# RISC-V Instruction Set Extensions for Multi-Precision Integer Arithmetic

A Case Study on Post-Quantum Key Exchange Using CSIDH-512

Hao Cheng
University of Luxembourg
Esch-sur-Alzette, Luxembourg
hao.cheng@uni.lu

Georgios Fotiadis
University of Luxembourg
Esch-sur-Alzette, Luxembourg
georgios.fotiadis@uni.lu

Johann Großschädl
University of Luxembourg
Esch-sur-Alzette, Luxembourg
johann.groszschaedl@uni.lu

Daniel Page
University of Bristol
Bristol, United Kingdom
daniel.page@bristol.ac.uk

Thinh H. Pham
University of Bristol
Bristol, United Kingdom
th.pham@bristol.ac.uk

Peter Y. A. Ryan
University of Luxembourg
Esch-sur-Alzette, Luxembourg
peter.ryan@uni.lu

## ABSTRACT

Multi-Precision Integer (MPI) arithmetic is a performance-critical component of many public-key cryptosystems, including besides classical ones (e.g., RSA, ECC) also isogeny-based post-quantum schemes. In this paper, we analyze and compare two widely-used MPI representations, namely full-radix and reduced-radix, for the efficient implementation of modular arithmetic operations on the 64-bit RISC-V (RV64GC) architecture. We also evaluate how the execution times of both can be further improved with Instruction Set Extensions (ISEs). The ISEs we propose are able to accelerate a CSIDH-512 class group action by a factor of 1.71 compared to a standard software implementation on a 64-bit Rocket core. This speed-up comes at the cost of a hardware overhead of about 10%.

## CCS CONCEPTS

• **Computer systems organization** → **Reduced instruction set computing**; • **Security and privacy** → **Cryptography**.

## KEYWORDS

Isogeny-Based Cryptography, Long-Integer Modular Arithmetic.

## 1 INTRODUCTION

The majority of public-key cryptosystems operate in certain algebraic structures like finite groups, rings or fields, whose elements are unsigned integers of a length of hundreds or even thousands of bits. Important examples include not only "classical" schemes such as RSA, DSA, and Diffie-Hellman, as well as elliptic-curve variants of the latter (i.e., ECDSA and ECDH), but also some more recent post-quantum cryptosystems based on isogenies between supersingular elliptic curves [5]. All these cryptosystems have in common that, at the lowest level, they perform arithmetic operations (e.g., addition, multiplication, inversion) on large integers modulo a prime or a product of primes or a near-prime [6].

Software libraries for Multi-Precision Integer (MPI) arithmetic usually represent the operands (i.e., integers of a length of $n$ bits) via arrays of $w$-bit *digits* or *limbs*, where $w$ equals the word-size of the target platform ("full-radix" representation) or is a few bits below the word-size ("reduced-radix" representation). Algorithms for MPI arithmetic operate on digit/limb arrays by executing the $w$-bit instructions supported by the processor, e.g., $w$-bit addition or $(w \times w)$-bit multiplication. The most basic approaches for MPI multiplication, namely the row-wise and column-wise technique (also known as *operand scanning* and *product scanning* [7]) have complexity $O(l^2)$ for $l$-digit operands (where $l = \lceil n/w \rceil$), i.e., the number of $(w \times w)$-bit multiply instructions to be executed grows with the square of the operand length $n$.

Depending on the operand length $n$ and the word-size $w$ of the processor, such basic techniques for MPI multiplication (and also MPI squaring and MPI modular reduction) can involve the execution of dozens, or even hundreds, of multiply instructions. This makes MPI arithmetic and, hence, most public-key cryptosystems relatively costly in software. The high execution times of public-key cryptosystems has initiated a body of research on hardware acceleration to support MPI arithmetic operations, especially the multiplication, squaring, and modular reduction. Besides classical acceleration through loosely-coupled co-processors, also various forms of hardware-software co-design have been analyzed in the literature. A promising approach is to extend the basic Instruction Set Architecture (ISA) of a general-purpose processor by a small set of "custom" instructions, tailored specifically to accelerate the most performance-critical operations of MPI arithmetic, i.e., the operations in the inner loop(s) of the arithmetic functions.

This paper describes the design and prototype implementation of ISEs for MPI arithmetic on the RISC-V architecture. RISC-V is a royalty-free and very permissively licensed ISA based on well-accepted RISC principles. The architecture is modular and comes with a minimalist base integer instruction set that contains less than 50 unique instructions [15]. In addition, different extensions have been developed in recent years to better support common application domains. One of the extensions targets cryptographic workloads and provides a set of special instructions to accelerate major symmetric cryptosystems like the AES [11]. Although some papers, e.g., [1, 10, 13], have studied *custom* ISE designs for lattice-based cryptography, however, currently 1) no *standard* extension for public-key cryptography exists, and 2) no paper proposes an ISE approach for flexible (i.e., scalable) MPI arithmetic. With the present paper we contribute to fill this gap and introduce an ISE design to speed up MPI arithmetic on RISC-V. Our design adheres to (most of) the general ISE design principles put forward by the

RISC-V community; therefore, the proposed instructions could be considered to become part of a standard extension.

