# Evaluating Instruction Set Extensions for Fast Arithmetic on Binary Finite Fields

A. Murat Fiskiran and Ruby B. Lee
*Department of Electrical Engineering*
*Princeton University*
*{fiskiran, rblee}@princeton.edu*

***Abstract***

*Binary finite fields GF($2^n$) are very commonly used in cryptography, particularly in public-key algorithms such as Elliptic Curve Cryptography (ECC). On word-oriented programmable processors, field elements are generally represented as polynomials with coefficients from {0, 1}. Key arithmetic operations on these polynomials, such as squaring and multiplication, are not supported by integer-oriented processor architectures. Instead, these are implemented in software, causing a very large fraction of the cryptography execution time to be dominated by a few elementary operations. For example, more than 90% of the execution time of 163-bit ECC may be consumed by two simple field operations: squaring and multiplication.*

*A few processor architectures have been proposed recently that include instructions for binary field arithmetic. However, these have only considered processors with small wordsizes and in-order, single-issue execution. The first contribution of this paper is to validate these new arithmetic instructions for processors with wider wordsizes and multiple-issue (e.g. superscalar) execution. We also consider the effects of varying the number of functional units and load/store pipes. We demonstrate that the combination of microarchitecture and new instructions provides speedups up to 22.4× for ECC point multiplication. Second, we show that if a bit-level* reverse *instruction is included in the instruction set, the size of the multiplier can be reduced by half without significant performance degradation. Third, we compare the benefits of superscalar execution with wordsize scaling. The latter has been used in recent processor architectures such as PLX and PAX as a new way to extract parallelism. We show that 2× wordsize scaling provides 70% better performance than 2-way superscalar execution. Finally, we suggest a low-cost method, which we call* multi-word result execution*, to realize some of the benefits of wordsize scaling in existing processors with fixed wordsizes.*

## 1. Introduction

Binary extension fields GF($2^n$), which are commonly used in public-key cryptography, present new datatypes not directly supported by traditional processor architectures with integer functional units. Binary field elements are usually represented in software as polynomials with coefficients from {0, 1}. Key arithmetic operations on these, such as polynomial multiplication, are not supported by integer-oriented architectures commonly used in embedded systems design, like MIPS32 [1] or ARM [2]. Polynomial arithmetic is implemented in software, causing the total execution time of cryptography algorithms to be dominated by a few elementary operations.

A few recent research papers propose instruction set extensions to support binary field arithmetic in embedded processors. The first contribution of this paper is to evaluate the performance benefits of these instruction set extensions in word-oriented programmable

This work was supported in part by Kodak (A. Murat Fiskiran is a Kodak Fellow) and by NSF Research Grants CCR-0105677 and CCR-0326372.

A. Murat Fiskiran and Ruby B. Lee, "Evaluating Instruction Set Extensions for Fast Arithmetic on Binary Finite Fields", *Proc. Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, pp. 125-136, Sept. 2004.

processors. We use 163-bit Elliptic Curve Cryptography (ECC) point multiplication to measure overall performance [3]. We consider multiple-issue execution with varying degrees of superscalar issue width and number of functional units. We also propose including a bit-level *reverse* instruction in the instruction set, which allows the size of the binary-field multiplier to be reduced by half without significant performance degradation. Next, we compare the performance benefits of multiple-issue execution with that of wordsize scaling, and show that the latter provides 70% higher performance. Finally, we suggest a low-cost method, which we call *multi-word result execution*, to realize some of the benefits of wordsize scaling in existing processors with fixed wordsizes.

**Past Work**

Some past work on instruction set extensions for public-key cryptography relate to prime fields GF($p$): [4] and [5] present optimized algorithms and microarchitecture methods on the ARM7 architecture to accelerate multi-precision integer exponentiation. [5] also proposes an *extended shift left* instruction to accelerate the critical loops in RSA. Two custom *multiply-add* instructions are proposed in [6] for a MIPS32 core to accelerate multi-precision multiplication using the Montgomery algorithm.

A significant amount of literature exists on the design of binary-field and dual-field multipliers for embedded cryptographic hardware [7]-[11]. Since most of these designs are either targeted for high-precision applications (greater than 160-bit operands) or depend on the structure of the primitive polynomial of the field (explained in Section 2), they are not suitable for programmable processors where wordsizes are smaller and the primitive polynomial may change from one application to another.

Regarding ISA design, the inclusion of a dedicated functional unit to accelerate binary field arithmetic was initially proposed in [12]. Later, binary-field multiplication instructions were added in [13] to a 16-bit RISC processor core. Finally, the PAX cryptographic processor [14] employed binary-field *multiply* instructions and bit-level *shuffle* instructions for primarily ECC acceleration. However, both [13] and [14] have only considered single-issue execution, while we consider multiple-issue ILP (instruction level parallelism) in this paper. We do not consider the more specialized multiplier designs [7]-[11] mentioned above, but only focus on smaller 32-bit and 64-bit dual-field or binary-field-only multipliers. A dual-field multiplier can be implemented with minor hardware additions to a standard integer multiplier as described in [13], and a binary-field multiplier can be very simply realized as an AND-array followed by an XOR-tree.

The rest of this paper is organized as follows. In Section 2, we review the arithmetic operations and algorithms used in binary finite fields. In Section 3, we evaluate the ISA extensions proposed for fast field arithmetic. In Section 4, we compare the performance benefits of multiple-issue execution with wordsize scaling. In Section 5, we propose multi-word result execution as a low-cost method to implement wordsize scaling. Section 6 is the conclusion.

