Chapter 9
Instruction Set Extensions for Cryptographic Applications

Sandro Bartolini, Roberto Giorgi, and Enrico Martinelli

9.1 Introduction

Instruction-set extension (ISE) has been widely studied as a means toimprove the performance of microprocessor devices running cryptographic applications. It consists, essentially, in endowing an existing processor with a set of additional instructions that can be useful for speeding up specific cryptographic computations. Recently, researchers became aware of the following: "The efficiency of an implementation algorithm often depends heavily on the details of the target platform, *e.g.*, on the instruction set or the pipeline of a processor. Hence, theoretical complexity measures, such as the bit complexity, can be misleading in practice" ([47]).

In this chapter, we will analyze the implications of designing and deploying an ISE for a microprocessor. We will give details on existing research proposals for various cryptographic applications, highlighting the associated benefits and limitations, and we will show the ISEs that are available in some market products and are proposed in research studies.

9.1.1 Instruction Set Architecture

Instruction-set extension can be better understood only after having a clear idea of what an instruction-set is. At the higher level, an instruction-set (or instruction-set architecture – ISA) can be defined as the pool of instructions made available by a processor to the assembler programmer, or to the compiler.

In this sense, the ISA defines a significant quote of the programming interface of the processor: the basic operations that the outside world can ask the processor to do. The whole programming interface of a processor is surely wider than the sole ISA and, in brief, it encompasses also the structure and features of the processor

Università di Siena, Italy,
e-mail: bartolini,giorgi,enrico@dii.unisi.it

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_9,

registers, the organization of the memory space, as it is perceived by the programmer (*e.g.*, virtual memory, permissions), and the features of the I/O (input/output) space. All this information is needed to take full advantage from the features exposed by a microprocessor.

Anyway, instructions cannot be arbitrarily complex: a trade-off has to be done in order to have fast circuits behind the implementation, and the RISC approach (simple, modular, efficient instructions, *e.g.*, MIPS, SPARC, PowerPC) versus the CISC approach (many powerful instructions, *e.g.* Intel x86) has characterized the main microprocessors on the market.

For instance, we will briefly outline some of the features of the ISA of Intel® processors and its evolution through specific extensions during the years.

The base ISA of the Intel Pentium-4 class processors [31] is almost the same since the old 386-class processor and is named x86 instruction set. It comprises hundreds of instructions, operating on eight 32-bit general-purpose registers. They can be seen also as the sole 16-bit lower part and four of them even allow using their 16-bit lower part as two 8-bit registers (*e.g.*, *AH* and *AL* are 8-bit registers that, together, form the *AX* 16-bit register, which is the lower part of the 32-bit register *EAX*). Specific instructions for integer arithmetic, bitwise operations, movement among registers, and between registers and memory or I/O space can use 8-, 16-, or 32-bit registers. Other instructions manage the program control flow at various levels: jumps, calls and loops, up to interrupt service routine management. This 32-bit ISA has been extended to a 64-bit backward compatible ISA (x86-64) in 2004, after that a similar proposal was done in 2002 by AMD. This extension was motivated by the need to access wide memory regions (*i.e.*, beyond 2–4 GB, according to the available operating system) easily, and to support an increased word-level parallelism which was needed by a number of high-end applications.

Since some versions of the 486 model, the processor was extended to natively support floating-point operations, without the need of an external coprocessor. In this way, the programmer sees some additional configuration registers and eight 80-bit registers for working on floating-point operands. Specific instructions move the operands to/from the floating-point register set and trigger floating-point computations. Specific circuits implement the register file and the operations that are performed upon instruction execution.

Another class of extensions have been proposed, in steps, for vector-like operations which are motivated by the need of supporting efficiently a variety of multimedia applications such as 2-D and 3-D graphics, motion video, image processing, speech recognition, audio synthesis, telephony, and video conferencing. Beginning with the Pentium II and Pentium with Intel MMX technology processor families, a number of incremental extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE, SSE2, SSE3, and SSE4 extensions. Each of these extensions provides a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements contained in specific registers (64-bit MMX or 128-bit XMM registers).

Multimedia extension allows the programmer to see eight 64-bit registers which can be used as groups of eight packed bytes, four packed 16-bit words, or two packed 32-bit doublewords. In this way, specific instructions can operate on such vector-like operands more efficiently (*e.g.*, parallel saturating addition on all packed elements) and other instructions are provided for loading/storing values from/to MMX registers. Architecturally, specific circuits for parallel elaboration of SIMD operations are added to the processor, while *the MMX register file is shared with the floating-point unit*. In this way, no additional storage for MMX registers was needed but, as a drawback, great attention has to be taken when using in the same time both floating-point and MMX instructions.

The SSE extension introduces a separate set of eight 128-bit registers (*XMM*) for SIMD operations, which are intended to support floating-point SIMD operations too. With a dedicated register file, the conflicts with other internal resources, as in the case of the floating-point register file used by MMX, are reduced. In particular, each *XMM* register can be seen by the programmer as four 32-bit single-precision floating-point values. The SIMD operations on such values can help in supporting advanced media and communications applications, for which MMX integer/fixed-point SIMD operations are limiting. Move and conversion instructions too are provided for the interaction of *XMM* registers with memory, MMX and general-purpose registers.

The SSE2 extension increases flexibility in using *XMM* registers as additional floating-point packed values, and introduces the support for packed integers too. In fact, each *XMM* register can be used also as two packed 64-bit double-precision floating-point values, 16 packed bytes, eight packed 16-bit words, four packed 32-bit doublewords, and two packed 64-bit quadwords. Operations on packed integers into SSE registers allow double parallelism than using MMX and avoid conflicts with the floating-point unit. Additional instructions are provided to operate on this variety of operand types.