The research contribution of this paper is twofold and can be summarized as follows. First, we study ISA-only implementations of MPI arithmetic on RV64GC, i.e., implementations using basic integer and multiply instructions, considering both full-radix and reduced-radix representations. We take the prime-field arithmetic of the post-quantum key exchange algorithm CSIDH [3] as case study, for which we developed and evaluated a number of highly-optimized Assembly functions. Although the RISC-V architecture does not support ALU status flags such as a carry flag, our results show that the full-radix representation is the better option. The second contribution is a proposal for two different ISEs for MPI arithmetic; one for a full-radix and the other for a reduced-radix representation. Each ISE includes a pair of novel fused multiply-add instructions and one instruction to efficiently implement the carry propagation. Using the two sets of custom instructions, we studied the question of which of the two radix representations is more suitable for acceleration via ISEs. Our experiments, carried out with a Rocket core [2], showed that 1) the reduced-radix ISEs reached better performance, and 2) ISE-supported CSIDH-512 is about 1.71× faster than the full-radix ISA-only version.

## 2 BACKGROUND

*RISC-V.* As we already stated above, a major tenet of RISC-V is modularity, i.e., the general-purpose RISC-V ISA can be enhanced with a set of special-purpose, standard or non-standard (custom) extensions. Due to these features and the availability of a diverse supporting infrastructure (e.g., development tools/utilities) from surrounding communities, a variety of RISC-V implementations exist, which are usually open-source. In this work, we define the base ISA (i.e., serving as "baseline") as RV64GC, which includes the 64-bit integer ISA RV64I *plus* the general-purpose M (multiplication), A (atomic), and C (compressed) extensions, as well as the F and D extensions for floating-point arithmetic.

*Relevant instructions.* The most relevant RV64GC instructions for MPI arithmetic include the integer addition `add`, integer subtraction `sub`, and bit-wise shift `slli`, `srli`, `srai` from RV64I, as well as the two integer multiply instructions `mul` and `mulhu` from RV64M. Notably, since RISC-V does not support a carry flag, the corresponding (i.e., overflow-related) carry propagation uses two instructions: one `sltu` from RV64I for carry-out (i.e., overflow) check, and one `add` for propagating the carry.

*Basic CSIDH facts.* Commutative Supersingular Isogeny Diffie-Hellman (CSIDH) [3] is a quantum-secure key exchange protocol using extremely short keys. It operates on supersingular elliptic curves $E$ given in Montgomery form [6] and defined over a prime field $\mathbb{F}_p$, where $p$ has the special form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$ and its factors $\ell_1 < \cdots < \ell_n$ are small odd primes. For $p \equiv 3 \bmod 8$, the $\mathbb{F}_p$-endomorphism ring satisfies $\mathrm{End}_{\mathbb{F}_p}(E) \cong \mathbb{Z}[\sqrt{-p}]$. At the core of the CSIDH key exchange is a free and transitive group action $\star : \mathrm{CL}(\mathbb{Z}[\sqrt{-p}]) \times S \to S$, where $\mathrm{CL}(\mathbb{Z}[\sqrt{-p}])$ is the ideal class group of order $\mathbb{Z}[\sqrt{-p}]$ and $S$ is the set of all supersingular elliptic curves with $\mathbb{F}_p$-endomorphism ring $\mathbb{Z}[\sqrt{-p}]$. The existence of the commutative group action is a very attractive feature, leading the authors of [3] to state that CSIDH-based key exchange "can serve as a drop-in replacement for the (EC)DH key-exchange protocol while maintaining security against quantum computers". CSIDH has gained a lot of attention over the past few years; however, its main drawback is the high execution time, with the commutative group action being the most demanding part of the protocol. The CSIDH protocol comes in three instantiations, determined by the (approximate) length of $p$, namely CSIDH-512, CSIDH-1024, and CSIDH-1792, targeting NIST PQ security levels 1, 2, and 3.

Although the work of Peikert [14] claims that the three above instantiations do not reach the targeted NIST security levels, the concrete security of CSIDH is still under debate. Nonetheless, the protocol attracts enormous interest due to its unique combination of commutativity with extremely short public keys (e.g., 64 bytes for the NIST security level 1). In this paper, we only consider the CSIDH-512 instantiation, where the prime $p$ is 511 bits long.