## 2. Overview of arithmetic operations and algorithms in GF($2^n$)

### 2.1. Arithmetic in GF($2^n$)

The binary field denoted GF($2^n$) contains $2^n$ unique field elements. On word-oriented programmable processors, *polynomial basis* representation of the field elements offers the simplest arithmetic and fastest execution [15]. In polynomial basis, field elements are represented as polynomials with coefficients from {0, 1}. For example, an element $a$ of the 163-bit binary field specified in [16] is a polynomial of maximum degree 162:

$$a = a_{162}x^{162} + a_{161}x^{161} + \ldots + a_1x + a_0 = \sum_{i=0}^{162} a_i x^i \ , \ a_i \in \{0, 1\}$$

This field is generated by the 163-bit irreducible pentanomial $p = x^{163} + x^7 + x^6 + x^3 + 1$. In software, each field element can be represented as a sequence of 163 bits corresponding to the polynomial coefficients. With a wordsize of 32 bits, each $a \in GF(2^{163})$ spans 6 words, $a = (a[5], a[4], \ldots, a[0])$. **Addition** of two field elements *a, b* can then be done by XOR'ing the corresponding pairs of words that contain these coefficients; for example:

```
for i from 5 down to 0 do c[i] := a[i] ⊕ b[i]
```

The **square** of a field element can be simply computed by interleaving the polynomial coefficients with 0's. In our baseline software implementation, we use table lookups to speed this process. For **multiplication**, we use the fastest method among those surveyed by Hankerson et al. in [15], which is the left-to-right comb method. The results of both the squaring and multiplication operations are polynomials of degree maximum 324, which are reduced to standard size (degree < 163) by a **modular reduction** operation, which is equivalent to dividing the result by *p* and taking the remainder. Of the three methods surveyed in [15] for field **inversion**, we use the fastest one, which is based on the Extended Euclidean Algorithm.

We illustrate the relative complexity of these operations in Table 1. Our results and those reported in [15] are obtained (using C) on 450 MHz and 400 MHz Pentium-II (P-II) workstations respectively. The third set of data is obtained using C++ on a 300 MHz UltraSPARC [17]. For all three platforms, the simplest operation is addition, followed by reduction, squaring, multiplication, and inversion.

**Table 1: Execution times for GF($2^n$) field operations and ECC point multiplication**

| Operation | Our results on 450 MHz P-II (C) | | Hankerson et. al on 400 MHz P-II (C) | | Lopez et. al on 300 MHz UltraSPARC (C++)* | |
|---|---|---|---|---|---|---|
| | Time (us) | Cycles | Time (us) | Cycles | Time (us) | Cycles |
| Addition | 0.01 | 5 | 0.10 | 40 | 0.6 | 180 |
| Reduction | 0.15 | 68 | 0.18 | 72 | N/A | N/A |
| Squaring excluding reduction | 0.09 | 41 | N/A | N/A | N/A | N/A |
| Squaring including reduction | 0.25 | 113 | 0.40 | 160 | 2.3 | 690 |
| Multiplication excluding reduction | 2.75 | 1238 | N/A | N/A | N/A | N/A |
| Multiplication including reduction | 2.92 | 1314 | 3.00 | 1200 | 10.5 | 3150 |
| Inversion | 39.58 | 15833 | 30.99 | 12396 | 96.2 | 28860 |
| Point multiplication | 3218 | $1.448 \times 10^6$ | 3240 | $1.296 \times 10^6$ | 13500 | $4.050 \times 10^6$ |

* Timing results from this study are reported in single-decimal precision. It is also unclear whether the reported times for squaring and multiplication include reduction or not. We assumed that they do.

**Table 2: Execution time consumed by point multiplication in ECC algorithms**

| Platform | Operation | Percent of execution time consumed by point multiplication |
|---|---|---|
| 175 MHz DEC Alpha 3000 | 155-bit eDH key exchange | 99.1 % |
| | 155-bit eElGamal encryption | 98.0 % |
| | 155-bit eElGamal decryption | 97.5 % |
| 450 MHz P-II | 163-bit eDSA signature generation | 94.2 % |
| | 163-bit eDSA signature verification | 97.1 % |

**Table 3: Field operations in point multiplication**

| Operation | Per Point Multiplication* | | % of Total Execution Time |
|---|---|---|---|
| | Number of calls | Time (us) | |
| Squaring including reduction | 807.96 | 210 | 6.33 |
| Multiplication including reduction | 975.95 | 2895 | 87.25 |
| Inversion | 1 | 50 | 1.51 |
| Other | N/A | 163 | 4.91 |
| Total = Point multiplication | 1 | 3318 | 100.00 |

* Projective coordinates are used in point multiplication.

| Step | Diffie-Hellman Key Exchange (DH)* | | | Elliptic-Curve Diffie-Hellman Key Exchange (eDH) | | |
|---|---|---|---|---|---|---|
| | Alice | | Bob | Alice | | Bob |
| 1 | Choose random $a \in [2, N-1]$ | | Choose random $b \in [2, N-1]$ | Choose random $a \in [2, N-1]$ | | Choose random $b \in [2, N-1]$ |
| 2 | Compute $T_a = g^a \bmod p$ | | Compute $T_b = g^b \bmod p$ | Compute $T_a = G \times a$ | | Compute $T_b = G \times b$ |
| 3 | Send $T_a$, receive $T_b$ | $T_a \rightarrow$ $\leftarrow T_b$ | Send $T_b$, receive $T_a$ | Send $T_a$, receive $T_b$ | $T_a \rightarrow$ $\leftarrow T_b$ | Send $T_b$, receive $T_a$ |
| 4 | Compute shared key $K = (T_b)^a \bmod p$ $= g^{ab} \bmod p$ | | Compute shared key $K = (T_a)^b \bmod p$ $= g^{ab} \bmod p$ | Compute shared key $K = T_b \times a = G \times ab$ | | Compute shared key $K = T_a \times b = G \times ab$ |

* In DH, $p$ is a large prime; $g$ is a generator of the multiplicative group $Z^*_p$; and $N$ is the order of $g$. Both $p$ and $g$ are known to Alice and Bob prior to the key exchange. In eDH, $G$ is a point on the elliptic curve; $N$ is the order of $G$. The elliptic curve equation and $G$ are known to Alice and Bob prior to the key exchange.