The SSE3 extension enhances the previous instruction set with only 13 instructions that accelerate some SSE, SSE2, and floating-point capabilities. For instance, an optimized floating-point to integer conversion instruction is provided, as well as an unaligned 128-bit load instruction for integer operands, and additional SIMD operations. This highlights the importance of easying the interaction between the existing processor data formats (*e.g.*, integers and float values) and hardware modules from one side, and the extended circuitry (*e.g.*, registers) and operands (*e.g.*, float or integer packed values) on the other side, in order to boost programmability and performance.

The SSE4 [32] extension was recently proposed with the Intel® Core™ processor family and further improves SSE capabilities. Essentially, it improves the flexibility of SSE in supporting compiler vectorization and significantly increases the available packed doubleword computations.

This brief story of ISA extensions in Intel® processors highlights that the study of the interaction between special hardware resources (*e.g.*, registers, circuits) and the set of available instructions is crucial for accelerating specific computations and thus for the final performance of an ISA extension.

It is very interesting to highlight here the implications of ISA extensions towards an operating system (OS). This occurs mainly when an OS, even minimal, is employed to enable multiprogramming through processor virtualization. In fact, in a multiprogrammed system, the physical processor is *assigned* by the OS to the different processes (*i.e.*, running programs) that are contemporarily in execution so that each one is able to execute, even if in an intermittent fashion. The OS switches the process–processor association according to the time spent by the running process (*e.g.*, round-robin) or when the latter blocks(*e.g.*, wait for an external event). When a process $P1$ leaves the processor to the next process $P2$ (*i.e.*, a context switch happens), the complete *processor state* has to be saved so that it can be restored later when $P1$ will be able to continue its execution exactly as if it was never interrupted at all.

The state of a process comprises, at least, the value of all processor registers, including the state registers (*e.g.*, flags). Therefore, extending the register organization of a processor implies modifications into the OSs in order to properly manage the machine state. If the OS is not updated upon a context switch, it saves only the original processor state and it neglects the additional extended registers. In this way, when the process state will eventually be restored on the processor, a part of its state can be corrupted and the elaboration can become erroneous. This might be taken into account when designing a solution based on ISE for a target processor for which a number of OSs are already present in the market. All of them have to be updated to support certain ISA extensions.

On this point, note that the Intel MMX extension could be supported without OS modifications because they relied on the registers already used by the floating-point unit.

In the following sections, we will analyze various proposals for ISEs for cryptographic application, highlighting, where possible, the motivation for the proposal, the hardware and software features, as well as the resulting performance benefits.

9.2 Applications and Benchmarks

It has become quite clear that a successful cryptographic system needs an understanding of the applications that it will run. Moreover, its performance strongly depends on an efficient implementation of its essential operations [47].

Therefore, possible ISEs also depend on the applications that we currently focus on. At higher level, secure IP (IPSEC), virtual private networks (VPNs), just to give some examples, are further emphasizing the importance of cryptographic processing among all types of communications. Also, mobile (cellular phone) systems, like 3G and beyond, will be using secure methods for payments, digital rights management, handset, and network authentication.

The above scenarios imply that ISEs will have to consider: (i) appropriate benchmarks in order to analyze the real demand for new instructions to be supported; (ii) appropriate platforms (high performance versus embedded systems) for considering the ISE.

9.2.1 Benchmarks

Current research [1, 6, 11, 16, 22, 23, 34, 39, 57] is essentially considering as benchmarks the most-used ciphers for both private-key (3DES, Blowfish, IDEA, Mars, RC4, RC6, AES) and private-key (RSA, DSA, Diffie-Hellman, El-Gamal, and their ECC variants).

Benchmark suites, which also highly condition the research on ISE and the choice of possible platforms, normally include some applications in the security domain. For example, MiBench [24] includes both applications like PGP [60], the famous encryption algorithm developed by Phil Zimmermann (relying on RSA or DSA), ciphers like Blowfish [8], Rijndael/AES [45] and hash calculations like SHA [46] (used in the well-known MD4 and MD5 hashing functions). Another effort to provide a reference benchmark suite for security is the Basicrypt benchmark package [3], which contains standard and elliptic curve code for Diffie-Hellman key exchange, digital signature algorithm (DSA), El-Gamal, and RSA encryption/decryption. Standard algorithms can be used with various key lengths (1024, 2048, and 3072), while for elliptic curve variants parameter files are defined according to fields and curves recommended by NIST standard FIPS 186-2. Public-key benchmarks in Basicrypt package were written using MIRACL C [53] procedures for big integer arithmetic. The MIRACL library consists of over 100 routines that cover all aspects of multiprecision arithmetic and offer procedures for finite-field elliptic curve operations.

Other approaches [23, 34] try to derive the most important operations for the ISE directly from an algorithm implementation.

The two most notable ciphers for private-key cryptography are 3DES and AES as they are selected as US encryption standards. More recently, AES tends to replace 3DES in many applications; we will discuss AES ISEs more in detail in Section 9.3.2.

For public-key cryptography, RSA continues to be a leader in the implementation while ECC-based applications are gaining consensus thanks again to the NIST standardization. RSA is essentially based on modular exponentiation and basic operations are common to other cryptosystems such as ECC. ECC ISEs will be the object of Section 9.3.3.

9.2.2 Potential Performance

Many of the above symmetric ciphers have little parallelism and few bottlenecks. For example, Blowfish, 3DES, IDEA, and RC6 can run within 20% of the performance of a pure dataflow machine [57]. There is more headroom for Mars and Twofish with potential speedups of 29% and 76% respectively. RC4 and AES have much more parallelism and could be sped up with more capable hardware [57]. Similar studies have been performed for ECC in the work of Bartolini et al. [1]. There are potentials for speeding up ECC, especially considering ISE for polynomial multiplication.

