector/SIMD instructions use larger register values and operate on component bits, bytes, up to full register values. This opens up the possibility of consuming values of one size and producing values of double that size, avoiding the need for high and low instructions. One such instruction is VMSL on z/Archicture[2].

1.3 VMSL: Vector Multiply Sum Logical

VMSL is an integer fused multiply-add instruction introduced as part of the IBM z14, which computes the 112-bit products of two 56-bit multiplications with the option to left shift either 112-bit intermediary product by 1 bit (to support schoolbook squaring). The resulting intermediate products after the optional shift are then added with a third 128-bit accumulator with all carry outs ignored and the resulting 128-bit sum stored in the destination register. See Figure 2. Note that although carry outs are ignored, carry outs cannot occur if less than 2^{13} VMSLs are chained together.

```
VMSL v1, v2, v3, v4, 3, m6
t1 = v2[0:56]  *v3[0:56]  *(2*m6[0])
t2 = v2[64:120]*v3[64:120]*(2*m6[1])
v1 = t1 + t2 + v4
```

Figure 2: Intrinsic-like description of VMSL. v1 is the destination register.

Inputs are typically organized in radix 2^{64} for big integer arithmetic on the z14, therefore the inputs must be converted into radix 2^{56} from radix 2^{64} before they can be used with VMSL. We call the process of converting a radix 2^{64} representation into a radix 2^{56} representation "limbification". Some authors call algorithms operating on limbs multi-precision, and others call representation with "pad" bits redundant representations. The digits or "limbs" of the new representation of the number can either be stored individually or in pairs across a double-word boundary (64-bits). The disadvantage of this approach is that bits must be rearranged into limbs, and even though efficient instructions for permuting bits and bytes, e.g., VPERM, are available with all SIMD ISA extensions, they are still extra instructions, and redundant representations must eventually be collapsed, requiring additional adds. This addition of padding increases the complexity of the computation, see Figure 3 [14].

And this hides the fact that some of the multiplies in the VMSLs are not used, requiring additional shuffling of inputs to include zeros. For example the $a_0 \times b_0$ in Figure 3 is the only product with place value 1, so it cannot be paired with another multiply. The simplified Figure 4 makes this easier to see. In this figure, the multiply-multiply-sums which require zeroed slots and wasted multiplies are outlined in red. They show that this approach results in 80% utilization of multiplies.

We propose a series of vector instructions which aim to resolve non-optimal multiplication operations. That is, there exists a selection of the proposed vector instructions for which there are no wasted multiplications.

2 DESIGN

We propose four variants of a vector instruction, which we collectively refer to as mmamma for multiply-multiply-add-multiply-multiply-add its variations mammma for multiply-add-multiply-multiply-multiply-add and mmmama for multiply-multiply-multiply-add-multiply-add. We would expect them to be treated as a single instruction with a 2-bit immediate argument. For inputs, the layout of the vector register would consist of 4 digits. In the case of outputs, the layout of the vector would consist of an *upper* and *lower* destination. This scheme would work with digits of any size. For example, with 128-bit register values, we would use 32-bit digits and 32-bit multipliers, and for 256-bit registers we would use 64-bit digits. Of course, it is

[2]The IBM Z is a modern high-performance 64-bit multi-core computer architecture. It is the successor to the line of mainframes which began in 1964 with the IBM 360. Still today, the largest banks depend on Z machines for time-sensitive and high-volume financial transactions. The currently available model, the z14, has a superscalar pipelined CPU with both scalar registers and 32 128-bit vector registers. A z14 mainframe can be configured with up to 240 cores (with a 5.2 GHz clock speed) and 32 TB of memory. There are two primary operating systems available on Z: z/OS and Linux (Ubuntu, SUSE Enterprise Server, and Red Hat Enterprise distributions). The work described in this paper was conducted on z14s running Linux. See [9].