**Figure 1: Integer and ECC variants of Diffie-Hellman key exchange**

### 2.2. ECC operations

Compared to previous generations of public-key algorithms such as Diffie-Hellman, ElGamal, and RSA, Elliptic Curve Cryptography (ECC) offers higher security per key bit, so that smaller keys are sufficient to achieve a desired level of cryptographic resilience [3]. For example, the security of an elliptic-curve algorithm with 160-bit keys is comparable to 1024-bit RSA. Smaller keys also enable faster encryption and require less storage, which is an important factor for very constrained environments such as sensors.

ECC derives its cryptographic strength from the *Elliptic Curve Discrete Logarithm Problem* (ECDLP), which is analogous to the Discrete Logarithm Problem used with the integer multiplicative groups $Z^*_p$ [3]. In ECDLP, a base point that lies on the elliptic curve is multiplied by a scalar $k$. This operation, called *point multiplication*, is realized with a series of field arithmetic operations explained previously. The result of point multiplication is another point on the elliptic curve, $P_{\text{final}} = P_{base} \times k$. While it is easy to compute $P_{\text{final}}$, it is computationally infeasible to recover $k$ when only $P_{\text{final}}$ and $P_{\text{base}}$ are given. By using this one-way property, elliptic-curve variants of integer-based algorithms can be constructed. Figure 1 shows this for Diffie-Hellman key exchange, where the modular exponentiation operation in the integer version (DH) is replaced by point multiplication in the ECC version (eDH) [18]. ECC variants of ElGamal and DSA can be similarly constructed [16].

Table 2 shows what percentage of the execution time of four ECC algorithms is consumed by point multiplication. The figures for eDH and elliptic-curve ElGamal (eElGamal) are from [18] and were obtained on a 175 MHz Alpha workstation. Our results for elliptic-curve Digital Signature Algorithm (eDSA) are for a 450 MHz P-II workstation. Because point multiplication dominates the execution time in every case (> 94%), we can use it as a proxy to measure overall ECC performance [13][15][17]. We use the Montgomery algorithm (with projective coordinates) described in [17] to implement point multiplication, which is the fastest method that does not require significant pre-computations and/or storage. We first use *gprof* to profile the point multiplication operation and examine how it decomposes into the field arithmetic operations (Table 3). On average, squaring takes 6.33% of the total execution time and multiplication 87.25%. The time shown as *other* is the execution overhead, which primarily includes the main control loop (which iterates over the point multiplication function). The time per point multiplication in Table 3 (3318 us) differs from Table 1 (3218 us) because execution with profiling slightly degrades performance.

### 2.3. Baseline simulation results

We use the SimpleScalar toolset [19] to evaluate the benefits of instruction set extensions proposed for binary field arithmetic. To establish baseline results, we first simulate the field

operations and ECC point multiplication on a single-issue processor. Throughout this discussion, we use the notation $n_1/n_2/n_3$ to refer to a processor that has $n_1$ integer ALUs (also equivalent to the issue width), $n_2$ load-store pipes, and $n_3$ multipliers. The single-issue processor is therefore labeled 1/1/1.

**Table 4: Execution cycles on a single-issue (1/1/1) processor**

| Operation | Cycles |
|---|---|
| Squaring | 309 |
| Multiplication | 8722 |
| Point multiplication | 10367502 |

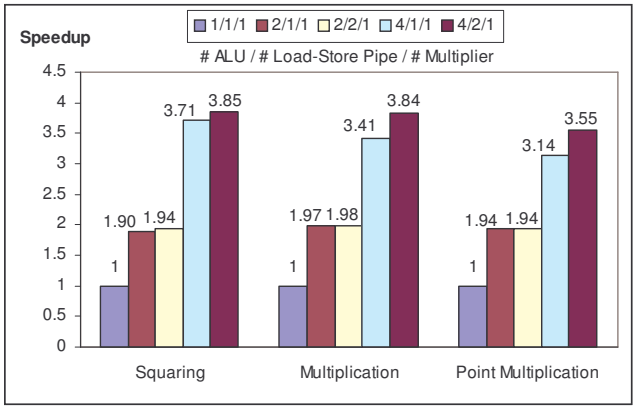


**Figure 2: Speedup at higher issue widths**

The baseline results for the single-issue processor and the subsequent speedups obtained for multiple-issue processors are summarized in Table 4 and Figure 2, where we normalize the single-issue performance for each operation to a speedup of 1.0×. The performance gains from multiple-issue execution are very similar for all three algorithms. In each case, two-way execution provides speedups between 1.90× and 1.97×, while a second memory-pipe has no additional performance benefit. For squaring and multiplication, four-way execution increases the speedups to 3.71× and 3.41× respectively, while the second load-store pipe at this issue width provides small extra benefit (increasing the speedups to 3.85× and 3.84× respectively).

## 3. ISA support for fast binary field arithmetic

### 3.1. PAX instruction set architecture

PAX is a minimalist instruction set architecture (ISA) for high performance cryptographic processing in constrained environments [14]. This includes embedded systems, PDAs, smart phones and secure sensors. The performance of a PAX-based system can be scaled up with microarchitectural techniques such as superscalar execution and wordsize scalability (Section 4). In the rest of Section 3, we study the performance provided by some specific PAX instructions, *shuffle* (Section 3.2) and *bfmul* (Section 3.3), and how much this performance may be further improved with microarchitectural features such as superscalar execution with different numbers of memory pipes and hardware multipliers.

