When Instruction Set Extensions Change Algorithm Design: A Study in Elliptic Curve Cryptography

Johann Großschädl
jgrosz@iaik.tugraz.at

Stefan Tillich
stillich@iaik.tugraz.at

Graz University of Technology
Institute for Applied Information Processing and Communications (IAIK)
Inffeldgasse 16a, A–8010 Graz, Austria

Paolo Ienne
Paolo.Ienne@epfl.ch

Laura Pozzi
Laura.Pozzi@epfl.ch

Ajay K. Verma
Ajay.Verma@epfl.ch

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland

ABSTRACT

In recent years, processor customization has matured to become a trusted way of achieving aggressive performance with limited cost/energy in embedded applications. In particular, instruction set extensions (ISEs) have been proven very effective in many cases. A large body of work exists today on creating algorithms that can select efficient ISEs given an application source code: ISE automation is paramount for increasing the efficiency of design teams. In this paper we show that an additional motivation to automate the ISE process is to help *algorithmic design*: the availability of ISE can have a dramatic impact on the effectiveness of different algorithmic choices to implement identical or equivalent functionality. Algorithm designers need fast feedbacks on the ISE-ability of various algorithmic flavors. We use a case study in elliptic curve (EC) cryptography to prove the following contributions: (1) ISE can reverse the relative interest of different algorithm versions and (2) automatic ISE, even without predicting speedups as precisely as detailed simulation can, is able to show exactly the trends that the algorithm designer should follow.

1. INTRODUCTION

One of the most successful ways to use processors for complex programmable *System-on-Chips (SoCs)* is to take a basic processor architecture and modify it to suit better the application-domain at hand. This opportunity for customization ranks among the most interesting differences between general-purpose computing—where, to a very large extent, compute power is the all-dominating parameter which drives evolution in a single direction—and embedded computing—where such one-fits-all strategy is not optimal: embedded SoC design goals are much more articulated and may express some minimal level of performance as a design constraint and energy consumption or silicon real-estate as the parameters to minimize.

If the generation from scratch of custom processors has never been very practical, several customizable architectures have now been on the market with some success [25, 13, 10], based on equally customizable tool chains. Roughly, these processor cores accept ISEs in the form of application-specific functional units as shown in Figure 1. Various processors differ in the number of operands they can supply in a single cycle from the register file, in the number of results they can write back to it, in the availability of memory ports, and in the existence of architecturally visible registers in the additional units.

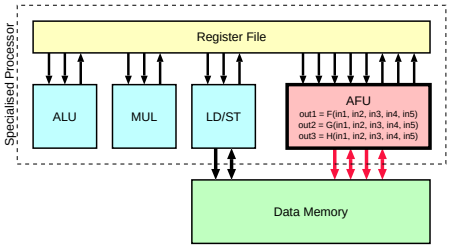

Figure 1: Typical extensible processor with a five-input three-output application-specific functional unit.

A large body of work exists today on automating the process of customization, e.g., on creating algorithms that can select efficient ISEs given an embedded application source code. While ISE has been seen so far essentially as an efficient paradigm for embedded application acceleration, and automatic ISE as a useful tool for fast and effective exploration of architectural choices, in this paper we claim and show that an additional motivation to automate the ISE process is to *help algorithmic design*. In many application areas, e.g., elliptic curve cryptography, different algorithmic choices arise to implement identical or equivalent functionality, e.g., use of prime fields or binary fields for arithmetic. The effectiveness of these options is typically evaluated on standard microprocessors, while we claim that they should be evaluated always bearing in mind the potentiality of ISE. In fact, we show in this paper that availability of ISEs can have a dramatic impact on the effectiveness of the different application algorithms at choice, and that it can completely redirect algorithm design. We also show that state of the art algorithms for automatic ISE selection are able to show ex-

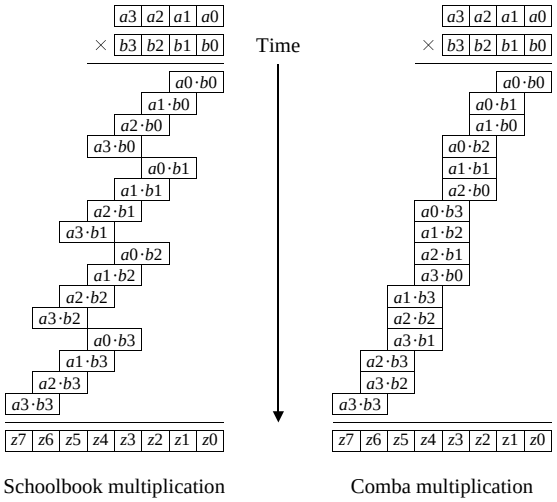

Figure 2: Schoolbook and Comba multiplication.

actly the correct trends that the algorithm designer should follow.

The rest of the paper is organized as follows: in Section 2 and 3 we overview elliptic curve (EC) cryptography and its underlying arithmetic operations. In Section 4 we study ISEs on different algorithmic flavors of EC cryptography, and in Section 5 we measure and recognize that algorithm importance is reversed in many real cases. We finally show that classic automatic ISE algorithms can predict correctly the same trends. Section 7 concludes the paper and proposes ways forward.