9.3 ISE for Cryptographic Applications

In the following sections we will analyze possible ISEs for cryptographic applications. In particular, each section goes into the details of the extensions specifically proposed for information confusion and diffusion, AES symmetric block-cipher and elliptic-curve public-key cryptosystem.

9.3.1 Instructions for Information Confusion and Diffusion

A common operation, especially in symmetric-key algorithms, consists in breaking a message into blocks and using confusion and diffusion to manipulate blocks of plaintext and transform it into ciphertext [50]. The goal of confusion is to obscure the relationship between plaintext and ciphertext by, for example, permuting certain bits. Cypher algorithms, such as Twofish, employ a series of reversible operations to implement a process called diffusion. The goal of diffusion is to impress upon each of the output bits some information from each of the input bits. The diffusion process is also conditioned by the private key, thus constituting an important step for increasing the resistance to ciphertext attackers.

Typical operations to support confusion and diffusion consist in permutations, rotates, substitutions. According to [57], the most common operations of cryptographic kernels such as 3DES, Mars, RC4, Rijndael, Twofish, besides other general operations of typical programs (such as arithmetic, logical, branches, and memory operations) are indeed: permutations, rotates, substitutions.

Rotates are easily reversible by rotating the same distance in the opposite direction and also have good diffusion properties.

The substitution operation can be implemented as a key-based transformation function using a byte-indexed array called an "SBOX", as in the Cryptomaniac processor [6]. Figure 9.1 illustrates the semantic of an SBOX.

Permutation operations rearrange the bits using a parametrized wire network called an "XBOX" in the work of [6]. Permutation is very effective in achieving diffusion [52] and is potentially very powerful and more general than rotates. In fact, an arbitrary permutation can achieve any one of $n!$ outcomes rather than one of n outcomes produced by a rotation.

Arbitrary permutation circuits are usually considered complex and have therefore been avoided in some algorithms such as Rijndael, RC5, RC6, IDEA, Mars, Kasumi. More recently [27], designs that propose to modify an existing shifter in order to perform both shift and also advanced operations to ease the permutations have been presented.

Another operation that has particularly good diffusion properties is modular multiplication [36]. This operation can be easily reversed with modular multiplication of the modular inverse. Also, modular addition has relatively good diffusion properties and it is easily reversed with modular subtraction. Modular operations will be analyzed in more detail in Section 9.3.3

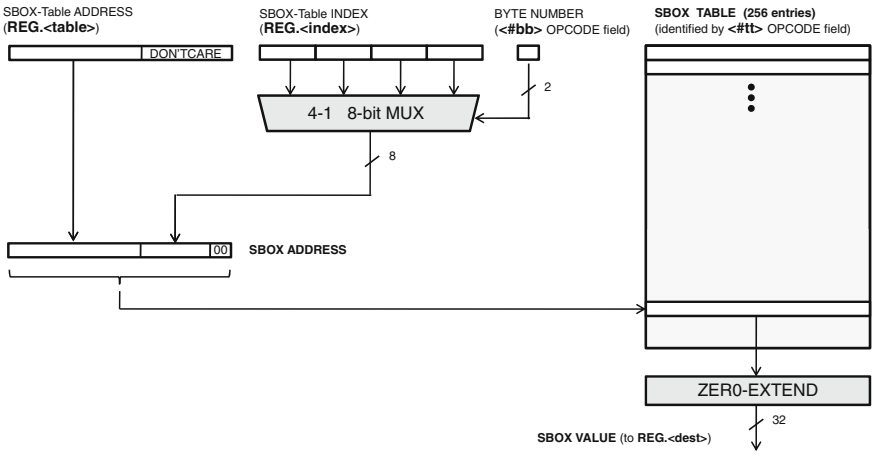

Fig. 9.1 SBOX semantics from Burke et al. [6].

9.3.1.1 Proposed Instructions

Existing ISAs mostly provide support for basic rotate operations, e.g., Intel, Alpha, SPARC, but lack support for permutations and substitutions.

A few ISAs such as PA-RISC [37] and IA-64 [30] support more advanced operations such as extract, deposit and mix, discussed in detail later in this section.

In the work of Burke et al. [6], some potentially useful instructions have been identified, in particular for substitutions (SBOX) and permutations (XBOX). Their methodology is also interesting: starting from commonly used cipher algorithms (such as 3DES, Blowfish, IDEA, Mars, RC4, RC6, Rijndael, Twofish), bottlenecks are identified, and an average of about 45% of dynamic instructions are identified as rotates, substitutions, and permutations.

Several instructions have been proposed in that work [6] and in a subsequent paper describing the Cryptomaniac processor [57], a flexible architecture for secure communications. Some of the proposed instructions are:

- `ROLX <src>, #<rot>, <dest>`
- `RORX <src>, #<rot>, <dest>`
  these two instructions respectively perform a rotate left and rotate right for the specified (`#<rot>`) number of bits of the source register (`<src>`), and a final XOR with another register `<dest>`. The result replaces the second register input `<dest>`. Such instructions are useful to speed up Mars and RC6. The timing analysis performed by the authors of this work indicated that such rotates easily fit in the cycle time of a small-sized ALU (arithmetic logic unit).