### 3.2. Field squaring using *shuffle* instructions

The first ISA extension we consider is the bit-level *shuffle* instruction included in PAX [14] for fast field squaring. This instruction reads individual bits alternating between two source registers, and writes these to a destination register. The first variant of the instruction, *shuffle.lo*, reads the lower halves of the source registers, while *shuffle.hi* reads the higher halves. *Shuffle*-like instructions for multi-bit subwords have been previously included in multimedia instruction sets IA-64 [20] and PLX [21]. The TI TMS320C64x DSP (C64x) also includes a bit-level *shuffle* instruction, but this can only shuffle the two halves of the same 32-bit source register and has a two-cycle execution latency [22].

*Shuffle* instructions are useful for field squaring because bits (coefficients) of a polynomial can be interleaved with 0's much faster than is possible with table lookups. The first row of Table 5 shows that with *shuffle* instructions, the execution time of the squaring operation has been cut down from 309 cycles to 81 cycles, which is a speedup of 3.81×. In Figure 3, we show the *additional* performance improvement obtained with superscalar execution. Here two-way execution provides a significant speedup of 1.78×, and four-way execution further increases this to 2.60× when one memory pipe is available, and to 2.99× when two memory pipes are used.

**Table 5: Execution cycles and speedup on a single-issue (1/1/1) processor**

| Operation (excluding reduction) | Cycles Per Operation | Speedup over software* |
|---|---|---|
| Squaring with *shuffle* | 81 | 3.81× |
| Multiplication with *bfmul* (writes RH and RL) | 349 | 24.99× |
| Multiplication with *bfmul.lo* + *bfmul.hi* | 351 | 24.85× |
| Multiplication with *bfmul.lo* + *rev* | 488 | 17.87× |

* Compared to the table-lookup method for squaring and left-to-right comb method for multiplication.

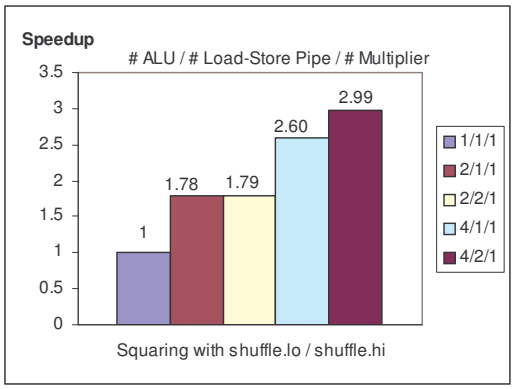


**Figure 3: Speedup of squaring at higher issue widths using *shuffle***

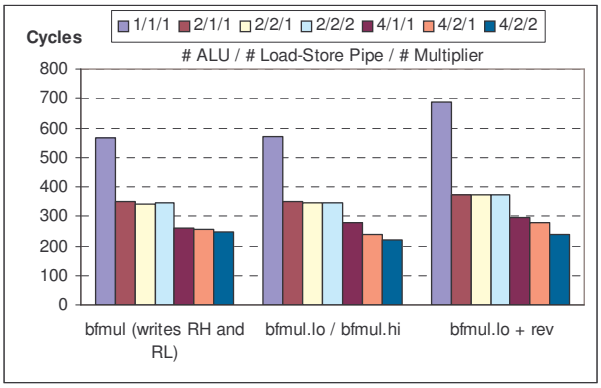


**Figure 4: Execution cycles per field multiplication including reduction**

### 3.3. Field multiplication using *bfmul* instructions and variants

A multiply instruction writes its result to the register file in at least three different ways:

**Case 1:** The higher and lower words of the product are written to two special registers, RH and RL, respectively. The contents of RH and RL can then be moved to general registers using additional instructions. MIPS32 [1] and PISA [19] define multiplication this way. We assume that a binary-field multiply instruction, which we will call *bfmul*, will work similarly.

**Case 2:** There are two separate instructions, *bfmul.lo* and *bfmul.hi*, that write the lower or higher word of the product, respectively, to any general register. PAX [14], PLX [21], the 16-bit RISC core studied in [13], and the ARM7TDMI define multiplication in this way.

**Case 3:** We consider using a bit-level *reverse* (*rev*) instruction that reverses the order of bits in a register, so that the least-significant bit of the source becomes the most-significant bit of the result, and all other bits are also swapped symmetrically. PAX processors [14] and TI C64x DSPs [22] include bit-level *reverse* instructions with 1 and 2-cycle latencies respectively; IA-64 [20] and PLX [21] only have byte-level *reverse* instructions. With a bit-level *reverse* instruction, a processor can use a smaller multiplier that only executes a *bfmul.lo* instruction, and can still generate the higher word of the product. We show this below for a 32-bit multiplier, and it can be shown similarly for larger multipliers. Let $a, b \in \mathrm{GF}(2^{32})$, then:

$$a = a_{31}x^{31} + a_{30}x^{30} + ... + a_1 x + a_0 \quad \text{and} \quad b = b_{31}x^{31} + b_{30}x^{30} + ... + b_1 x + b_0$$

We can split the product $a \times b = ab$ into higher and lower halves, such that the higher half, $ab_H$, contains all terms with degrees greater than 31, and the lower half, $ab_L$, contains all terms with degrees less than 31.

$$ab_H = a_{31}b_{31}x^{62} + (a_{31}b_{30} + a_{30}b_{31})x^{61} + ... + (a_{31}b_1 + ... + a_1 b_{31})x^{32}$$
$$ab_L = (a_{31}b_0 + ... + a_0 b_{31})x^{31} + ... + (a_1 b_0 + a_0 b_1)x + a_0 b_0$$