2. ELLIPTIC CURVE CRYPTOGRAPHY

Public-key cryptography is an integral part of virtually all modern security protocols like SSL or IPSec [19]. The recent years have seen a growing interest in *elliptic curve (EC) cryptography*, a special variant of public-key cryptography characterized by a good balance between security and performance [4]: compared to their traditional counterparts like RSA, EC systems can use much shorter keys (in the range of 160–512 bits vs. 1024 and more) to guarantee a certain level of security. Today, EC cryptography is well on its way of becoming the de-facto standard for public-key services on mobile devices [16].

From a mathematical point of view, EC systems operate in the group of points of an EC defined over a finite field [4]. Several standard bodies recommend to use either a *prime field* GF(p) or a *binary field* GF(2^m) for the implementation of EC cryptography. The elements of a prime field GF(p) are nothing else than the integers $0, 1, \ldots, P-1$, whereas the elements of a binary field GF(2^m) are usually represented by binary polynomials of degree up to $m-1$. Efficient implementation of the field arithmetic (in particular the field multiplication) is crucial for the overall performance of an EC cryptosystem. The arithmetic operations are very computation-intensive since the operands have a length of ≥ 160 bits. Moreover, certain arithmetic operations, such as the multiplication in binary fields, are not very well supported by general-purpose processors. This motivated a number of micro-processor vendors to extend their instruction set architectures by special instructions for efficient arithmetic

Algorithm 1: Schoolbook multiplication.

Input: $A = (a_{s-1}, \ldots, a_1, a_0)$ and $B = (b_{s-1}, \ldots, b_1, b_0)$.
Output: Product $Z = A \cdot B = (z_{2s-1}, \ldots, z_1, z_0)$.
1: $Z \leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: $\quad u \leftarrow 0$
4: $\quad$ **for** j from 0 by 1 to $s-1$ **do**
5: $\quad\quad (u, v) \leftarrow a_j \cdot b_i + z_{i+j} + u$
6: $\quad\quad z_{i+j} \leftarrow v$
7: $\quad$ **end for**
8: $\quad z_{i+s} \leftarrow u$
9: **end for**
10: **return** $Z = (z_{2s-1}, \ldots, z_1, z_0)$

in finite fields; two familiar examples are SmartMIPS [21] and the ARM SecurCore architecture [2].

3. ARITHMETIC ALGORITHMS

Formally, a *finite field* or *Galois field* can be described as a finite set of elements on which two operations—generally called addition and multiplication—are defined such that the field axioms hold [17].

3.1 Multiplication in Prime Fields

The arithmetic in a prime field GF(p) is the conventional modular arithmetic, i.e., addition and multiplication of integers modulo the prime P. However, the length of the operands may exceed the wordsize of the processor by an order of magnitude and long integers can be represented by arrays of unsigned integers.

The following *notation* is used throughout this paper. Uppercase letters represent field elements (long integers), while lowercase letters, usually indexed, refer to the individual words of a field element. We denote the bitlength of field elements by n and the wordsize of the processor by w. Consequently, an n-bit integer A can be written as

$$A = \sum_{i=0}^{s-1} a_i \cdot 2^{i \cdot w} = a_{s-1} \cdot 2^{(s-1) \cdot w} + \cdots + a_1 \cdot 2^w + a_0,$$

whereby s is the number of words $\lceil n/w \rceil$ and all words a_i are in the range of $0 \leq a_i \leq 2^w - 1$. For example, a 192-bit integer A can be represented by an array of six words on a 32-bit processor, i.e., $A = (a_5, a_4, a_3, a_2, a_1, a_0)$.

The multiplication of elements of a prime field GF(p) is performed in two steps: multiplication of two s-word integers A and B, yielding a $2s$-word product, and reduction of this product modulo the s-word prime P. The product $Z = A \cdot B$ can be calculated by generation and addition of partial products $a_j \cdot b_i$:

$$Z = A \cdot B = \sum_{j=0}^{s-1} \sum_{i=0}^{s-1} a_j \cdot b_i \cdot 2^{(i+j) \cdot w} = \sum_{k=0}^{2s-1} z_k \cdot 2^{k \cdot w}.$$

There exist two different algorithms to realize such long integer multiplication: the *schoolbook multiplication* [19] and the *Comba multiplication* [8]. Both algorithms require exactly s^2 single-precision multiplications for s-word operands, but they differ in the order in which they process the partial products and in the number of load/store operations.

Algorithm 2: Comba multiplication.

Input: $A = (a_{s-1}, \ldots, a_1, a_0)$ and $B = (b_{s-1}, \ldots, b_1, b_0)$.
Output: Product $Z = A \cdot B = (z_{2s-1}, \ldots, z_1, z_0)$.

```
(t, u, v) ← 0
for i from 0 by 1 to s − 1 do
    for j from 0 by 1 to i do
        (t, u, v) ← (t, u, v) + a_j · b_{i−j}
    end for
    z_i ← v
    v ← u, u ← t, t ← 0
end for
for i from s by 1 to 2s − 2 do
    for j from i − s + 1 by 1 to s − 1 do
        (t, u, v) ← (t, u, v) + a_j · b_{i−j}
    end for
    z_i ← v
    v ← u, u ← t, t ← 0
end for
z_{2s−1} ← v
return Z = (z_{2s−1}, ..., z_1, z_0)
```

Algorithm 3: Shift-and-XOR multiplication.