- `SBOX.#<tt>.#<bb>.<aliased> <table>,<index>,<dest>`

extracts byte #<bb> (0..3) from register <index>and concatenates the resulting 8-bit value with register <table> to produce a 32-bit aligned address to point an SBOX-table (see Figure 9.1). The 32-bit value from the SBOX-table is zero-extended and loaded into register <dest>. To speed up most SBOX operations, stores to SBOX-table are not visible by later SBOX instructions until an SBOXSYNC instruction is executed, unless the <aliased> flag is indicated. The instruction operates on a table identified by the table designator #<tt> (useful for a subsequent SBOXSYNC instruction). SBOX implementations take advantage of SBOX caches and other quite complex implementation details, making the implementation quite costly. Anyway, SBOX produces great benefits for algorithms such as Rijndael (AES), almost doubling the performance of this algorithm; in particular, having support in hardware for SBOX, reduces the latency for SBOX-table accesses from three instructions to one and speeds up from five cycles to two.

- SBOXSYNC.#<tt>
  synchronizes SBOX-table #<tt> with memory. This eliminates the need for SBOX instructions to snoop on store values in the processor core.

- XBOX.<bbb> <srca>, <srcb>, <dest>
  performs a partial general permutation of register <srca>, given the bit permutation map in register <srcb>; the result of the permutation is placed in register <dest>. The permutation map describes where each input operand bit is written in the destination and contains eight 6-bit indices to address each of the 64-bit <srca> register bits. The XBOX instruction opcode indicates through <bbb>, which of the weight bytes in the destination register are permuted. When XBOX is used, the 32-bit permutations in, e.g., 3DES are completed in 7 instructions (and executed in 3 cycles), yielding a significant improvement over the baseline code which requires 39 instructions.

Performance analysis done in the work of Burke [6] indicated a performance improvement when using the proposed instructions (plus a modular multiplication with modulus 33, and some processor refinements, i.e., doubling execution resources of a baseline aggressive superscalar processor such as Alpha). The improvement can achieve a 59% speedup over machines with basic rotate instructions and 74% speedup over machines without rotates.

Applications that take advantage from the speedup of this algorithm include web servers relying on SSL (or TLS) protocols, disk encryption/decryption, secure network using secure protocols such as IPSEC, and virtual private networks (VPNs).

In emerging applications such as cryptography (but also imaging and biometrics) more advanced bit-manipulation instructions are needed.

In the work of Lee et al. [27, 52], several bit-manipulation instructions are proposed:

- `EXTR r1=r2, <pos>, <len>`
- `EXTR.U r1=r2, <pos>, <len>`
  extracts and right justifies a single field from `r2` of bit length `<len>` from bit position `<pos>`; the high-order bits are filled with the sign bit of the extracted field (`EXTR`) or zero-filled (`EXTR.U`) as in Figure 9.2(a)) and the result is left in `r1`.

- `DEP.Z r1=r2, <pos>, <len>`
- `DEP r1=r2. r3, <pos>, <len>`
  deposits at bit position `<pos>` of single right-justified field from `r2` of bit length `<len>`; remaining bits are zero-filled (`DEP.Z`) or merged from second source register `r3` (`DEP`) as in Figure 9.2(b)).

- `MIX.{r,l}.{0,1,2,3,4,5} r1=r2, r3`
  selects right or left subword from pair of subwords, alternating between source registers r2 and r3; subword sizes are 2^i bits for i=0,1,2, ... ,5 for a 64-bit processor (see Figure 9.3).

A `MIX` operation is implemented in the PA-RISC [38] and IA-64 [30].

Another important class of operations for cryptography is permute [27, 52]. In most current ISAs, permute can be done using logical operations or table lookups (see `XBOX` mentioned earlier). This method may suffer memory latencies and cache misses. A generic n-bit permutation takes $O(n)$ instructions [52]. In the work of Shi et al. [51], the following simple permute instruction – named 'group' – is proposed:

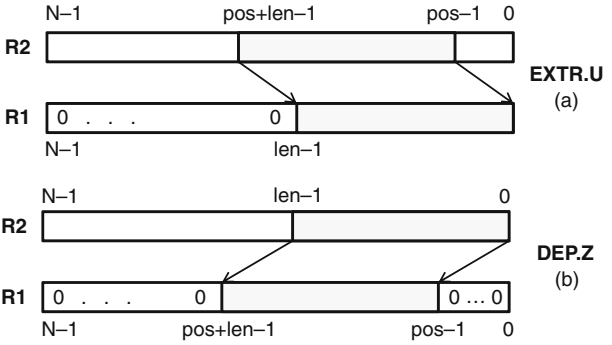

Fig. 9.2 (a) `EXTR.U r1=r2,<pos>,<len>; (b) DEP.Z r1=r2,<pos>,<len>` from Hilewitz et al. [27].

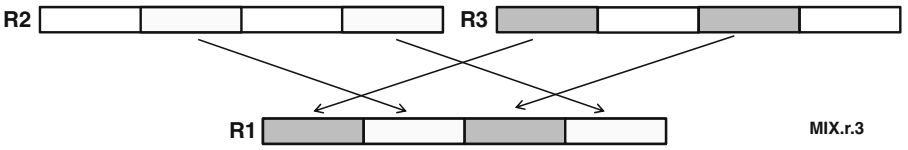

Fig. 9.3 Mix right operation, from Hilewitz et al. [27].

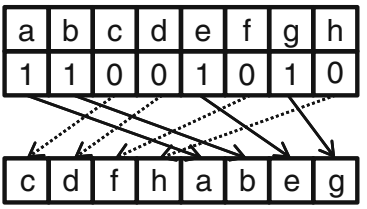

Fig. 9.4 An 8-bit group operation from Shi et al. [51].