Now, define a function called *reverse* that performs the same operation as the 32-bit *reverse* instruction. Then:

$$a_{rev} = \text{reverse}(a) = a_0 x^{31} + a_1 x^{30} + ... + a_{31} \qquad \text{and} \qquad b_{rev} = \text{reverse}(b) = b_0 x^{31} + b_1 x^{30} + ... + b_{31}$$

The lower half of the product $a_{rev} \times b_{rev} = a_{rev}b_{rev}$ is:

$$(a_{rev}b_{rev})_L = (a_{31}b_0 + ... + a_0 b_{31})x^{31} + ... + (a_{31}b_{30} + a_{30}b_{31})x + a_{31}b_{31}$$

We now multiply both sides by $x$, and apply the *reverse* function to the lower half of the result:

$$\left[x(a_{rev}b_{rev})_L\right]_L = (a_{31}b_1 + ... + a_1 b_{31})x^{31} + ... + (a_{31}b_{30} + a_{30}b_{31})x^2 + a_{31}b_{31}x$$
$$\left\{\left[x(a_{rev}b_{rev})_L\right]_L\right\}_{rev} = a_{31}b_{31}x^{30} + (a_{31}b_{30} + a_{30}b_{31})x^{29} + (a_{31}b_1 + ... + a_1 b_{31}) = ab_H / x^{32}$$

The left side of the last equation can be written in software as follows, the result of which is equivalent to `bfmul.hi t, a, b`:

```
rev        t1,  a          # t1 and t2 are temporary variables
rev        t2,  b          # two rev instructions can be parallelized
bfmul.lo   t1, t1, t2
slli       t1, t1,    1    # logical shift left by 1 bit
rev        t,  t1
```

Therefore, at the expense of four additional instructions (two of which can be executed in parallel) and two temporary registers, the high word of the product is obtained by using a multiplier half as large.

We now compare the performances for these three cases while assuming single-cycle latencies for the *rev* and *bfmul* instructions. In the PAX processors, the *rev* instruction is executed in the shift unit by adding a 2-to-1 multiplexer to each output line of the barrel shifter core. This is illustrated in Figure 5 for a 4-bit shifter. When the select signal is 0, the multiplexers connect the output of the barrel shifter (the lower inputs of the multiplexers) to the result bus, implementing a normal shift/rotate. To implement a *rev* instruction, no shift is performed on the input and the select signal is set to 1. The multiplexers then connect each result line to the symmetric output line of barrel shifter (the higher inputs of the multiplexers). The wiring complexity in the last stage can be reduced by first rotating the input by half the number of bits in a word when implementing a *rev*. The extra circuitry required for *rev* does not impact the cycle time and has small area cost. Our synthesis results using the TSMC's 90 nm process technology indicate that the increase in area[1] compared to a plain barrel shifter is 6.0% for 32-bit shifters, 5.7% for 64-bit shifters, and 5.4% for 128-bit shifters.

The single-cycle latency assumed for the *bfmul* instructions is also justified because field multiplication has a time complexity approximated by $t_{AND} + \lceil \log_2 n \rceil t_{XOR}$, where $t_{AND}$ and $t_{XOR}$ are the delays for AND and XOR gates respectively. Our synthesis results for 32-bit, 64-bit, and 128-bit input words show that the multiplication delay is similar to that of a carry-save

[1] Number of equivalent minimum-sized 2-input NAND gates is used as a proxy for area.

adder/subtractor, which we assume to be a single-cycle functional unit. Even for the multi-cycle multipliers (this would be the case if a dual-field multiplier was used), the full latency can usually be hidden by instruction scheduling, achieving an effective pipelined latency of 1 cycle.

The data in the last 3 rows of Table 5 shows the execution cycles required for a single field multiplication on the single-issue processor (1/1/1), excluding reduction. The execution times for the first two cases are very similar (349 versus 351 cycles), whereas the third case using the smaller multiplier with *reverse* instructions requires 488 cycles. Even though the *bfmul* instruction can compute and write a full 64-bit product in a single-cycle, its performance is not visibly better than the second case, which requires two separate instructions to generate the same result. This is due to the additional instructions required with the *bfmul* instruction to move the multiplier results from the special registers to the general registers.

Data in Figure 4 compares the execution cycles for field multiplication (including reduction) for multiple-issue processors. While the *bfmul* instruction gives the best results for single-issue execution, the second and third schemes become comparably fast for two-way and four-way execution. This is because: (a) the first scheme cannot utilize the second multiplier unit effectively as both multipliers need to use the same physical target registers (RH and RL), and (b) the latency of the binary-field multiply instruction is a single cycle. Perhaps a surprising result is that while the execution cycles for the third case are the highest for single-issue execution, its performance matches the other two cases at wider issue widths. This is achieved with a smaller multiplier and a low-cost *reverse* instruction.