Input: Binary polynomials $a(t)$ and $b(t)$ of degree $m-1$.
Output: Product $z(t) = a(t) \otimes b(t)$ of degree $2m-2$.

```
z(t) ← 0
for i from m − 1 by 1 downto 0 do
    z(t) ← z(t) · t + a(t) · β_i
end for
return z(t)
```

Schoolbook Multiplication

The schoolbook multiplication (Algorithm 1) is essentially the standard pencil-and-paper multiplication [19]. This algorithm has a nested loop structure with a relatively simple inner loop that performs the bulk of computation. In any iteration of the inner loop, an operation of the form $a \cdot b + z + u$ is carried out, whereby a, b, z, and u are all w-bit words. The tuple (u, v) denotes a $2w$-bit quantity, i.e., $(u, v) = u \cdot 2^w + v$. It can be easily shown that the result of $a \cdot b + z + u$ is always a $2w$-bit quantity.

The inner loop is iterated exactly s^2 times, and hence the number of $(w \times w)$-bit (i.e., single-precision) multiplications is also s^2. Besides the single-precision multiplication and additions, two load operations (for a_j and z_{i+j}) and one store operation (for z_{i+j}) take place in the inner loop. The word b_i is loaded in the outer loop and can be kept in a general-purpose register.

Comba Multiplication

Comba multiplication, shown in Algorithm 2, forms the product $Z = A \cdot B$ by computing each word z_i of the result Z at a time, starting with the least significant word z_0. Algorithm 2 contains two nested loops; the first calculates the s least significant words of Z, while the second forms the words z_s to z_{2s-1}. The differences to the schoolbook algorithm are the order in which the partial products are generated and that the store operations are deferred to the outer loop. However, the number of single-precision multiplications is still s^2. Comba's method requires fewer memory accesses since it writes a word of the result to memory only after it has been completely calculated.

In each iteration of an inner loop, two single-precision words are multiplied and the $2w$-bit product $a \cdot b$ is accumulated. This cumulative sum is held in the three single-precision words t, u, and v, whereby the triple (t, u, v) represents the quantity $t \cdot 2^{2w} + u \cdot 2^w + v$. In summary, the two inner loops of Comba multiplication are iterated s^2 times cumulatively, and each iteration performs a multiply-accumulate operation and loads two words from memory. The store operations are performed in the outer loops.

3.2 Multiplication in Binary Fields

The binary extension field $\mathrm{GF}(2^m)$ can be viewed as a vector space of dimension m over $\mathrm{GF}(2)$. Consequently, there exist m linearly independent elements, called a basis, such that any element of $\mathrm{GF}(2^m)$ can be written uniquely as a linear combination of basis elements. In this paper, we shall represent the elements of a binary extension field $\mathrm{GF}(2^m)$ by binary polynomials of degree up to $m-1$, i.e.,

$$a(t) = \sum_{i=0}^{m-1} \alpha_i \cdot t^i = \alpha_{m-1} \cdot t^{m-1} + \cdots + \alpha_1 \cdot t + \alpha_0$$

with coefficients α_i from the subfield $\mathrm{GF}(2) = \{0, 1\}$. For example, the field $\mathrm{GF}(2^3)$ has eight elements, which are the following binary polynomials.

$$\mathrm{GF}(2^3) = \{0, 1, t, t+1, t^2, t^2+1, t^2+t, t^2+t+1\}.$$

Any element $a(t) \in \mathrm{GF}(2^m)$ can be written as a bit-string consisting of its m coefficients, i.e., $a(t) = (\alpha_{m-1}, ..., \alpha_1, \alpha_0)$. The binary fields used in elliptic curve cryptography have a degree of $m \geq 160$ and can be stored in arrays of $s = \lceil m/w \rceil$ words. In general, the word a_i of a polynomial $a(t)$ consists of the w coefficients $\alpha_{i \cdot w}$, $\alpha_{i \cdot w+1}$, ..., $\alpha_{i \cdot w+w-1}$. We only consider here one of the bases possible to represent binary extension fields: the polynomial basis representation.

Shift-and-XOR Multiplication

Multiplication in $\mathrm{GF}(2^m)$ involves multiplying the two field elements (binary polynomials) together, yielding a binary polynomial of degree $2m-2$, and then finding the residue modulo the irreducible polynomial $p(t)$. The standard algorithm for multiplying two binary polynomials is the so-called *Shift-and-XOR* method (shown in Algorithm 3), which is similar to the shift-and-add algorithm for the multiplication of integers. Algorithm 3 forms the product $z(t) = a(t) \otimes b(t)$ by scanning the coefficients of the multiplier polynomial $b(t)$ from β_{m-1} to β_0 and adding the ordinary partial product $a(t) \cdot \beta_i$ to the intermediate result $z(t)$. Before adding $a(t) \cdot \beta_i$, the intermediate result $z(t)$ must be left-shifted by 1 bit. After m steps, $z(t)$ is actually the product of $a(t)$ and $b(t)$.