*Notation.* We denote multiple-precision integers with a capital letter (e.g., $A$) and its $i$-th word/limb with $a_i$. Any integer within the machine word-size is referred to by a lowercase letter such as $a$. With $x \ll y$ (resp., $x \gg y$) we denote a logical left (resp., right) shift of $x$ by $y$ bits. In addition, based on the notation in [11], the prefix `EXTS` (short for "Sign-EXTended") represents an arithmetic shift. This means $\texttt{EXTS}(x \ll y)$ and $\texttt{EXTS}(x \gg y)$ is an arithmetic left and right shift, respectively, of $x$ by $y$ bits. Finally, we write $x \parallel y$ for a concatenation of $x$ and $y$, and $x_{\{h \ldots l\}}$ for an extraction of all bits with index $h$ (the more-significant index) through $l$ (the less-significant index) inclusive from some operand $x$.

## 3 DESIGN AND IMPLEMENTATION

The high-level computations (e.g., curve and isogeny arithmetic) of CSIDH rely, in essence, on low-level arithmetic operations in the prime field $\mathbb{F}_p$. Our focus is on multiplication in $\mathbb{F}_p$ since it is particularly performance-critical. We implemented this operation through Montgomery multiplication, which is a common choice for moduli that do not have a special form [9]. Our work includes two full-radix (64 bits/digit) and two reduced-radix ($w$ bits/limb where $w < 64$ bits) implementations. The latter two actually use a radix-$2^{57}$ representation, i.e., 57 bits/limb. Two variants execute solely "native" RV64GC instructions and, therefore, we call them *ISA-only implementations*, whereas the remaining two, referred to as *ISE-supported implementations*, use besides native RV64GC also custom instructions to speed up the inner-loop operations.

### 3.1 ISA-Only Implementation

*Montgomery multiplication: High-level techniques.* A very basic way to carry out a Montgomery modular multiplication involves the separate (i.e., sequential) execution of MPI multiplication and MPI Montgomery reduction. Among the implementation options for these operations are operand/product scanning, Karatsuba's algorithm, etc. [7]. In addition, the integration of multiplication steps and modular reduction steps has been studied, which led to coarsely and finely integrated approaches, see [9] for details. The differences between these separated or integrated techniques are mainly apparent in the loop structure (e.g., number of outer and inner loops), the operations performed in the inner loop, and the number of memory accesses when the register-space is small. In our case (i.e., RV64GC), the register space is large enough to store

the operands and intermediates up to 512 bits, and we also unroll the loops fully. Therefore, the mentioned integration techniques are very similar in performance on our base ISA.

*Montgomery multiplication: Low-level optimizations.* The Montgomery multiplication techniques we implemented are based on product-scanning in some form; thus, the main building block is an operation of the form $S \leftarrow S + a_i \cdot b_j$, commonly referred to as Multiply-and-ACcumulate (MAC). This MAC operation computes a double-precision partial product $a_i \cdot b_j$ of digits/limbs from the operands $A$ and $B$, and adds it to an accumulator or a cumulative sum $S$. After several such MAC operations, a part of $S$ yields the digit/limb $r_{i+j}$ of the final result $R$. When the MPI multiplication and/or MPI reduction is implemented in a looped way, the MAC is actually performed in the inner-loop, which means it dominates the execution time of Montgomery multiplication. Therefore, the MAC deserves careful optimization at Assembly level.

**Listing 1: ISA-only full-radix MAC operation.**

```
/* Input/Output: 192-bit accumulator  e || h || l */
/* Input:         64-bit operands     a and b     */
mulhu z, a, b; mul y, a, b; add  l, l, y; sltu y, l, y;
add   z, z, y; add h, h, z; sltu z, h, z; add  e, e, z;
```

**Listing 2: ISA-only reduced-radix MAC operation.**

```
/* Input/Output: 128-bit accumulator  h || l  */
/* Input:         64-bit operands     a and b */
mulhu z, a, b; mul y, a, b; add  l, l, y; sltu y, l, y;
add   z, z, y; add h, h, z;
```

Listing 1 shows an RV64GC Assembly implementation (where each line has to be read from left to right) of the MAC operation for full-radix arithmetic, which means $S$ is held in three registers $(e, h, l)$ and the MAC computes $(e \parallel h \parallel l) \leftarrow (e \parallel h \parallel l) + a \cdot b$. The corresponding Assembly source code for the reduced-radix case (given in Listing 2) differs a bit since $S$ can now be placed in two registers ($h$ and $l$) and the computation $(h \parallel l) \leftarrow (h \parallel l) + a \cdot b$ is carried out. When considering only the MAC operation itself, the reduced-radix variant seems better as it needs fewer instructions than its full-radix counterpart (i.e., 6 versus 8). However, for the full Montgomery multiplication, the reduced-radix representation introduces a few implicit overheads. First, for common operand sizes, e.g., 512 bits, the number of limbs in the reduced-radix case is normally higher than the number of digits/words of a full-radix representation, which is unfavorable since the number of MACs increases with the square of the limb/digit count. In addition, the reduced-radix version needs a few extra instructions to align the accumulator and propagate the carries in the outer loop. On the other hand, in the full-radix setting, the proper alignment of the accumulator is "naturally" given and the carry-propagation in the outer loop is basically free. All these aspects make it very hard to predict which radix representation is more efficient on RISC-V.