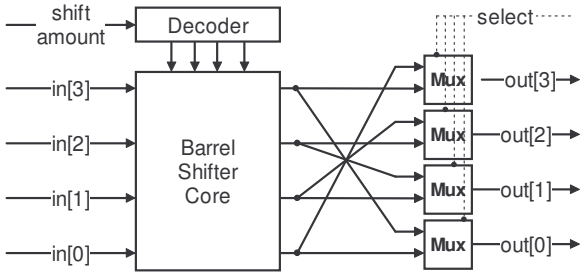


**Figure 5: Implementation of a 4-bit *reverse* instruction**

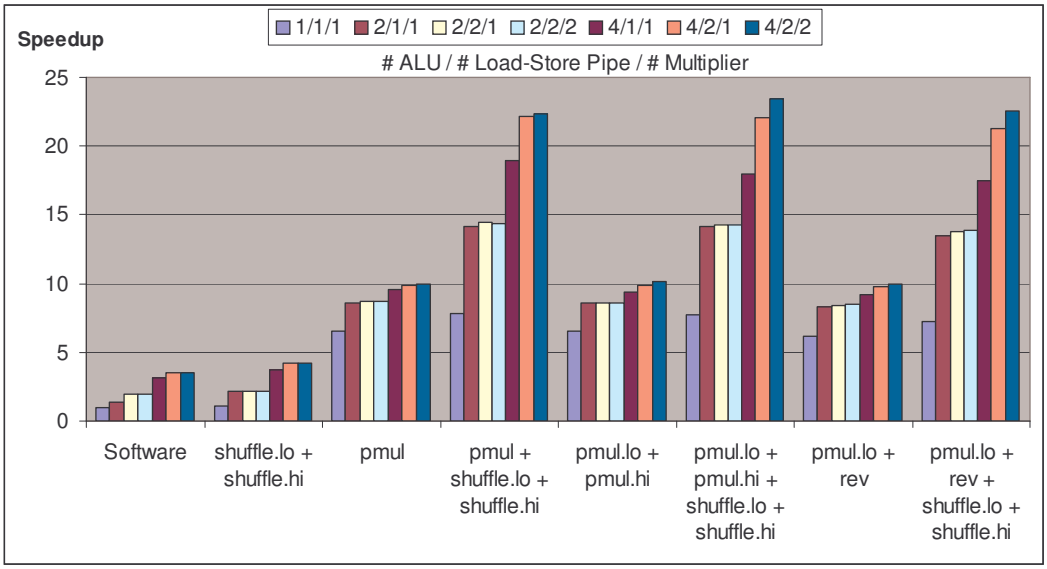


**Figure 6: Speedups for ECC point multiplication from new ISA and superscalar execution**

**Table 6: Speedup due to wordsize scaling in PAX**

| Operation | Single-issue (1/1/1) | | | | | | Two-way (2/2/2) | |
|---|---|---|---|---|---|---|---|---|
| | PAX-32 | | PAX-64 | | PAX-128 | | PAX-32 | |
| | Cycles | Speedup | Cycles | Speedup | Cycles | Speedup | Cycles | Speedup |
| Addition | 6 | 1.00 | 3 | 2.00 | 2 | 3.00 | 3 | 2.00 |
| Reduction | 149 | 1.00 | 106 | 1.41 | 41 | 3.63 | 86 | 1.73 |
| Squaring with *shuffle.lo* + *shuffle.hi* | 28 | 1.00 | 8 | 3.50 | 3 | 9.33 | 15 | 1.87 |
| Multiplication with *bfmul.lo* + *bfmul.hi* | 142 | 1.00 | 36 | 3.94 | 13 | 10.92 | 74 | 1.91 |
| Inversion | 11873 | 1.00 | 7916 | 1.50 | 6156 | 1.93 | 10324 | 1.15 |
| Point multiplication | 534468 | 1.00 | 185579 | 2.88 | 122024 | 4.38 | 316253 | 1.69 |

### 3.4. Results for ECC point multiplication

Figure 6 summarizes the overall speedups for ECC point multiplication obtained with new ISA and superscalar execution. Software implementation for the single-issue processor has been normalized to a speedup of 1.0×. While multiplication with *bfmul.lo+rev* (using the smaller multiplier) gives the lowest overall performance for single-issue execution, the performance improves at higher issue widths and matches the other two multiplication schemes. Overall, using a binary-field multiplier (without *shuffle*) plus superscalar execution gives speedups between 6.5× to 10.1×. At this point, field squaring begins to dominate the execution time since multiplication is accelerated by one order of magnitude. If the *shuffle* instruction is introduced now, the cumulative speedups exceed 22.4× for the four-way superscalar processors (4/2/2).

## 4. Wordsize scaling versus superscalar execution

So far, we have assumed a fixed wordsize of 32 bits, and tried to exploit parallelism via multiple-issue superscalar execution. For the PAX architecture [14], wordsize scalability offers another very effective way to exploit parallelism. First introduced in the PLX multimedia ISA [21], wordsize scalability refers to the feature that the same instruction set can be synthesized to processors with different wordsizes. Both PLX and PAX can be implemented as 32-bit, 64-bit, or 128-bit processors. For PAX, these are denoted PAX-32, PAX-64, and PAX-128 respectively.

To evaluate the performance due to wordsize scalability, we use PAX assembly and the PLX and PAX toolset [23][24] to code the field operations and the ECC point multiplication for PAX-32, PAX-64, and PAX-128. Our results, which are based on single-issue execution, are shown in Table 6. We also show the results for two-way superscalar PAX-32, which can be compared to single-issue PAX-64 since both have equivalent levels of operand parallelism. The results for single-issue PAX-32 are normalized to a speedup of 1.00×. We see that wordsize scaling is far more effective in exploiting parallelism than multiple-issue execution. This is because the running times for the dominant squaring and multiplication operations are $O(m^2)$, where $m$ is the number of words needed to store a single field element (163 bits). This number is reduced by wordsize scaling from PAX-32 ($m = 6$) to PAX-64 ($m = 3$), but not by superscalar execution. The speedups for PAX-64 over PAX-32 are 3.50× for squaring, 3.94× for multiplication (both excluding reduction), and 2.88× for point multiplication. The corresponding speedups for PAX-128 rise to 9.33×, 10.92×, and 4.38× respectively. In contrast, 2-way superscalar execution provides speedups of only 1.87× for squaring, 1.91× for multiplication, and 1.69× for point multiplication.