- `GRP rd, rs, rc`
  permutes its data input `rs` into its data output `rd`, by grouping to the right those bits flagged with a 1 in a configuration input `rc`, and grouping to the left those bits flagged with a 0 (see Figure 9.4). A series of $log(n)$ `GRP` instructions can generate any permutation.

The `CROSS` [59] and `OMFLIP` [58] instructions use a virtual Beneš network or omega-flip network, respectively, to permute the n data bits. A n-stage Baneš network for permuting n bits is a butterfly network followed by an inverse butterfly network, each of which has $log(n)$ stages (Figure 9.5). An omega-flip network is isomorphic to a Beneš network. Since each `CROSS` or `OMFLIP` instruction executes the equivalent of two stages of the network, both can achieve any of the $n!$ permutations in at most $log(n)$ instructions.

- `CROSS.m1.m2 rs, rc, rd`
  executes the operation of the two butterfly stages specified by m1 and m2 that are characterized by a "butterfly width" of 2^{m1} and 2^{m2} respectively. The bits to be permuted are in `rs` and the permuted bits are in `rd`, while `rc` holds two groups of configuration bits (the left $n/2$ configuration bits in `rc` are for $m1$ and the right $n/2$ configuration bits in `rc` are for $m2$. There are $n/2$ butterflies in each n-bit stage; each configuration bit in a group specifies if the butterfly will propagate data into the straight path (0) or into the cross path (1) of each butterfly. By chaining `CROSS` instructions in sequence, a Baneš network for a desired permutation can be configured.

- `OMFLIP.c rd, rs, rc`
  The 2-bit subopcode `c` indicates which two basic operations are used in this instruction. For each bit, 0 indicates that an omega operation is used and 1 indicates that a flip operation is used. There are four combinations of `c`: omega-omega, omega-flip, flip-omega and flip-flip. The first basic operation takes the source register `rs` and moves the bits in it based on the least significant half of the configuration register `rc` to an intermediate result. The second basic operation moves the bits in the intermediate result according to the most significant half of `rc` to the destination register `rd`.

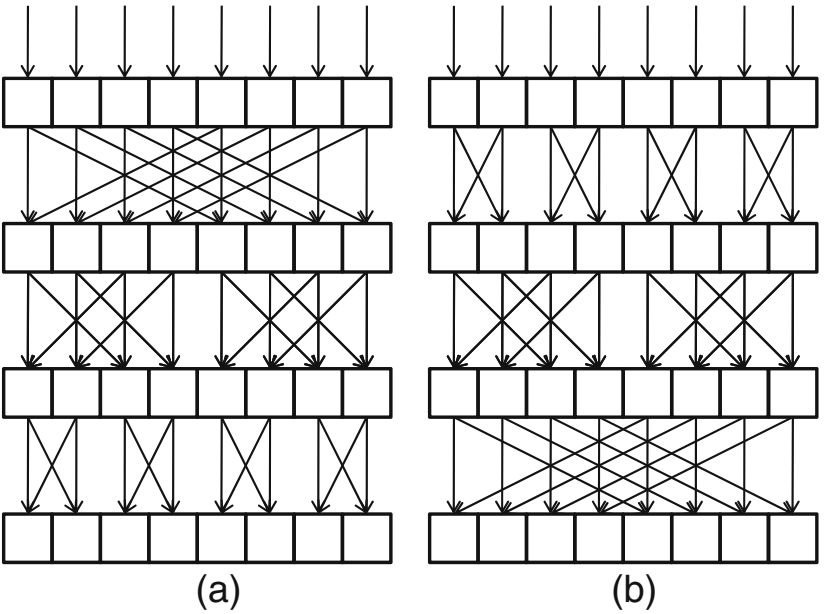

Fig. 9.5 (a) 8-input butterfly network; (b) 8-input inverse butterfly network from Shi et al. [52].

When comparing `OMFLIP` instructions versus table lookup, Yang et al. [58] achieved a 1.33 speedup for DES encryption/decryption and 16.55 speedup on DES key scheduling for a 2-way superscalar architecture with 1 load-store unit and a cache system similar to Pentium III processor. An omega-flip network can achieve the same performance as a `CROSS` instruction with a far smaller hardware implementation, according to [39].

Other proposed instructions are `PPERM` [39], `SWPERM` with `SIEVE`, `BFLY`, and `IBFLY`. Some fast implementations of these permutation instructions (including `CROSS`, `OMFLIP`, `GRP`) is proposed in the work of Hilewitz et al. [28]. The `GRP` can also be used to perform hardware radix sorting and has strong inherent differential cryptographic properties, but the implementation is relatively slow. On the other hand, the `BFLY/IBFLY` have a relatively fast implementation, as they can perform their operation in a single cycle.

The `PPERM` instruction [39] represents an intuitive way to do permutations (Figure 9.6), where the position of each bit in the destination `rd` is specified by a bit configuration register `rc`, `rs` being the source. This is similar to the `PERMUTE` instruction in the MAX-2 ISA extension [38]:

- `PPERM.x rs, rc, rd`
  permutes its input `rs` according to an explicit list of indices that are packed (in k groups) into `rc`; $x = 0, 1, \ldots, 7$ is a byte offset that specifies which k contiguous bits in `rd` will receive the source bits given by `rs`, while the remaining bits in