Despite Algorithm 3's simplicity, the polynomials have a very high degree and are stored in an array of single-precision words. Shifting an array of words one bit to the left is costly since, for any word z_i, the most significant bit (MSB) must be extracted before the shift operation, and thereafter the MSB of the neighboring word z_{i-1} must be set at the least significant bit (LSB) position of the word z_i. Furthermore, it must be considered that the shift-and-XOR algorithm processes the multiplier-polynomial $b(t)$ one bit at a time: the inner loop of an implementation of the algorithm is iterated $32s^2$ times if a single word consists of 32 coefficients.

Optimized variants of the classical shift-and-XOR algorithm, such as the *left-to-right comb method*, use look-up

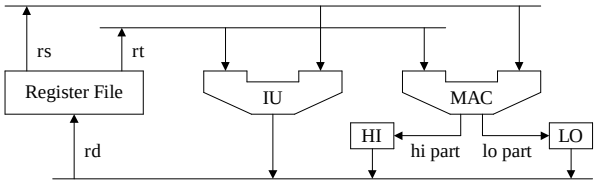

Figure 3: MIPS32 datapath with IU and MAC.

tables to reduce the number of shift and XOR operations [18]. The table entries are multiples of $a(t)$, which means that the look-up table must be computed on the fly. This makes it necessary to find a trade-off between the size of the look-up table (and the cost for computing the table entries) on the one hand and the speed-up achieved through table look-ups on the other hand. Results from previous work [18, 14] indicate that the maximum performance is reached with a look-up table containing 16 entries (i.e., 16 multiples of $a(t)$). In this case, four coefficients of $b(t)$ are processed at a time, thereby reducing the number of loop iterations from $32s^2$ to $8s^2$.

Word-Level Multiplication

Virtually all modern processors provide instructions for a $(w \times w)$-bit multiplication of integers, but not for binary polynomials. Some researchers proposed to emulate this missing instruction in software with the help of shifts and XORs [15]. This instruction, which we call MULGF2, emulated in software or executed in hardware, makes it possible to apply algorithms for long integer arithmetic, such as the schoolbook or Comba multiplication [11].

4. IMPLEMENTATION ON MIPS32

Typical software implementations of an EC cryptosystem spend most of the execution time in the inner loop of the field multiplication. Speeding up these critical code sections (e.g., through hand-crafted assembly code or dedicated instruction set extensions) can result in a tremendous performance gain. In the following we discuss the efficient implementation of diverse algorithms for field multiplication on a MIPS32 processor assuming also the possibility of extending the native instruction set.

The MIPS32 architecture is a superset of the MIPS I and MIPS II instruction set architectures and incorporates new instructions for standardized DSP operations like "multiply-and-add" (`MADD`) [20]. MIPS32 specifies the result of multiply instructions, such as `MULT` and `MULTU`, to be placed into two result accumulation registers, referenced by the names `HI` and `LO` (see Figure 3). By using the `MFHI` (move from HI) and `MFLO` (move from LO) instructions, these values can be transferred to general-purpose registers. The multiply-and-add (`MADD`) instruction multiplies two 32-bit words and adds the product to the 64-bit concatenated values in the `HI/LO` register pair. Then, the result is written back to `HI/LO`.

There exist a large number of processors compliant to the MIPS32 architecture. We chose the MIPS32 4Km for our performance evaluation because it contains a fast integer multiplier, which is an important feature for cryptographic applications. Main characteristics of the 4 Km are a 5-stage pipeline with branch control and single-cycle execution for most instructions, a multiply/accumulate (MAC) unit with a (32×16)-bit multiplier, and up to 16 kB of instruction and data cache. The MAC unit has a separate pipeline that works in parallel with the integer unit (IU) pipeline, but does not stall when the IU pipeline stalls. Long-running multiply instructions (i.e., multiply instructions which need more than one cycle to complete) can be partially masked by IU instructions.

```
l1: lw    $t0, 0($t1)   # load A[j]
    addiu $t1, $t1, 4   # increment A pointer
    multu $t0, $t4      # multiply A[j] by B[i]
    lw    $t2, 0($t3)   # load Z[i+j]
    maddu $t5, $t7      # add previous U to prod.
    maddu $t2, $t7      # add Z[i+j] to product
    addiu $t3, $t3, 4   # increment Z pointer
    mflo  $t6           # read V
    mfhi  $t5           # read U
    sw    $t6, -4(t3)   # write V to Z[i+j]
    bne   $t1, $t8, l1  # branch to l1 if j != s
```

Figure 4: Inner loop of schoolbook multiplication.

4.1 Multiplication in Prime Fields

The execution time of a multiplication (or a squaring) in a prime field GF(p) is proportional to s^2, whereby s is the number of 32-bit words needed to store an element of the field. Both the schoolbook and Comba's algorithm execute single-precision additions and multiplications in their inner loops. However, the inner loop of the schoolbook algorithm can be better optimized on MIPS32 processors.

Inner Loop of Schoolbook Multiplication

The inner loop of the schoolbook method (see Algorithm 1) performs an operation of the form $a \cdot b + z + u$, whereby all four operands are 32-bit words. However, it must be considered that adding a 32-bit word to a 64-bit quantity may produce a carry which needs to be processed properly, and therefore the two additions are actually double-precision additions. A recent whitepaper [5] by MIPS Technologies recommends to use the `MADDU` instruction to add the 32-bit words z and u to the 64-bit product. Figure 4 illustrates a highly-optimized assembly implementation of the inner loop of the schoolbook method (see [5] for a detailed description). In summary, a MIPS32 4Km processor executes the instruction sequence shown in Figure 4 in 11 clock cycles, provided that the load/store instructions hit the data cache.