## 5. Multi-word result (MR) execution

While wordsize scalability is an effective tool for custom cryptographic processors, it cannot be retroactively applied to existing programmable processors, which have fixed ISAs with a fixed wordsize and a fixed number of registers. We now describe *multi-word result* execution,

which allows some of the benefits of wordsize scalability to be realized on existing multiple-issue programmable processors.

We define a *multi-word result* (MR) functional unit as one that generates a result that spans multiple words, and can write these words to multiple target registers in each cycle of execution. In contrast to multiply instructions that can write to only two special registers as in MIPS [1] and PISA [19], MR functional units can write their results to any general register(s).

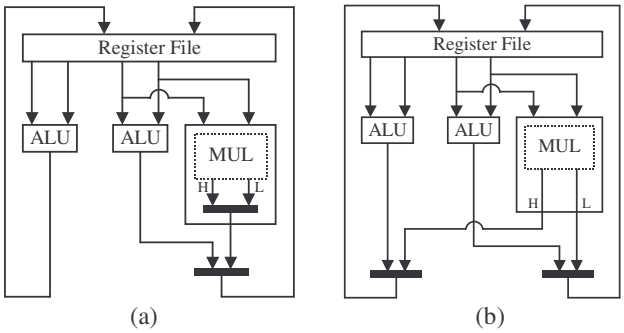


**Figure 7: (a) Standard datapath for 2-way superscalar processor (b) Modified datapath for 2-R multiplier execution**

**Table 7: Speedups from multi-word result execution**

| Operation | (a) Single-issue PAX-32 | (b) 2-way superscalar PAX-32 with one 1-R multiplier | (c) 2-way superscalar PAX-32 with one 2-R multiplier | (d) Wordsize doubling to single-issue PAX-64 |
|---|---|---|---|---|
| Field multiplication using *bfmul.lo* + *bfmul.hi* | 1.00 (142 cycles) | 1.15 | 1.61 | 3.94 |
| Point multiplication | 1.00 (534468 cycles) | 1.32 | 1.90 | 2.88 |

Figure 7 shows the differences between a standard (1-word result, or 1-R) multiplier and a multi-word result (2-R) multiplier, both for a 2-way superscalar processor. The 1-R multiplier executes two instructions, *bfmul.lo* and *bfmul.hi*, to write either the lower or the higher word of the product to the result bus. With the modifications made to the datapath as shown Figure 7(b), a 2-word result (2-R) multiplier is obtained. The full 64-bit product of two 32-bit multiplicands can now be generated with a single instruction. A 2-way superscalar processor with two 1-R multipliers can achieve the same performance as a single 2-R multiplier, but with twice the area for two multipliers. Hence, multi-word result functional units are more cost-effective.

We can simulate 2-R multiplier execution by dynamically monitoring the instruction issue window and looking for consecutive *bfmul.lo/bfmul.hi* pairs using the same source registers. For example:

```
bfmul.lo    Rd1, Rs1, Rs2
bfmul.hi    Rd2, Rs1, Rs2
```

When such instruction pairs are detected, each pair is issued as a single multiply instruction, where Rd1 and Rd2 get the low and high words of the product respectively.

In Table 7 we compare the performance of 2-R multi-word result execution with 2-way superscalar execution and 2× wordsize scaling. All of these three cases have twice the operand parallelism of single-issue PAX-32. We use the results for single-issue PAX-32 as baseline and normalize it to a speedup of 1.00×. For PAX-32, 2-way superscalar execution with one standard (1-R) multiplier gives speedups of 1.15× for field multiplication and 1.32× for point multiplication. When multi-word result execution is

employed with one 2-R multiplier, these speedups increase to 1.61× and 1.90× respectively. While MR execution does not give as much speedup as 2× wordsize scaling, it does improve over the standard 2-way superscalar execution. Moreover, MR execution has the advantage that no ISA changes and only minor microarchitecture changes are required. Therefore, it can be implemented in existing general-purpose processors with a fixed wordsize. In contrast, wordsize scaling requires that a larger 64-bit multiplier is used in PAX-64 (column *d*) versus a 32-bit multiplier in PAX-32 (columns *a*-*c*).

## 6. Conclusions

Binary extension fields GF($2^n$), whose elements are generally represented as binary polynomials in programmable processors, present a new datatype not well-supported by traditional integer-oriented processor architectures. When the key arithmetic operations of this datatype are implemented in software, we find that a very high fraction of the execution time of public-key algorithms like ECC is dominated by a few elementary operations.

In this paper, we first presented a performance evaluation of recent instruction set extensions aimed at accelerating binary field arithmetic. We used multi-way superscalar execution to represent any multiple-issue machine where more than one instruction is issued and executed in a single-cycle. This includes, for example, very long instruction word (VLIW) processors. We found that compared to an optimized software implementation, multiple-issue execution provides 3.55× speedup (4/2/1 processor); inclusion of a dedicated binary-field multiplier provides about 6.5× speedup (1/1/1 processor), and the combined speedup from new ISA (multiplication only) and superscalar execution reaches 10.1× (4/2/2 processor using *bfmul.lo*+*bfmul.hi*). While a dedicated binary-field multiplier allows an impressive 10.1× speedup over software, by including a low-cost bit-level *shuffle* instruction, this speedup can be further increased to 22.4× (Figure 6). This is achieved by speeding up the field squaring operation whose fraction of the execution time increases significantly as multiplication is accelerated by 10×.

Next, we compared the performance benefits of superscalar execution with wordsize scaling. At equivalent levels of operand parallelism (2× wordsize scaling versus 2-way superscalar execution), wordsize scaling provides 70% better performance than superscalar execution. However, wordsize scaling is difficult to apply to existing programmable processors, which have fixed ISAs with fixed wordsize. So, we showed how to realize some of the benefits of full wordsize scaling by multi-word result (MR) execution, which is a low-cost method that requires minimal changes to the datapath.