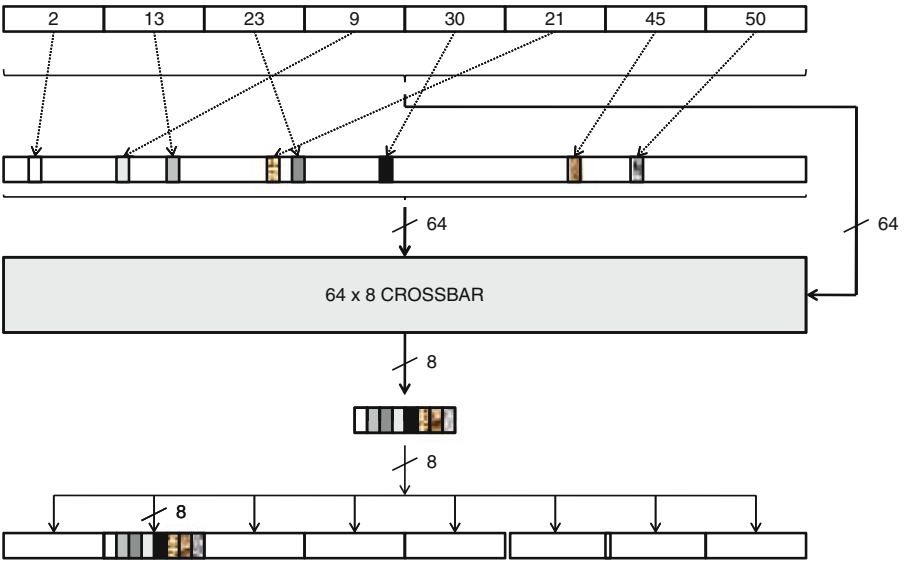

Fig. 9.6 Diagram flow of bits for `PPERM.1 R1,R2,R3`; the number in `R2` are the bit positions in `R1` [39].

`rd` are zeroed. Each index in the list `rc` specifies where in `rs` to extract each of the k bits (being $k <= n/log(n)$ for n-bit information, at most $n/log(n)$ indices are available in each instruction). Therefore, to specify a generic permutation, at most log(n) instructions are needed.

In a similar way the `SWPERM` with `SIEVE` [42] selects a source bit by its numeric index.

In the work of Shi et al. [52], the `BFLY/IBFLY` instructions are proposed. These can be useful for example in P-box permutation of DES cipher round function. By cascading these two instructions, a generic n-bit permutation can be achieved. While the implementation is fast, an ISE issue could be how to supply the configuration bits for the butterfly network. For $n = 64$, a 6-stage butterfly network requires $3n$ configuration bits, that is $n/2$ bits for each stage.

- `BFLY rd, rs, ar.b1, ar.b2, ar.b3`
- `IBFLY rd, rs, ar.b1, ar.b2, ar.b3`
  permute their input `rs` (and leave the results in `rd`)using a butterfly and inverse butterfly circuit, respectively (Figure 9.5). Assuming a 6-stage network, three extra special registers are needed (`ar.b1, ar.b2, ar.b3`, which contain the configuration bits). The configuration bits have similar meaning as in the `CROSS` instruction discussed above.

9.3.2 ISE for AES

Private-key (symmetric-key) cryptosystems are typically devoted to the encryption/decryption of the bulk of communications between two parties, while public-key cryptosystems are usually employed for authentication, exchange of the session key of symmetric algorithms, digital signature and other security operations involving a limited amount of information. One of the reasons for this is that public-key approaches are far more computationally expensive than private-key counterparts. However, as the bulk data encryption/decryption is performed by private-key algorithms, it is very important to implement these techniques very efficiently.

Recent research proposals analyzed ISEs, as well as specific coprocessors for speeding up these cryptosystems. AES [9] and the older DES [13] and triple-DES, or TDEA [14], block ciphers are and have been widely used for symmetric-key cryptography. The Rijndael block cipher [9] was accepted in 2000 as the new advanced encryption standard (AES) by the U.S. government and standardized by FIPS in 2001 [15]. AES is adopted in an increasing quote of applications and systems because of its security properties, flexibility and good implementability features on a wide range of architectures (*e.g.*, from 8-bit processors and up) both in hardware and in software. For these reasons we will focus mainly on AES in this section.

AES is a block cipher that operates on 128-bit blocks and can support 128, 192 and 256-bit keys. The 128-bit block is seen as a 4x4 matrix of bytes (*state*) and the encryption/decryption operations work on such a matrix, performing permutations on the rows, on the columns, and substituting bytes in it according to modular arithmetic with polynomials. The particular set of elaborations is briefly summarized in Figure 9.7 as pseudocode. Note that some operations (*rounds*) are performed a number of times (*e.g.*, 10 in the case of 128-bit key).

Initially, the *state* is the input block and then, it represents the intermediate results of the rounds. At the end, the *state* represents the encrypted block.

The SubBytes operation updates each byte in the array using an 8-bit S-Box, which introduces the non-linearity in the algorithm. The S-Box is derived from the multiplicative inverse over GF(2^8) and can be calculated directly or through a look-up table.

The ShiftRows operation rotates the rows of the state cyclically to the left. The first row is not shifted, the second row is shifted by 1 byte, the third by 2 bytes and the last row by 3 bytes.

In the MixCol operation, the four bytes of each column of the state are combined as follows. Each element of a column is considered as a polynomial over GF(2^8) and multiplied modulo x^4+1 with a fixed polynomial $c(x) = 3x^3+x^2+x+2$.

The AddRoundKey operation mixes a key, which is derived at each round from the main key, to the state.

The encryption and decryption steps are very similar and are based on the same set of base operations. Full details on AES internals can be found in the reference literature [9, 15].

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

```
AES_encrypt(in, out, w)
byte in[4*4],
byte out[4*4],
word w[4*Nr+1]
{
    byte state[4,4];
    state = in;

    // Initial AddRoundKey
    AddRoundKey( state, w[0, 3] );

    for round = 1 step 1 to Nr-1  // (Nr-1) rounds
    {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, w[round*4, (round+1)*3]);
    }

    // Last round (no MixColumns)
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, w[Nr*4, (Nr+1)*3]);

    out = state;
}
```

Fig. 9.7 Implementation of the inner loop of schoolbook multiplication in GF(p).