*Fast reduction modulo $p$.* When an operand $A$ is known to be in the range $[0, 2p-1]$, a fast modulo-$p$ reduction can be performed (instead of the costly Montgomery reduction) to get a residue in $[0, p-1]$. The first step of this fast reduction is a MPI subtraction

**Algorithm 1:** Addition-based fast modulo-$p$ reduction.

**Input:** An operand $A \in [0, 2p-1]$ and modulus $P$.
**Output:** Fully reduced result $R \in [0, p-1]$.

1 $T \leftarrow A - P$
2 $M \leftarrow 0 - SLTU(A, P)$ `// M is either 0 or an all-1 mask`
3 $M \leftarrow M \wedge P$
4 $R \leftarrow T + M$
5 **return** $R$

**Algorithm 2:** Swap-based fast modulo-$p$ reduction.

**Input:** An operand $A \in [0, 2p-1]$ and the modulus $P$.
**Output:** Fully reduced result $R \in [0, p-1]$.

1 $T \leftarrow A - P$
2 $M \leftarrow 0 - SLTU(A, P)$ `// M is either 0 or an all-1 mask`
3 $M \leftarrow M \wedge (A \oplus T)$
4 $R \leftarrow T \oplus M$
5 **return** $R$

of the form $T = A - P$. Thereafter, depending on whether $T < 0$ (i.e., $A$ was already in $[0, p-1]$) or not, either operand $A$ or the difference $T$ is returned. This second step can be implemented to have constant execution time according to two approaches, one is *addition-based* and the other *conditional-swap-based*, shown in Algorithm 1 and 2, respectively. The operation $SLTU(A, P)$ gives either 0 or 1 as result (to detect underflows); it is actually a "side-product" of the subtraction $A - P$. Note that a fast reduction can be directly used for the $\mathbb{F}_p$-addition and also as final step of the Montgomery reduction. A variant of Algorithm 1, where line 1 is modified to $T = A - B$ (here, $B$ is another operand instead of the prime modulus $P$), can be used for $\mathbb{F}_p$-subtraction.

For full-radix representation, the addition-based method is the more efficient option on most ISAs since it can save an operation compared to a swap-based version (see line 3 in Algorithm 1 and 2). However, this does not hold for RISC-V because of the absence of a carry flag, making the addition at the end of Algorithm 1 (in line 4) rather costly. This, in turn, helps the swap-based version to become the faster option for our full-radix implementation. In the reduced-radix case, a swap-based version is the better choice for the final step of Montgomery reduction since it can avoid the carry propagation caused by an addition. But for $\mathbb{F}_p$-addition, the swap-based version is slower because it requires the sum (before reduction) to be in canonical form with strictly 57 bits/limb.

## 3.2 ISE Design

*Design guidelines.* We aim to provide an ISE design that could be considered to be part of a standard extension, and to this end we adopt the principles described in [12, Sect. 3] as guidelines. In detail, our guidelines are: 1) the ISE must use the general-purpose scalar register file to store operands; 2) the ISE must not introduce a special-purpose architectural state, nor rely on special-purpose micro-architectural state, e.g., cache, scratch-pad memory; 3) the instruction encodings use at most two source register addresses and one destination register address, but we permit instructions

**Table 1: Overview of our ISEs.**

| Functionality | ISEs | |
|---|---|---|
| | full-radix | reduced-radx |
| Integer multiply-add | `maddlu`, `maddhu` | `madd57lu`, `madd57hu` |
| Carry propagation | `cadd` | `sraiadd` |

**Table 2: Examples of some existing integer fused multiply-add instructions on ARM and Intel AVX-512.**

| Instruction | ISA/ISE | Computation | Radix |
|---|---|---|---|
| `mla` | ARM | `rd <- lo(rs1*rs2)+rs3` | F + R |
| `umlal` | ARM | `(rd2||rd1) <- (rs1*rs2)+(rd2||rd1)` | F + R |
| `umaal` | ARM | `(rd2||rd1) <- (rs1*rs2)+rd2+rd1` | F + R |
| `vpmadd52luq` | AVX-512 | `rd <- lo52(rs1*rs2)+rs3` | R |
| `vpmadd52huq` | AVX-512 | `rd <- hi52(rs1*rs2)+rs3` | R |

for the MAC operation (which is highly performance-critical) to exceptionally use an R4-type instruction format[1].

*Overview of our ISEs.* As shown in Table 1, we present two sets of ISEs in this paper: one for full-radix arithmetic and one for the reduced-radix realm. Each set has three custom instructions.