Our results and findings are applicable to a broad variety of programmable processors. For example, a minimalist cryptographic processor may utilize the ISA extensions we considered and may also use wordsize scaling for additional performance without incurring the complexity costs of multiple-issue processors. An application-specific instruction-set processor (ASIP) designed for higher performance may utilize a combination of new instructions, superscalar execution, and wordsize scaling to achieve a desired performance and cost target. A general-purpose processor may add the discussed ISA extensions to its base instruction set to achieve higher cryptographic performance. General-purpose processors may also use multi-word result (MR) execution to achieve some of the benefits of wordsize scaling with only small microarchitectural changes.

For future work, we will extend our results to binary fields of larger dimensions. We will also create hardware models for the functional units that implement the new instructions proposed. These will be used to generate estimates of latency, area, and power requirements, which will be used for further architectural tradeoff studies. We will also study the applicability of the proposed ISA features on other applications that use binary extension fields or polynomial arithmetic.

## References


[1] MIPS, "MIPS32 Architecture for Programmers Volume 2: The MIPS32 Instruction Set, v2.00", available at <http://www.mips.com>.

[2] ARM, "ARM Instruction Set Quick Reference Card v2.1", available at <http://www.arm.com>.

[3] Kiyomichi Araki, Takakazu Satoh, and Shinji Miura, "Overview of Elliptic Curve Cryptography", *Lecture Notes in Computer Science*, vol. 1431, Springer-Verlag, pp. 29-48, Feb. 1998.

[4] J.F. Dhem, "Design of an Efficient Public-Key Cryptographic Library for RISC-Based Smart Cards", Ph.D. Thesis, Universite Catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.

[5] B.J. Phillips and N. Burgess, "Implementing 1,024-Bit RSA Exponentiation on a 32-Bit Processor Core", *Proc. IEEE. Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, pp. 127–137, Jul. 2000.

[6] J. Großschädl and G.-A. Kamendje, "Optimized RISC Architecture for Multiple-Precision Modular Arithmetic", *Lecture Notes in Computer Science*, vol. 2802, Springer-Verlag, pp. 253-270, Mar. 2003.

[7] E.D. Mastrovito, "VLSI Architectures for Computations in Galois Fields", PhD Thesis, Dept. Electrical Engineering, Linkoping University, 1991.

[8] B. Sunar and C. Koc, "Mastrovito Multiplier for All Trinomials", *IEEE Tran. Computers*, May 1999.

[9] F. Rodriguez-Henriquez and C. Koc, "Parallel Multipliers Based on Special Irreducible Pentanomials", *IEEE Tran. Computers*, 2002.

[10] E. Savas, A.F. Tenca, and C.K. Koc, "A Scalable and Unified Multiplier Architecture for Finite Fields GF($p$) and GF($2^m$)", *Lecture Notes in Computer Science,* vol. 1965, Springer-Verlag, pp. 277-292, Jan. 2000.

[11] L.S. Au and N. Burgess, "Unified Radix-4 Multiplier for GF($p$) and GF($2^n$)", *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, pp. 226-236, Jun. 2003.

[12] E.M. Nahum et al., "Towards High-Performance Cryptographic Software", *Proc. IEEE Workshop Architecture and Implementation of High-Performance Communication Subsystems (HPCS),* pp. 69-72, 1995.

[13] J. Großschädl and G.-A. Kamendje, "Instruction Set Extension for Fast Elliptic Curve Cryptography Over Binary Finite Fields GF($2^m$)", *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, pp. 455-468, Jun. 2003.

[14] A.M. Fiskiran and R.B. Lee, "PAX: A Datapath-Scalable Minimalist Cryptographic Processor for Mobile Environments", to be published in *Embedded Cryptographic Hardware: Design and Security,* Nova Science Publishers, NY, USA.

[15] D. Hankerson, J.L. Hernandez, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields", *Lecture Notes in Computer Science,* vol. 1965, Springer-Verlag, pp. 1-24, Jan. 2000.

[16] NIST, "Digital Signature Standard (DSS) – FIPS Pub. 186-2", Feb. 2000.

[17] J. Lopez and R. Dahab, "Fast Multiplication on Elliptic Curves over GF($2^m$) without Precomputation", *Lecture Notes in Computer Science*, vol. 1717, Springer-Verlag, pp. 316-327, 1999.

[18] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems", *Lecture Notes in Computer Science*, vol. 963, Springer-Verlag, pp. 43-56, Jan. 1995.

[19] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0", *Computer Architecture News*, pp. 13-25, Jun. 1997.

[20] R.B. Lee, A.M. Fiskiran, and A. Bubshait, "Multimedia Instructions in IA-64", *Proc. IEEE Int. Conf. Multimedia and Expo (ICME)*, pp. 281-284, Aug. 2001.

[21] R.B. Lee and A.M. Fiskiran, "PLX: A Fully Subword-Parallel Instruction-Set Architecture for Fast Scalable Multimedia Processing", *Proc. IEEE Int. Conf. Multimedia and Expo (ICME)*, pp. 117-120, Aug. 2002.

[22] Texas Instruments, "TMS320C6000 CPU and Instruction Set Reference Guide", doc. SPRU189F, Oct. 2000, available at <http://www.ti.com>.

[23] R.B. Lee and A.M. Fiskiran, "PLX: An Instruction Set Architecture and Testbed For Multimedia Information Processing", *to be published in the Journal of VLSI Signal Processing*, submitted Apr. 2004.

[24] Princeton Architecture Laboratory for Multimedia and Security (PALMS), PAX Project, <http://palms.ee.princeton.edu/PAX>.