The AES was thought of for easy implementation and execution efficiency on both 32-bit microprocessors, as well as on small 8-bit microcontrollers. However, each round of the algorithm requires a significant number of assembler instructions because of the specific features of the involved operations. This fact limits the encryption/decryption bandwidth achievable in software. Highly optimized software approaches tend to employ look-up tables, which are acceptable for general-purpose systems but can be inadequate for memory-constrained embedded devices. However, when high speed is required, software implementations are too inefficient and ad hoc hardware approaches (*e.g.*, cryptoprocessors) are preferable. However, a crypto-processor might lack the flexibility to accommodate to different algorithm parameters (*e.g.*, key size) and, typically, is not able to run general-purpose code.

Using specifically extended instruction-sets and functional unit extensions on existing processors, significant performance gains are achievable and complete compatibility with the unmodified general-purpose processor can be maintained, keeping low the design complexity.

9.3.2.1 Extended Instructions

Bertoni et al. in [2] propose a word-level ISE for an ARM processor. Given its generality, the approach could be applicable to almost all 32-bit processors. As a baseline, they focus on a software implementation which relies on three look-up tables: two for the values of the nonlinear transformation and its inverse, and one for storing constants for the key schedule. Based on the execution time profiling of this software version, the most time-consuming code parts have been moved into hardware and specific instructions have been added to trigger the new hardware units.

They propose to include a couple of instructions: SBox and SMix which perform S-Box and both S-Box and MixColumns transformations in a single step, respectively. First, they propose a byte-oriented approach, where the new hardware and the new instructions work on one byte at a time. Then, they extend their proposal to 32-bit words, modifying both the hardware and the instructions and exploiting in this way the parallelism of the architecture and of the algorithm.

The byte-oriented instructions operate on only one byte of the AES status at a time, as shown in Figure 9.8. As the processor registers are word-oriented, a *Selector* block selects the right byte from the register word. After the byte is extracted, the nonlinear transformation (bytewise S-Box) is applied.

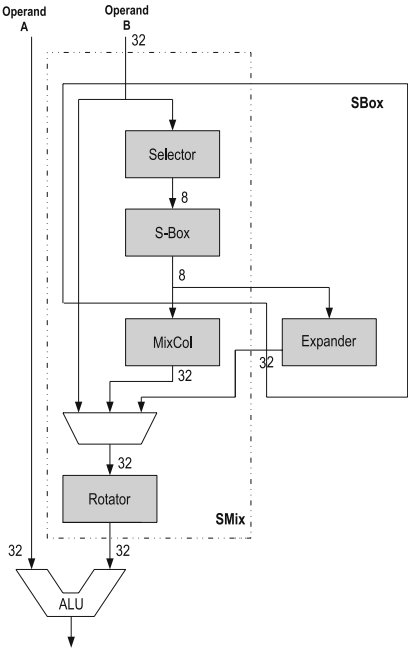

Fig. 9.8 Circuit structure to support *SMix* and *SBox* instructions.

The MixCol module processes the byte and completes the SMix instruction: it outputs four bytes, representing the different contributes. At every step of the AES, the polynomial changes but the multiplicative coefficients remain the same. Therefore, the correct result can be obtained through a reordering step of the bytes. The *Expander* on the right in Figure 9.8 is responsible to expand the processed byte to a word. If X is the input byte, 000X is the output word of the expander. In this way, its output can be sent to the final *Rotator* which is able to position the nonzero byte in the needed position of the result through an XOR. In this way, the four bytes of an output word (*i.e.*, 000A, 00B0, 0C00, D000) can be produced independently and XOR-ed together to form the output word (*i.e.*, DCBA). The suggested byte instructions are:

- **SBox Rs, Rd, Index**, which performs the nonlinear substitution on a byte. Four of them are needed to process an entire 32-bit word. *Rs* indicates the source register, *Rd* represents the destination register, and *Index* indicates the byte to extract and configures the *Rotator* operation. First, the accumulator must be initialized to "0000" and a load instruction is needed to initialize the source register. This instruction is used only in the key-scheduling phase and in the last round, where the MixColumns transformation must be skipped.
- **SMix Rs, Rd, Index**, which performs both the Sbox and the MixColumns transformations on the selected byte. As in the previous case, *Rs* indicates the source register , *Rd* represents the destination register and *Index* selects the correct byte. Here, the *Expander* is not needed since the output from the MixColumns module is 32-bit wide. This module can be used to produce all the needed contributes since the rotator is responsible for their reordering.

The assembler code to perform an AES round using the presented instruction extensions is shown in Figure 9.9.

Moving to a word-oriented approach, the MixColumns transformation can be easily extended for producing the new columns, while the ShiftRows transformation is not straightforward since it works in the orthogonal direction of the state matrix than the MixColumns one. The authors propose to solve the problem by modifying four fixed registers of the CPU in order to allow access to a single byte in each of them in parallel, thus actually retrieving the word exactly as if it was passed through the ShiftRows transformation. From the implementation point of view, this implies modifying the register file of the processor and can be more or less difficult, depending on the internal architecture of the processor.

For the SubWord (S-Box) transformation, a word-level S-Box module is used. As in the byte-oriented solution, the MixCol module is cascaded to it. The rotator is still needed for the key-scheduling process. In this step, a word of the key contribute must be rotated and processed by the S-Box module. The final result is obtained through an *XOR* operation in the ALU.