Custom Instruction for Schoolbook Multiplication

The inner loop of the schoolbook method performs a single-precision multiplication and two additions. Obviously, the highest performance gain is achieved when all operations of the inner loop are combined into one custom instruction, which was first proposed in [9]. We call this custom instruction `MADDL` as in [12]. The multiply/accumulate unit of a MIPS32 processor can be easily modified to provide the extra functionality for this instruction. However, the modifications may entail a slight increase in area and delay.

The availability of the `MADDL` instruction allows to simplify the assembly code shown in Figure 4. While the original implementation consists of 11 instructions, the new version with `MADDL` implements the same functionality with only 7 instructions (see [12] for further details).

```
l1: lw    $t0, 0($t1)   # load A[j]
    lw    $t2, 0($t3)   # load B[i-j]
    addiu $t1, $t1, 4   # increment A pointer
    maddu $t0, $t2      # (HI|LO)=(HI|LO)+A*B
    addiu $t3, $t3, -4  # decrement B pointer
    bne   $t3, $t4, l1  # branch if j != i
```

Figure 5: Inner loop of Comba multiplication.

Inner Loop of Comba Multiplication

The inner loop of Comba's multiplication technique, shown in Algorithm 2, performs a "classical" multiply/accumulate (MAC) operation, i.e., two 32-bit words are multiplied and the product is added to a running sum. At a first glance, it seems that the `MADDU` instruction provides exactly the functionality needed to implement this operation. However, the problem is that the accumulator and `HI/LO` register pair of a standard MIPS32 core is only 64 bits wide, and thus the MAC unit is not able to sum up several 64-bit products without overflow and loss of precision. According to our experiments, the inner loop of Comba's algorithm can not be executed in less than 18 clock cycles on a MIPS32 4Km core. The 64-bit accumulator is a significant drawback for the implementation of multiple-precision multiplication.

Custom Instruction for Comba Multiplication

All limitations of MIPS32 can be easily mitigated by tailoring the MAC unit towards the needs of multiple-precision arithmetic. The Comba algorithm requires a MAC unit with a "wide" accumulator so that several 64-bit products can be summed up without loss of precision. For instance, extending the accumulator by eight guard bits means that we can accumulate up to 256 products, which is sufficient for EC cryptography. Moreover, register `HI` must be able to accommodate 40 bits instead of 32. The extra hardware cost is negligible, and a slightly longer critical path in the MAC unit's final adder is no concern for most applications.

On a MIPS32 processor with a "wide" accumulator, the inner loop of Comba multiplication can be implemented as shown in Figure 5. The two `ADDIU` instructions, which do simple pointer increments, can be used to fill load or branch delay slots, respectively. Any iteration of the loop requires only six clock cycles to complete, even on a MIPS32 core with a (32×16)-bit multiplier, since an IU instruction can be executed during the second cycle of the `MADDU` instruction.

4.2 Multiplication in Binary Fields

All performance-critical inner loop operations of the multiplication algorithms for binary polynomials execute only four types of instructions: loads, stores, shifts and XORs.

Inner Loop of Shift-and-XOR Multiplication

The inner loop of Algorithm 3 performs simple operations like loads, stores, shifts, and XORs on 32-bit words (see [14] for for a detailed description). While the inner loop operation is quite fast on MIPS32 processors (less than 10 cycles), the performance of the algorithm suffers from the large number of iterations. The MIPS32 architecture has no special features from which the inner loop operation could profit. Therefore, a straightforward C implementation of the inner loop reaches almost the same performance as handcrafted assembly code.

Custom Instruction for Shift-and-XOR Multiplication

The inner loop operation can be accelerated by combining a shift and an XOR operation to a custom instruction. This custom instruction is simply realized by passing one operand of the XOR instruction through a barrel shifter, similar to the ARM architecture. Using the custom instruction saves two clock cycles in any iteration of the inner loop.

Inner Loop of Word-Level Multiplication

The word-level algorithms are adoptions of the schoolbook and the Comba multiplication for binary polynomials. Both have in common that their performance depends primarily on the efficiency of the MULGF2 instruction. However, as this instruction is not available on MIPS32 processors, it must be emulated in software using shift and XOR instructions. Despite our best efforts, we were not able to reduce the execution time of the MULGF2 operation to less than 190 cycles on a MIPS32 core.

Custom Instruction for Word-Level Multiplication

The obvious way to accelerate the word-level algorithm is to implement the MULGF2 instruction in hardware, e.g., on a dedicated polynomial multiplier. Another option is to equip the processor with a so-called unified multiplier, which is a multiplier that integrates the multiplication of both integers and binary polynomials into the same datapath [24]. The performance gain when MULGF2 is executed in hardware (instead of being emulated software) is tremendous, but this is not surprising since the MULGF2 instruction takes only one or two cycles in hardware, but 190 cycles when emulated in software.