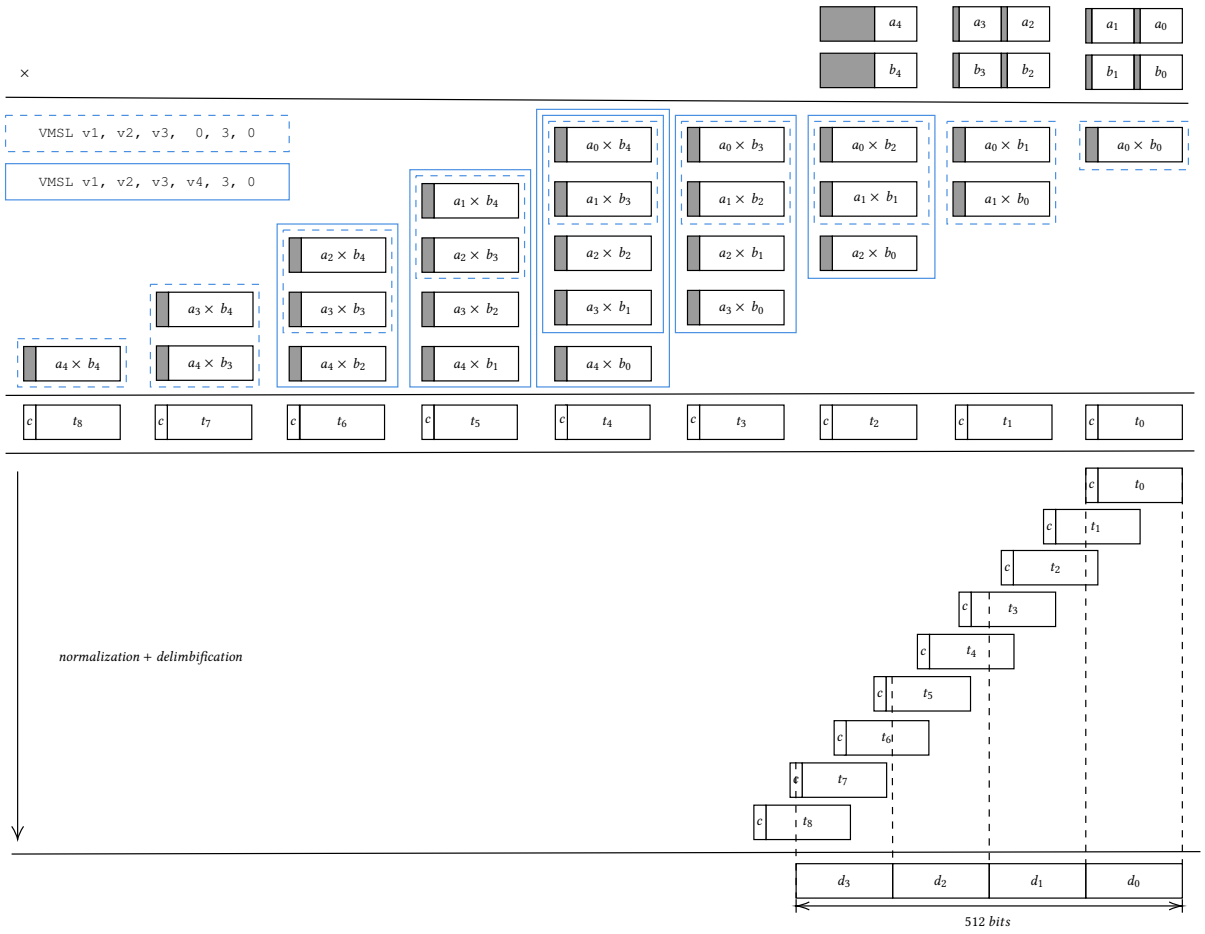

Figure 3: Schoolbook multiplication of two 256-bit inputs stored in 56-bit limbs using multiple `VMSL` instructions. The colouring indicates bit strings actively used, and used as padding to fill the register or slot widths. On the first line, the input values used have white backgrounds, and are 56-bits wide, meaning that the grey padding on the left consists of 8 bits in each of four slots, and for the first vector value 72 bits (representing the first 64-bit word plus the first 8 bits of the second slot). Within the diagonal block of intermediate products, the products in white are 112 bits wide, with 16 bits of padding shown in grey to fill the 128-bit registers.