*Existing ISEs for MPI arithmetic.* An integer fused multiply-add instruction exists for almost any mainstream ISA, e.g., ARM and x64. Some of these instructions were specifically added to speed up MPI arithmetic for public-key cryptosystems; a good example are the 8-way SIMD integer fused multiply-add instructions from the AVX-512 ISA, which according to Intel are "new instructions for big number multiplication" [8]. Table 2 specifies two families of fused integer multiply-add instructions; the first is part of the ARM architecture and the second of Intel's AVX-512 ISA. These instructions were chosen to illustrate different design approaches and so we omit similar (e.g., signed) variants. In Table 2, `lo` refers to the lower half of a product; `lo52` (resp., `hi52`) takes exactly the lower (resp., higher) 52 bits of a 104-bit product; F (resp., R) is an abbreviation of full-radix (resp., reduced-radix) representation.

We analyze the instructions from three angles: 1) *Computation*: all instructions (except `umaal`) can be formalized into a Multiply-Shift-And-Add (MSA2) paradigm, which can be written in general form as `rd <- (((rs1*rs2) >> j) & m) + rs3`. We assume the length of the multiplier is in line with the register size (resp., the element width in the case of SIMD instructions). The offset $j$ and mask $m$ jointly control whether the full product or a part of it is accumulated and, in the latter case, which part exactly. In order to accumulate the full product, instructions like `umlal` or `umaal` are required perform a "widening" multiplication. The `umaal` instruction can even include two additions in the result. However, since we use standard R4-type format, accumulating the full product is not possible for our instructions. 2) *Instruction Encoding*: all these instructions use at least three source register addresses, and some instructions overwrite one or two source registers, which means a source register also serves as destination. 3) *Supported Radix*: in general, multiply-add instructions can support both full-radix and reduced-radix representation. However, `vpmadd52[luq/huq]` are exceptions as they only work properly with reduced radix.

*Our multiply-add instructions for full-radix.* When designing an ISE for fused multiply-add for full-radix MPI arithmetic, `mla` can serve as a valuable starting point. We first propose an instruction `maddlu`, which has exactly the same functionality and is used to accumulate the lower half of the 128-bit product. It makes sense to accordingly have a `maddhu` instruction to be able to accumulate the upper half of the product (of an unsigned multiplication). The details of `maddlu` and `maddhu` are shown in Figure 1. It should be noted that our `maddhu` follows the paradigm Multiply-Add-Shift-And instead of the conventional MSA2 paradigm. Because in this way, the carry propagation can be handled *inside* `maddhu` to save an explicit carry-out check (i.e., an `sltu` instruction).

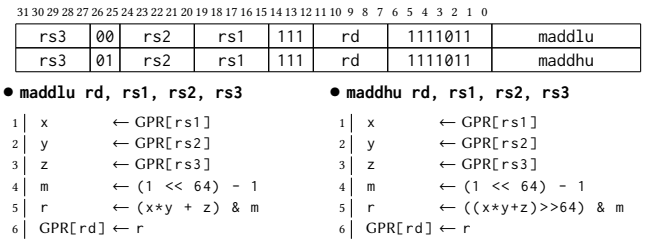


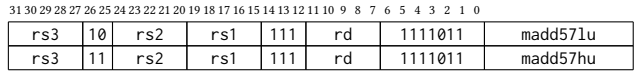

**Figure 1: Proposed multiply-add instructions for full-radix implementation.**

*Our multiply-add instructions for reduced-radix.* In the reduced-radix setting, `vpmadd52luq` and `vpmadd52huq` (both part of Intel's AVX-512IFMA) can serve as inspiration. However, as explained in [4, Sect. 3.1], when using an MPI representation with 52-bit limbs (or slightly shorter, e.g., 51 bits), one has to pay attention to the so-called *multiplier saturation problem*. For some reasons, such as delayed carry propagation during the computation of a cryptosystem, a limb can increase by few bits and, therefore, exceed the maximum input size for the multiplier. Thus, when executing an AVX-512IFMA instruction on such limbs, some extra instructions are required to bring the limbs back into the valid range. On the other hand, an implementer may also opt for a more conservative radix representation, i.e., shorter limbs, to avoid this problem.

We decided to solve the saturation problem at the instruction-design level (for our custom instructions). Concretely, rather than using a reduced-length multiplier with respect to the word size or element width (e.g., 52 instead of 64 bits in AVX-512IFMA), we employ a "full" 64-bit multiplier that produces 128-bit results and control the part of the product to be accumulated with help of the offset $j$ and the mask $m$. The offset $j$ equals the limb-length of the given MPI representation; $m$ equals $(1 \ll j) - 1$ in the instruction accumulating lower part of the product and $(1 \ll 64) - 1$ in the instruction accumulating higher part. Note that the latter has to return more bits (i.e., requires a larger $m$), because the product is usually larger than $2j$ bits, especially when the carry-propagation is delayed. More details of our custom multiply-add instructions for radix-$2^{57}$ (designed in MSA2 style) are given in Figure 2.