5. EXPERIMENTAL RESULTS

In order to evaluate the performance of the above implementations, we have simulated them with *SimpleScalar*, a trace-driven instruction set simulator [6]. The SimpleScalar tools support different architectures; one of these is PISA (Portable Instruction Set Architecture), which is similar to MIPS32. The main differences between PISA and MIPS32 are that PISA supports additional addressing modes, but has no architectural delay slots. However, we have not used any PISA-specific features which are not available in MIPS32. We configured SimpleScalar to comply with the characteristics of the MIPS32 4Km (i.e., single-issue integer unit with in-order execution). All algorithms have been implemented with a comparable level of optimizations.

Besides the different algorithms for arithmetic operations in prime and binary extension fields, we have also simulated the execution time of a complete point multiplication. As mentioned in Section 2, a point multiplication consists of a sequence of arithmetic operations in the underlying finite field and is the core operation of virtually all elliptic curve cryptosystems. Therefore, we use the point multiplication as a benchmark to compare the efficiency of the arithmetic algorithms.

5.1 Prime Fields

Table 1 summarizes the simulation results for the 192-bit prime field GF(p) with $P = 2^{192} - 2^{64} - 1$ as specified in the NIST standard [22]. All timings are given in clock cycles and have been achieved without loop unrolling (except for the reduction operation). The point multiplication is realized

Operation	Schoolbook	Comba
Field mul. (conv. SW)	629	827
Field mul. (with ISE)	485	441
Point mul. (conv. SW)	$2.16 \cdot 10^6$	$2.84 \cdot 10^6$
Point mul. (with ISE)	$1.67 \cdot 10^6$	$1.47 \cdot 10^6$
Speed-up factor	1.29	1.93

Table 1: Simulation results for a 192-bit prime field.

Operation	Shift+XOR	Word-level
Field mul. (conv. SW)	2758	7848
Field mul. (with ISE)	2151	456
Point mul. (conv. SW)	$4.05 \cdot 10^6$	$10.42 \cdot 10^6$
Point mul. (with ISE)	$3.28 \cdot 10^6$	$0.87 \cdot 10^6$
Speed-up factor	1.23	11.97

Table 2: Simulation results for a 191-bit binary field.

with Jacobian coordinates and a NAF representation of the scalar (a detailed description can be found in [14]). For the given operand length of 192 bits, a point multiplication consists of 1,280 multiplications and 960 squarings in the underlying finite field GF(p). Besides multiplications and squarings, also other field operations like additions, subtractions, halvings, or inversions are carried out during a point multiplication. However, the execution time of the point multiplication is dominated by the field multiplications.

Conventional Software Implementation

Our simulations show that a 192-bit schoolbook multiplication (including reduction modulo P) requires 629 clock cycles on a standard MIPS32 processor with a (32×16)-bit multiplier. Comba's algorithm is approximately 30% slower than the schoolbook method, which is the expected result since the inner loop of Algorithm 1 executes in 11 clock cycles, while the inner loop of Algorithm 2 requires 18 clock cycles for a single iteration (see Section 4.1). A full point multiplication executes in about $2.16 \cdot 10^6$ clock cycles when the field multiplication/squaring is realized according to the schoolbook algorithm. On the other hand, the point multiplication with Comba-like field arithmetic is significantly slower, which reflects the relative performance figures from the field multiplication. Thus, a standard MIPS32 processor favors the schoolbook method over Comba's algorithm.

Impact of Instruction Set Extensions

The relative performance of schoolbook and Comba multiplication changes when both algorithms are accelerated by instruction set extensions. Table 1 shows the impact of the `MADDL` instruction on the execution time of the schoolbook multiplication. The number of cycles for a field multiplication is reduced from 629 to 485. Furthermore, the full scalar multiplication is approximately 29% faster compared to the conventional software implementation.

Thanks to the wide accumulator, a Comba multiplication can be executed in 441 clock cycles (see Table 1), which is almost twice as fast as the implementation with native MIPS32 instructions. Also the full point multiplication is accelerated by roughly the same factor. However, the most important result is that the speed-up factor through instruction set extensions is much higher for the Comba algorithm than for the schoolbook method. When executed on a standard MIPS32 processor, the Comba algorithm is slower than the schoolbook method. Contrarily, when both algorithms are supported by custom instructions, the Comba algorithm outperforms the schoolbook method. This is due to the fact that the two algorithms execute a different number of store instructions in their inner loops.

5.2 Binary Fields

Table 2 shows the simulation result for the binary extension field GF(2^{191}) with $p(t) = 2^{191} + t^9 + 1$ as irreducible polynomial [1]. The point multiplication uses Lopez/Dahab projective coordinates and is realized according Algorithm 3.40 in [14]. For a binary field of degree 191, the point multiplication requires to perform 1,156 field multiplications and 961 field squarings, as well as other operations. The overall execution time of the point multiplication depends mainly on the field multiplication since squaring is relatively cheap in GF(2^m).

Conventional Software Implementation

A field multiplication (including reduction) according to the optimized shift-and-XOR method with a 16-entry look-up table takes 2,758 cycles on a MIPS32 processor. The computation of the table entries consumes some 25% of the overall execution time of a field multiplication. On the other hand, the word-level algorithm using an emulated MULGF2 instruction is significantly slower, which is obvious when considering that a single MULGF2 operation requires 190 clock cycles on MIPS32.