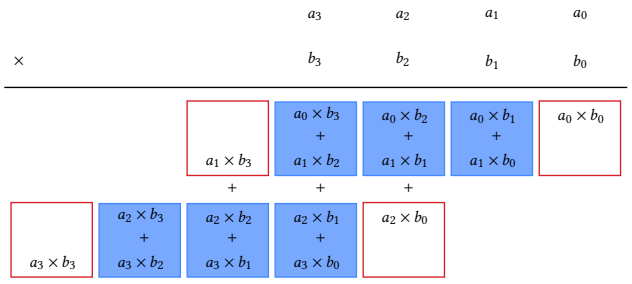

Figure 4: Grouping of vector multiply sum instructions for 4×4 digit multiplications with a `vmsl`-like fixed $a \times b + c \times d$ structure necessarily leaves four $(4/20 = 20\%)$ unused multiplication slots. This puts a maximum 80% efficiency ceiling on $n \times n$-digit multiplications.

also possible to replicate the functionality of the instructions using 32-bit digits in a 256-bit register, and so on.

It is easiest to understand the instructions as a regrouping of the multiplies in Figure 4 which we present using colours in Figure 5. In this figure, the multiplies with a common highlight colour are performed by one instruction. There are four instructions needed to multiply four-digit numbers, and each one is a different variation of the basic instruction. In addition to balancing the numbers of multiplies, combining multiplies while skipping a column in Figure 5 results in the output of contiguous bits as shown in Figure 6. This allows the carry-out from the low-multiply-adds to be carried in to the high result, requiring only one carry out per instruction. To understand this in terms of bits, we translate them into intrinsic-style specifications as we did for `VMSL` above, and include the carry ins:

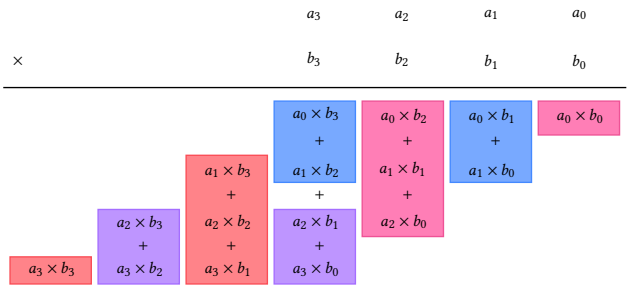

Figure 5: Grouping of vector multiply sum instructions to achieve 100% utilization of multipliers. Three novel instructions, or one novel instruction with three immediate arguments are needed. The pink instruction is `mmmama`, the blue and purple instructions are `mmamma` and the coral instruction is `mammma`.

```
MMMAMA v1, v2, v3, cOut
v1, cOut = v1 + v2[0:63]    * v3[0:63]
         + ((v2[128:195] * v3[128:195]) << 128)
         + ((v2[196:255] * v3[196:255]) << 128)
```

```
MMAMMA1 v1, v2, v3, cIn, cOut
v1, cOut = v1 + v2[196:255] * v2[128:195]
         + v1[128:195] * v2[196:255]
         + ( (v2[196:255] * v3[0:63])   << 128 )
         + ( (v2[128:195] * v3[64:127]) << 128 )
         + (cIn << 196)
```

```
MMAMMA2 v1, v2, v3, cIn, cOut
v1, cOut = ( (v2[64:127] * v3[0:63])  << 128 )
         + ( (v2[0:63]  * v3[64:127]) << 128 )
         + v2[64:127] * v3[128:195]
         + v2[0:63]  * v3[196:255]
         + ( cIn << 128 )
```

```
MAMMMA v1, v2, v3, cIn, cOut
v1, cOut = ( (v2[0:63]    * v3[0:63]) << 128 )
         + v2[128:195] * v3[0:63]
         + v2[64:127]  * v3[64:127]
         + v2[0:63]    * v3[128:195]
         + (cIn << 196)
```

For even wider multiplies, this basic pattern is tiled without any additional bit shifting and shuffling, as can be seen by comparing Figure 7 to Figure 5. In this figure, the pattern of Figure 5 is shown with grey background, and is repeated four times, with the original pattern corresponding to the low register value multiplied by the low register value. The top left copy corresponds to the low register value multiplied by the high value, and so on.