*Our carry-propagation instructions.* The implementation of the full-radix MAC in Listing 1 contains two propagations of carries (i.e., two `sltu` and the two corresponding `add` instructions). Even though our `maddhu` saves one carry propagation, the other one is still there and slows down the execution. Therefore, we designed an instruction to reduce the cost of this carry propagation, shown in Figure 3 (`cadd` stands for compute-Carry-then-ADD). While in the full-radix case the carries are propagated instantly, a reduced-radix implementation usually delays some carry propagations in

[1]We note that there exists a standard R4-type instruction format, defined with three source register addresses and one destination register address. The R4 format is used by the floating-point fused multiply-add (i.e., `F[N]MADD` and `F[N]MSUB`) instructions of the RV64GC ISA [15, Sect. 11.6]. However, since an R4-type instruction consumes a significant amount of encoding space, we restrict R4 to the MAC operation.



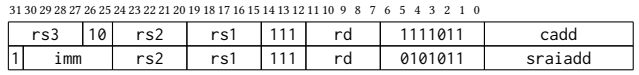

**• madd57lu rd, rs1, rs2, rs3**

```
1 x       ← GPR[rs1]
2 y       ← GPR[rs2]
3 z       ← GPR[rs3]
4 m       ← (1 << 57) - 1
5 r       ← ((x*y) & m) + z
6 GPR[rd] ← r
```

**• madd57hu rd, rs1, rs2, rs3**

```
1 x       ← GPR[rs1]
2 y       ← GPR[rs2]
3 z       ← GPR[rs3]
4 m       ← (1 << 64) - 1
5 r       ← (((x*y)>>57)&m)+z
6 GPR[rd] ← r
```

**Figure 2: Proposed multiply-add instructions for reduced-radix implementation.**




**• cadd rd, rs1, rs2, rs3**

```
1 x       ← GPR[rs1]
2 y       ← GPR[rs2]
3 z       ← GPR[rs3]
4 r       ← ((x+y)>>64) + z
5 GPR[rd] ← r
```

**• sraiadd rd, rs1, rs2, imm**

```
1 x       ← GPR[rs1]
2 y       ← GPR[rs2]
3 r       ← x + EXTS(y>>imm)
4 GPR[rd] ← r
```

**Figure 3: Proposed carry-propagation instructions.**

intermediate computations and performs a one-time propagation at the end. Though delaying the carries can save instructions, the final one-time propagation still introduces high latency and, even worse, high dependency compared to other operations. It makes therefore sense to have a custom instruction to support this final propagation in a reduced-radix implementation. Our design (see Figure 3), in essence, fuses `sari` and `add` into one instruction.

*Impact of our ISEs on software.* After utilizing `madd[l|h]u` and `cadd` in the full-radix implementation, the MAC operation takes fewer instructions. The ISE-supported full-radix MAC, shown in Listing 3, executes the operation $(e \parallel h \parallel l) \leftarrow (e \parallel h \parallel l) + a \cdot b$ as described in Sect. 3.1. Compared to the ISA-only implementation given in Listing 1, our full-radix ISEs save half of the instructions (i.e., the number of instructions decreases from eight to four). On the other hand, the ISE-based MAC for reduced-radix arithmetic (Listing 4) performs the operation $(h \parallel l) \leftarrow (h \parallel l) + a \cdot b$ via the two steps $l \leftarrow l + (a \cdot b)_{\{56\ldots0\}}$ and $h \leftarrow h + (a \cdot b)_{\{120\ldots57\}}$. The ISEs for reduced-radix implementation do not only decrease the instruction count from six (Listing 2) to two (Listing 4), but also make the accumulator automatically aligned, which saves some further cycles. Thus, one can expect the reduced-radix version to experience a higher performance gain when switching from the ISA-only to the ISE-supported version. The instruction sequence for carry propagation from a given limb $x$ to the next limb $y$ in an ISA-only radix-$2^{57}$ implementation is as follows.

```
srai z, x, 57; add y, y, z; and x, x, m;
```

The register $m$ holds the mask $2^{57} - 1$. After utilizing the custom instruction `sraiadd`, the total number of instructions for the final carry propagation decreases from three to two and, moreover, the dependency chain is weakened, as shown below.

```
sraiadd y, y, x, 57; and x, x, m;
```

**Listing 3: ISE-supported full-radix MAC operation.**

```
/* Input/Output: 192-bit accumulator  e || h || l */
/* Input:         64-bit operands     a and b     */
maddhu z, a, b, l; maddlu l, a, b, l;
cadd   e, h, z, e; add    h, h, z;
```

**Listing 4: ISE-supported reduced-radix MAC operation.**

```
/* Input/Output:  64-bit accumulators h and l */
/* Input:         64-bit operands     a and b */
madd57hu h, a, b, h; madd57lu l, a, b, l;
```