The overall execution time of the point multiplication in GF(2^{191}) is closely tied to the execution times of the field multiplication. Our timings show that the optimized Shift-and-XOR algorithm outperforms the word-level algorithm by a factor of more than three.

Impact of Instruction Set Extensions

Table 2 summarizes the timings for the polynomial arithmetic operations. The custom instruction accelerates the Shift-and-XOR multiplication by about 600 cycles, although only two clock cycles are saved in the inner loop. This is possible since the inner loop is iterated $8s^2 = 288$ times. In relative numbers, the custom instruction allows to achieve a performance gain of roughly 25%. Unfortunately, there are dependencies between the iterations of the inner loops which enforce a sequential execution of the operations. Therefore, it is not possible to further accelerate the Shift-and-XOR method with custom instructions.

The speed-up obtained through hardware support for the MULGF2 operation is also specified in Table 2. Executing the MULGF2 instruction in hardware in one or two cycles accelerates the word-level algorithm by a factor of as much as 12 compared to a standard software implementation.

Another important result is that instruction set extensions change the relative performance of the two algorithms. The conventional software implementation of the word-level algorithm with emulated MULGF2 instruction is much slower than the Shift-and-XOR algorithm. However, the situation is completely different when MULGF2 is executed in hardware. In this case, a field multiplication according to the word-level algorithm is almost five times faster than the Shift-and-XOR method.

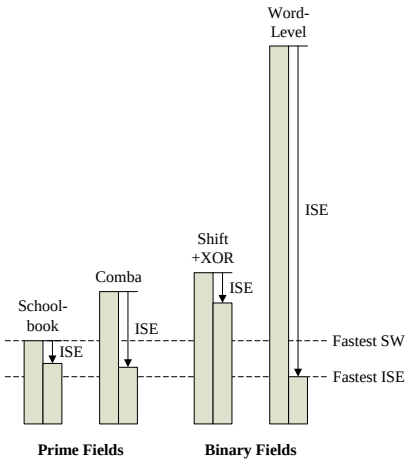

Figure 6: Relative performance of field types.

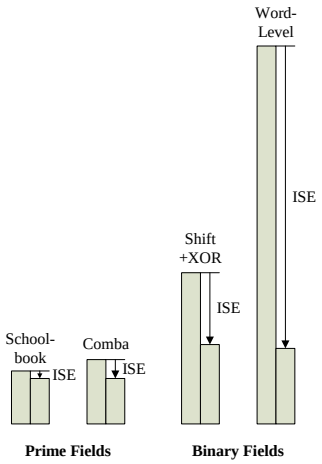

Figure 7: Relative performance with AutoISE.

5.3 Analysis and Discussion

The simulation results given in Table 1 and 2 can be interpreted as follows:

- In the prime field case, a software implementation using schoolbook multiplication is faster than the implementation based on Comba's algorithm. However, when both algorithms are accelerated by custom instructions, Comba's method shows better results than the schoolbook technique.
- The algorithm that is typically used for software implementation of polynomial multiplication (namely the Shift-and-XOR method or one of its optimized variants) can not be significantly accelerated with custom instructions. On the other hand, the word-level algorithm with MULGF2, which is completely inefficient in software, achieves a 12-fold performance gain when the MULGF2 instruction is implemented in hardware, and becomes much faster than the Shift-and-XOR algorithm with custom instructions.
- Instruction set extensions change the relative performance between prime fields and binary extension fields. When implemented in software, prime fields outperform binary fields. The main reason for the good performance of prime fields is the fast (32×16)-bit integer multiplier of the MIPS32 4Km, which facilitates long integer multiplication. On the other hand, when a hardware/software co-design approach with instruction set extensions is employed, binary fields achieve better results than prime fields. The by far slowest configuration in software (binary field with word-level algorithm) becomes the fastest configuration when custom instructions are available. Figure 6 illustrates the relative performance of point multiplication.

6. AUTOMATIC EXPLORATION

The typical design flow for manual selection of ISE is as follows: the designer takes the source code of the fastest possible software implementation as starting point and tries to identify the critical code sections. Thereafter, ISEs are defined and evaluated with the goal to speed up these critical code sections. However, this approach is not viable (i.e., leads to sub-optimal results) for application domains where a number of different but equivalent implementation options exist, e.g., different data structures or different algorithms. Elliptic curve cryptography is a typical example of such an application domain since there exist many different algorithms for one and the same arithmetic operation, e.g., multiplication in a finite field. This calls for a systematic approach to explore the algorithmic design space.

Devising ISE identification algorithms is an active research discipline that attempts to automate the process outlined above. Several algorithms have been presented in the past years [3, 7, 26, 23], and all aim at efficiently and quickly generating the best ISEs for a given application by automatic analysis of the application source code. We have applied the automatic ISE (AutoISE) generation technique proposed by Atasu et al. [3] to the arithmetic algorithms for EC cryptography from Section 3. The AutoISE tool allows selection of any number of ISEs from an application source code, under user-given micro-architectural constraints defining the number of input and output ports that the chosen ISEs are allowed to use. AutoISE uses a simple but effective estimation of the speedup for selecting among millions of identified ISEs. Table 3 summarizes the achieved speed-up factors depending on the number of input/output ports.