2.1 Accumulation: Latency versus Throughput

Up to this point we have optimized argument placement in order to fully utilize four multipliers in each instruction. This gives our instructions an advantage. The next design decision is how to accumulate the results from multiple `mammma` instructions. There are three possible design goals: maximizing throughput, minimizing latency and maintaining flexibility to write software with either goal.

If we want to maximize throughput, we should include a third input to `mammma` for an accumulator input. We can then chain the

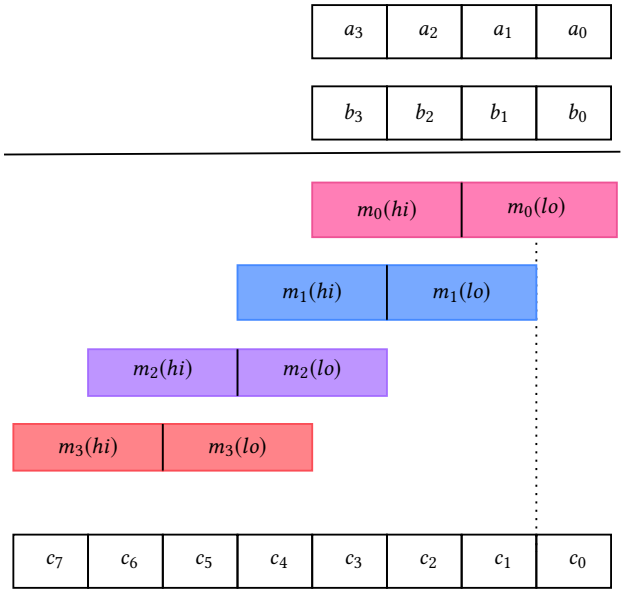

Figure 6: Another view of Figure 5 shows the results of the four `mammma` instructions with the results aligned to show their place values. Along the bottom the place values are symbolized with by the output digits $c_0, c_1, ..., c_7$. Note that there is an irregularity to the offsets, with the blue `mmamma` result shifted by one digit relative to the bubblegum `mmmama` result, while the purple `mmamma` result is shifted by two digits relative to the blue `mmamma`.
Each of the results is divided into two two-digit halves which are each the result of a 1×1-digit multiplication. The fact that the place values of the digits in the results are contiguous digits is important for carry operations, because carries out of the lower half can be incorporated into the sum of the upper half. Note that the `mmmama` instruction would not have a carry out of the lower product in the latency-optimized version of the instruction, but would have a carry out in the throughput-optimized version which has accumulator and carry inputs.

instructions together, also chaining carry inputs and outputs. This approach requires that each version of our instruction shift the accumulator by one or two digits. In Figure 6, we see that the bubblegum output m_0 must be shifted right by one digit before being added to the results of the multiplications to form the blue output m_1. The next instruction takes that blue output m_1 shifts it right by *two* digits before adding it to the next multiplies to form the purple output m_2, which is in turn shifted right by one digit before adding. The output m_3 is the high register value of the final product, but the low register value needs to be assembled from m_1, m_2 and m_3 using shifts or byte permutations. There are three problems with this approach: The additional permutations, the complexity, and overall latency involved in calculating larger products using the tiling in Figure 7. If this computation can be interleaved with another high-latency operation, then this may be the best solution, but it will be complicated.