## 3.3 Hardware Implementation

*Host core.* We used the highly-configurable Rocket chip generator [2] to obtain a RISC-V compliant core for the implementation of our ISEs. Very briefly, this RV64GC core executes instructions using a 5-stage, in-order pipeline; support is included within the core for a branch prediction mechanism, and in the wider system for a 16 kB instruction cache and a 16 kB data cache. In order to support our custom instructions, two modifications were made to the core. First, an eXtended MULtiplier (XMUL) was integrated to execute the presented instructions. This XMUL unit *extends* the core's original pipelined multiplier, i.e., it supports not only the special multiply-add and carry-handling instructions, but also the base-ISA multiply instructions. All custom instructions (and also `mul[hu]`) execute in one cycle. Second, ISE-related modifications were made to the instruction decoder, which, for example, allows it to correctly provide input operands to XMUL, control XMUL's computations, and accept output operands from XMUL.

*XMUL.* XMUL was extended from the original multiplier so as to support the additional (third) input operand for the two fused multiply-add and the carry-propagation instruction, as defined in Sect. 3.2. Similar to the normal operands, the additional operand can be fetched from the forwarding path to resolve a read-after-write hazard associated with the previous instructions. Like the original multiplier, XMUL is implemented with a 2-stage pipeline (including one register stage at input operands and another at the output result) to avoid timing-critical paths. Indeed, XMUL does not extend the existing critical path and thus does not impact the clock frequency. Our implementation of the XMUL computations comes directly from the definition of the associated instructions without aiming to optimize the arithmetic circuitry. Certainly, the extension of XMUL causes increased hardware usage in relation to the original multiplier of the Rocket core.

# 4 EVALUATION

*Experimental platform.* We used an Arty A7-100T board, which hosts a Xilinx Artix-7 model XC7A100TCSG324 FPGA device[2], as experimental platform to evaluate the performance and hardware cost of our ISEs. Furthermore, we utilized Xilinx Vivado 2019.1 to synthesize the "stand-alone" design for our implementations (the default synthesis settings were applied, with no effort invested in synthesis or post-implementation optimizations). The FPGA gets a 100 MHz external clock input, which is scaled down to 50 MHz internal clock frequency for the host core itself.

[2]https://digilent.com/reference/programmable-logic/arty-a7/start

**Table 3: Results of hardware-oriented evaluation.**

| Components | LUTs | Regs | DSPs | CMOS |
|---|---|---|---|---|
| Base core | 4807 | 2156 | 16 | 428680 |
| Base core + ISE (full-radix) | 5019 | 2390 | 16 | 483248 |
| Base core + ISE (reduced-radix) | 5223 | 2352 | 16 | 495290 |

**Table 4: Results of software-oriented evaluation (execution times of CSIDH-512 operations are given in clock cycles).**

| Operation | Full-radix | | Reduced-radix | |
|---|---|---|---|---|
| | ISA-only | ISE-sup. | ISA-only | ISE-sup. |
| Integer multiplication | 608 | 371 | 625 | 303 |
| Integer squaring | 440 | 371 | 398 | 216 |
| Montgomery reduction | 730 | 469 | 818 | 389 |
| Fast modulo-$p$ reduction | 107 | 107 | 112 | 104 |
| $\mathbb{F}_p$-addition | 163 | 163 | 148 | 132 |
| $\mathbb{F}_p$-subtraction | 143 | 143 | 139 | 123 |
| $\mathbb{F}_p$-multiplication | 1446 | 954 | 1561 | 799 |
| $\mathbb{F}_p$-squaring | 1279 | 951 | 1334 | 712 |
| CSIDH group action | 701.0 M | 502.9 M | 736.2 M | 411.1 M |
| | 1.00× | 1.39× | 0.95× | 1.71× |

*Hardware evaluation.* Table 3 presents a summary of synthesis results for each of the two ISE designs. We give the (cumulative) hardware costs in terms of the number of LUTs, Regs, DSPs, and CMOS. The costs are compared to the RV64GC core alone, and so excludes the wider system; doing so seems more representative in that, for example, the caches would dominate otherwise. In line with our definition of base ISA (see Sect. 2) is what we term the *base core* (i.e., a core serving as "baseline" for our work), which is a 64-bit Rocket core that supports only RV64GC. We extend this RV64GC core with support for our custom instructions to obtain what we term an *extended core*. The implementation of full-radix (resp., reduced-radix) ISEs entails an increase of 4% (resp., 9%) in the number of LUTs, as well as an increase of 11% (resp., 9%) in the number of Regs, on the extended core versus the base core.