The relative performance numbers obtained through the use of AutoISE for an I/O constraint of 3/2 are graphically represented in Figure 7. The key point to observe is that, even when using a very rough estimation model, the results obtained through the automatic ISE tool are in good agreement with those detailed in Section 5; in particular, the word-level algorithm for binary fields is accelerated by a

Arithmetic algorithm	**2/1**	**3/1**	**3/2**	**6/2**
GF(p) Schoolbook	1.11	1.19	1.28	1.34
GF(p) Comba	1.11	1.13	1.21	1.21
GF(2^m) Shift-and-XOR	1.44	1.59	1.77	1.89
GF(2^m) Word-level	2.02	2.17	5.14	5.48

Table 3: Speed-up factors achieved by automatic ISE, for various I/O constraints.

much higher degree than the other algorithms. In a manual ISE design flow, the word-level algorithm for GF(2^m) would likely be ignored due to its poor software performance. On the other hand, automatic ISE tools reduce the risk of overlooking a good candidate algorithm. It can predict trends correctly and guide efficiently and effectively algorithm designers, while screening them from architectural details.

We also point out that the results detailed in Section 5 have required several weeks of work, while the AutoISE tool ran in fractions of seconds for all experiments. The results of Table 3 confirm that automatic ISE tools can help an algorithm designer to explore efficiently and quickly the algorithmic design space.

7. CONCLUSIONS

In this paper we have shown that automatic instruction set extension is not only a tool for boosting the performance of embedded application execution, or for achieving fast exploration of customized architecture solutions. An additional motivation to automate the ISE process is to help *algorithmic design*. Via a study based on elliptic curve cryptography, we have shown that the availability of ISE can have a dramatic impact on the effectiveness of different algorithmic choices to implement identical functionality. We have first made an accurate manual choice of ISEs for different EC implementations, and have carefully measured achieved speedup by simulation, using a detailed model of the ISEs chosen. The study shows for the first time that availability of ISE can reverse the relative interest of different algorithm versions. Furthermore, we have run an automatic ISE tool and demonstrated that, even without predicting speedups as precisely as detailed simulation can, it is able to show exactly and in a matter of seconds the correct trends that the algorithm designer should follow.

8. REFERENCES

[1] American National Standards Institute (ANSI). X9.62-1998, Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA), Jan. 1999.

[2] ARM Limited. ARM SecurCore Solutions. Product brief, available for download at `http://www.arm.com/aboutarm/4XAFLB/$File/SecurCores.pdf`, Feb. 2002.

[3] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pp. 256–261, Anaheim, Calif., June 2003.

[4] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. Cambridge University, 1999.

[5] W. Bond. 64-bit architecture speeds RSA by 4x. MIPS Technologies White Paper, available for download at `http://www.mips.com/content/PressRoom/TechLibrary/WhitePapers/files/RSA.pdf`, 2002.

[6] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, Computer Sciences Department, University of Wisconsin, Madison, WI, USA, 1997.

[7] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customisation. In *Proceedings of the 36th Annual Int. Symposium on Microarchitecture*, San Diego, Calif., Dec. 2003.

[8] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.

[9] J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. Ph.D. Thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.

[10] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual Int. Symposium on Computer Architecture*, pp. 203–13, Vancouver, June 2000.

[11] J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m). In *Proceedings of the 14th IEEE Int. Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468. IEEE Computer Society, 2003.

[12] J. Großschädl and G.-A. Kamendje. Optimized RISC architecture for multiple-precision modular arithmetic. In *Security in Pervasive Computing — SPC 2003*, vol. 2802 of *Lecture Notes in Computer Science*, pp. 253–270. Springer Verlag, 2003.

[13] T. R. Halfhill. ARC Cores encourages "plug-ins". *Microprocessor Report*, 19 June 2000.

[14] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[15] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, Apr. 1998.

[16] K. Lauter. The advantages of elliptic curve cryptography for wireless security. *IEEE Wireless Communications*, 11(1):62–67, Feb. 2004.

[17] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University, 1994.

[18] J. López and R. Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. In *Progress in Cryptology — INDOCRYPT 2000*, vol. 1977 of *Lecture Notes in Computer Science*, pp. 203–212. Springer Verlag, 2000.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC, 1996.

[20] MIPS Technologies, Inc. MIPS32™ Architecture for Programmers. Available for download at `http://www.mips.com/publications/index.html`, Mar. 2001.

[21] MIPS Technologies, Inc. SmartMIPS Architecture Smart Card Extensions. Product brief, available for download at `http://www.mips.com/ProductCatalog/P_SmartMIPSASE/SmartMIPS.pdf`, Feb. 2001.

[22] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). FIPS Publication 186-2, Feb. 2000.

[23] L. Pozzi, K. Atasu, and P. Ienne. Optimal and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005. To appear.

[24] E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 277–292. Springer Verlag, 2000.

[25] J. Turley. Tensilica CPU bends to designers' will. *Microprocessor Report*, 8 Mar. 1999.

[26] P. Yu and T. Mitra. Scalable custom instructions identification for instructionset extensible processors. In *Proceedings of the Int. Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 69–78, Washington, D.C., Sept. 2004.