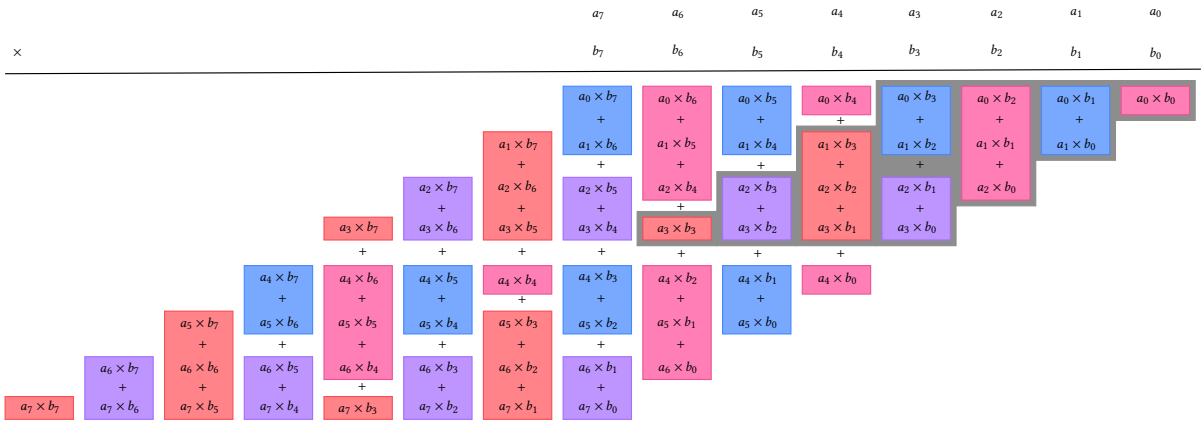

Figure 7: This diagram shows that the instruction sequence in Figure 5 can be tiled for 8×8 multiplications, and that this tiling does not require any rearrangement of the input vector register values. It is clear from the diagram that the same is true of $4n \times 4m$ multiplications, for any integers n and m.

If we want to minimize latency, we can omit the accumulator input and use existing shift instructions and vector-width add instructions which also take carries, or instructions which insert carries into register values and add instructions which do not use carries. In a modern super-scalar CPU, these low-latency instructions can be executed in parallel, starting as soon as the first product-sums are available. Alternatively, we propose adding two shift-add instructions `shaddhi` and `shaddlo`, each of which takes three register inputs, one or three carries, and produces one of the halves of the final product. These instructions must be sequenced. The first, `shaddlo`, takes m_0, m_1 and m_2 and the carry out from the calculation of m_0, shifts m_1 left by one digit, shifts m_2 left by three digits, calculates the sum and stores it in a register, and combines the carry out from the sum with the carry out from the calculation of m_0. The second, `shaddhi`, takes m_1, m_2 and m_3 and the carry out from the calculations of m_2 and m_3 and `shaddlo`, shifts m_1 right by three digits, shifts m_2 right by one digit, calculates the sum, incorporating the carries with appropriate shifts and stores it in a register. This calculation cannot carry out, because we know the result of the product fits in two register values. If computing with multiple tiles, these vector-register results can be added together with carries without any alignment operations.

If we want to maintain flexibility, we can use existing processor conventions to allow instructions to be used in both configurations. For example, some processors use a register-zero convention, in which in some argument positions, zero is treated as an immediate zero value, rather than as a reference to the value in register zero. This convention could be used for accumulator inputs, for situations where the accumulator input is not needed. At a minimum, this would eliminate the need to set an accumulator to zero for use in the first instructions.

2.2 Expected Performance

In terms of power, we are fully utilizing the multiplies, and not wasting power multiplying zeros, and even in the case where multipliers are clock-gated to preserve this power, we are not wasting dispatch slots by dispatching instructions which do not use all of their inputs. On this metric, our instructions are optimal. If implemented with 64-bit digits in 256-bit registers, it would allow for a 256-bit multiplication in four instructions which execute in parallel, and since the internal multiplies are also parallelizable, it could have the latency of a multiply instruction, followed by two instructions of low latency in sequence, if using the novel accumulate instructions. If not using the novel accumulate instructions, it would require more low-latency instructions with some parallelism.