*Software evaluation.* Using the source code[3] from the designers of CSIDH, we developed a total of four different implementations of CSIDH-512. All implementations are based on the same code for the high-level computations, but the low-level $\mathbb{F}_p$-arithmetic uses (constant-time) Assembler functions, which we wrote from scratch for both the ISA-only and the ISE-supported version. The execution times of operations at different arithmetic layers of the CSIDH-512 protocol are summarized in Table 4. Our experiments showed that product-scanning is more efficient than Karatsuba's algorithm for MPI multiplication, and so we used the former. The fastest ISA-only implementation, i.e., the full-radix version, is the baseline for comparison in Table 4. In the ISA-only case, the full-radix implementation is faster for multiplication and squaring in $\mathbb{F}_p$ due to the smaller number of digits (resp., limbs) compared to reduced-radix, but slower for addition/subtraction in $\mathbb{F}_p$ because of costly carry checks and propagations (while the reduced-radix implementation can just delay the carry propagation). But when using our ISEs, the reduced-radix multiplication and squaring in $\mathbb{F}_p$ become faster than the full-radix versions. This performance gain for multiplication/squaring propagates all the way up to the highest arithmetic layer, i.e., the CSIDH group action. Thanks to our ISEs, the reduced-radix option becomes the more efficient one and the execution time of the CSIDH-512 group action decreases from 701.0 M to 411.1 M cycles, i.e., by a factor of 1.71×.

[3] https://csidh.isogeny.org/software.html

## 5 CONCLUSIONS

We provided some new insights into the efficient implementation of MPI arithmetic (i.e., addition, multiplication, reduction) on the RV64GV architecture, both with and without ISEs. Focussing on implementations that use only the base RV64GC instructions, we analyzed the impact of full-radix versus reduced-radix representation of operands. Intuitively, one would expect a reduced-radix implementation to outperform its full-radix counterpart since the RISC-V ISA lacks an "add-with-carry" instruction. However, the results we obtained show that, for 512-bit operands (i.e., CSIDH-512), using full-radix representation is the better option. We then designed a small set of custom instructions in order to accelerate the execution of MPI arithmetic, whereby we focused mainly on the inner loop of Montgomery multiplication. Using our ISEs, we again analyzed the performance of the two representations, and concluded that the reduced-radix representation is more suitable for ISE-supported MPI arithmetic on an RV64GC processor. This is again somewhat counterintuitive since almost all previous ISEs for MPI arithmetic were designed for full-radix representation.

**Acknowledgements.** This work was supported, in part, by the Lux4QCI project and the Luxembourg National Research Fund via the PABLO project (grant ref. 16326754) and, also in part, by the EPSRC through grant EP/R012288/1 under the RISE programme and Innovate UK through project 10065634 (SCHEME).

## REFERENCES

[1] E. Alkim, H. Evkan, N. Lahr, et al. 2020. ISA extensions for finite field arithmetic: Accelerating Kyber and NewHope on RISC-V. In *CHES 2020*. IACR, 219–242.
[2] K. Asanović, R. Avizienis, J. Bachrach, et al. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. University of California, Berkeley.
[3] W. Castryck, T. Lange, C. Martindale, et al. 2018. CSIDH: An efficient post-quantum commutative group action. In *ASIACRYPT 2018 (LNCS 11274)*. Springer, 395–427.
[4] H. Cheng, G. Fotiadis, J. Großschädl, et al. 2022. Highly Vectorized SIKE for AVX-512. In *CHES 2022*. IACR, 41–68.
[5] L. De Feo, D. Jao, and J. Plût. 2014. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology* 8, 3 (2014), 209–247.
[6] S.D. Galbraith. 2012. *Mathematics of Public Key Cryptography*. Cambridge University Press.
[7] D.R. Hankerson, A.J. Menezes, and S.A. Vanstone. 2004. *Guide to Elliptic Curve Cryptography*. Springer.
[8] Intel Corporation. 2022. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Technical Report. Intel Corporation.
[9] Ç.K. Koç, T. Acar, and B.S. Kaliski Jr. 1996. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro* 16, 3 (1996), 26–33.
[10] H. Li, N. Mentens, and S. Picek. 2022. A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-Kyber. In *DAC 2022*. ACM, 733–738.
[11] B. Marshall. 2022. *RISC-V Cryptographic Extension Proposals*. Technical Report Volume I: Scalar & Entropy Source Instructions (version 1.0.1).
[12] B. Marshall, G.R. Newell, D. Page, et al. 2021. The design of scalar AES instruction set extensions for RISC-V. In *CHES 2021*. IACR, 109–136.
[13] P. Nannipieri, S. Di Matteo, L. Zulberti, et al. 2021. A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms. *IEEE Access* 9 (2021), 150798–150808.
[14] C. Peikert. 2020. He gives C-sieves on the CSIDH. In *EUROCRYPT 2020 (LNCS 12106)*. Springer, 463–492.
[15] A. Waterman and K. Asanović. 2019. *The RISC-V Instruction Set Manual*. Technical Report Volume I: User-Level ISA (Version 20191213).