3 RELATED WORK

Other work in this area comes in two flavours: algorithm development designed to be vendor-agnostic by leveraging common features in vector ISA extensions and the specification or application-of vendor-specific instructions to accelerate these specific workloads. An architecture-agnostic optimized message authentication code is described by Bernstein in [1]. Bernstein and Schwabe, Gueron and Krasnov describe vector extensions for accelerating integer arithmetic in [2] and [7] for the ARM and Intel x86 instruction set architectures respectively.

The second flavour concerns vendor-specific optimizations. The acceleration of specific cryptographic kernels using vendor-specific scalar and vector instructions [6], [10]. While the application of vector extensions to more general arithmetic computation such as squaring and multiplication is described by Gueron and Drucker in [4] and Edamatsu and Takashi in [5], respectively.

At the outset, we considered the design of the underlying multiplication engines out of scope, this is because multiplication engines are used by multiple instructions, both integer and floating-point, and we wanted to suggest incremental improvements to existing CPUs. That said, there is ongoing work on developing algorithms for integer arithmetic which could lead to better implementations of the underlying engines or better alternatives to schoolbook multiplication [8].

4 CONCLUSION

We have demonstrated that the design of the `mammma` instruction allows for the full use of width-w multipliers in the case of multiplying integers of length $4w$, within the constraints of an instruction on a RISC-like CPU, i.e., consuming less than four input register values and producing a single register value. It can be optimized for throughput (using chained accumulators) or for latency, using an addition tree post-multiplication, and is easy to understand with a simple diagram. In the case where an addition tree is used, we also propose `shaddhi` and `shaddlo` triple shift-add instructions, resulting in both low latency and high throughput.

Big-integer multiplication is important because it is the most expensive step in many cryptographic algorithms. Choosing an appropriate cryptographic scheme requires balancing computational cost and security. These generic instructions permit deferring scheme selection, which makes the hardware future-proof.

REFERENCES

[1] Daniel J Bernstein. 2005. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*. Springer, 32–49.

[2] Daniel J. Bernstein and Peter Schwabe. 2012. NEON Crypto. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, Emmanuel Prouff and Patrick Schaumont (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–339.

[3] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. 2016. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*. Springer, 106–125.

[4] Nir Drucker and Shay Gueron. 2018. Fast modular squaring with AVX512IFMA. *IACR Cryptology ePrint Archive* 2018 (2018), 335.

[5] Takuya Edamatsu and Daisuke Takahashi. 2020. Accelerating Large Integer Multiplication Using Intel AVX-512IFMA. In *Algorithms and Architectures for Parallel Processing*, Sheng Wen, Albert Zomaya, and Laurence T. Yang (Eds.). Springer International Publishing, Cham, 60–74.

[6] Shay Gueron and Vlad Krasnov. 2013. Fast Prime Field Elliptic Curve Cryptography with 256 Bit Primes. Cryptology ePrint Archive, Report 2013/816. https://eprint.iacr.org/2013/816.

[7] S. Gueron and V. Krasnov. 2016. Accelerating Big Integer Arithmetic Using Intel IFMA Extensions. In *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*. 32–38. https://doi.org/10.1109/ARITH.2016.22

[9] IBM Corporation. [n.d.]. *z/Architecture Principles of Operation*. IBM.

[10] Dusan Kostic and Shay Gueron. 2019. Using the New VPMADD Instructions for the New Post Quantum Key Encapsulation Mechanism SIKE. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 215–218. https://doi.org/10.1109/ARITH.2019.00050

[11] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[12] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. *CoRR* abs/1805.11390 (2018). arXiv:1805.11390 http://arxiv.org/abs/1805.11390

[13] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[14] James You, Qi Zhang, Curtis D'Alves, Bill O'Farrell, and Christopher K. Anand. 2019. Using Z14 Fused-Multiply-Add Instructions to Accelerate Elliptic Curve Cryptography. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19)*. IBM Corp., USA, 284–291.