The three word-level instructions needed are:

- **SMixW N, Rd**, which performs the S-Box and the MixColumns transformation in sequence. It does not need source registers as operand, since the registers containing the state are fixed due to the changes performed in the register file to

```
# Round computation                    # Key unroll
 1: SMix R0, R0, 0   ▷ 1st column      1: SBox R10, R4, 3   ▷ word rot. + S-box
 2: SMix R0, R0, 1                     2: SBox R10, R4, 0
 3: SMix R0, R0, 2                     3: SBox R10, R4, 1
 4: SMix R0, R0, 3                     4: SBox R10, R4, 2
 5: Xor R0, R4                         5: XOR R10, R8       ▷ Add round const
 6: SMix R1, R1, 0   ▷ 2nd column      6: XOR R4, R10       ▷ Calculate words of next
 7: SMix R1, R1, 1                        contribute
 8: SMix R1, R1, 2                     7: XOR R5, R4
 9: SMix R1, R1, 3                     8: XOR R6, R5
10: XOR R1, R5                         9: XOR R7, R6
11: SMix R2, R2, 0   ▷ 3rd column
12: SMix R2, R2, 1
13: SMix R2, R2, 2
14: SMix R2, R2, 3
15: XOR R2, R6
16: SMix R3, R3, 0   ▷ 4th column
17: SMix R3, R3, 1
18: SMix R3, R3, 2
19: SMix R3, R3, 3
20: XOR R3, R7
```

Fig. 9.9 Implementation of the inner loop of schoolbook multiplication in GF(p).

allow the selection. The arguments are the column index N and the destination register *Rd*.

- **SubWord Rs**, which performs only the S-Box transformation of the incoming word. It is used in the last round and in the key scheduling.
- **KSFW Rs, RCon**, which causes the word contained in the register *Rs* to feed the SubWord module; after that, it is rotated by one position and summed with the *RCon* constant stored in *RCon*, producing a word of the new key contribute. This represents the transformation of the first word of the key contribute.

The assembler code resulting from availability of these word-level instructions is shown in Figure 9.10.

Using the proposed ISE, the performance speedup in AES encryption is about 2.54x for AES-128. In particular, 497 cycles are needed using all the extended word-level instructions against 727 cycles when using the byte-level ones and 1771 cycles for the original software version. The speedup is very interesting, especially because it is obtained on top of a high-performance software implementation employing various look-up tables.

```
# Round computation        # Key unroll
1: SMixW 0, R10, R4        1: KSFW R4, R8
2: SMixW 1, R11, R5        2: XOR R5, R4
3: SMixW 2, R12, R6        3: XOR R6, R5
4: SMixW 3, R13, R7        4: XOR R7, R6
```

Fig. 9.10 Implementation of the inner loop of schoolbook multiplication in GF(p).

Another approach is proposed by Elbirt [12], who presents an ISE for Galois Field (GF) constant multiplication within a 32-bit SPARC v8 compatible processor. The approach can be applied to all the algorithms that use GF constant multiplication, not only to the AES. As the a central computation of AES rounds is the modular multiplication of GF(2^8) elements, the core of the proposal is to adopt an 8x8 *matrix* circuit that operates on the 8 bits of the input polynomial $a(x)$ and generate the 8 bits of the multiplication of $a(x)$ for the constant polynomial $k(x)$. The circuit implements a vector–matrix multiplication, where the 8 bits of $a(x)$ are the vector and the 64 coefficients of the matrix comprise both the bits of the constant $k(x)$ and the operations for executing also the modular reduction step. The total number of gates for such a circuit is about 7200, which is shown to be 18 times smaller than a look-up table approach.

The cycle count needed to perform an 8-bit modular multiplication comprising reduction is 1 cycle for the matrix approach, while, for instance, it is 28 cycles in software for a Pentium-class Intel processor. The authors show that the speedup over the software version without extension is 1.67x, and further improvements (up to more than 8x) are achievable if other extensions proposed in literature, like *S-Box*, are used together with the *matrix* extension.

Fiskiran and Lee in [17] analyze a variety of cryptographic algorithms and techniques for mobile devices. They consider symmetric-key, hash, public-key, ECC, and DSAs and analyze the main features of their software implementations in order to single out possible extensions to accelerate their execution. For the considered symmetric-key algorithms (AES, DES/3DES, RC4, Blowfish, MARS, Twofish), the workload characterization gives the following indications. Six of the 10 ciphers use table lookups, and 5 of these spend the largest fraction of their execution time during these table lookups (*e.g.*, 72% for AES, 58% for RC4). For all ciphers, tables are small and constant in size. The number of entries per table is 256 for 5 of the ciphers (8 index bits), and the data read is either 8 or 32 bits. Apart from RC4, tables are accessed only for reading. Moreover, the typical round structure of the ciphers allows also parallel table lookups. For example, all lookups in an AES round (16 lookups) can be performed in parallel, given enough hardware resources.

Based on this analysis, the authors propose an ISE for supporting table lookups in symmetric-key ciphers. The reference platform used is a PAX architecture, a minimalist high-performance cryptographic processor ISA designed at Princeton University [49], which features an ALU, a shift unit, a multiplier, and 8 on-chip tables (T0–T7). Each table has 256 entries and the size of each entry is equal to the processor word size, which may be 32, 64 or 128 bits.

The proposed lookup instruction for 32-bit word size is able to perform up to 4 simultaneous lookups and is named *ptlu* which is short for parallel table lookup. The exact format of the instruction is *ptlu.subword.table.offset.step Rd, Rs*, where the *subword* field selects the number of bytes to read from the table (size of data to read). The *table* field is 3 bits wide and selects the desired table. The byte-sized indices used to access the tables are read from the source register *Rs*, which is 32-bit

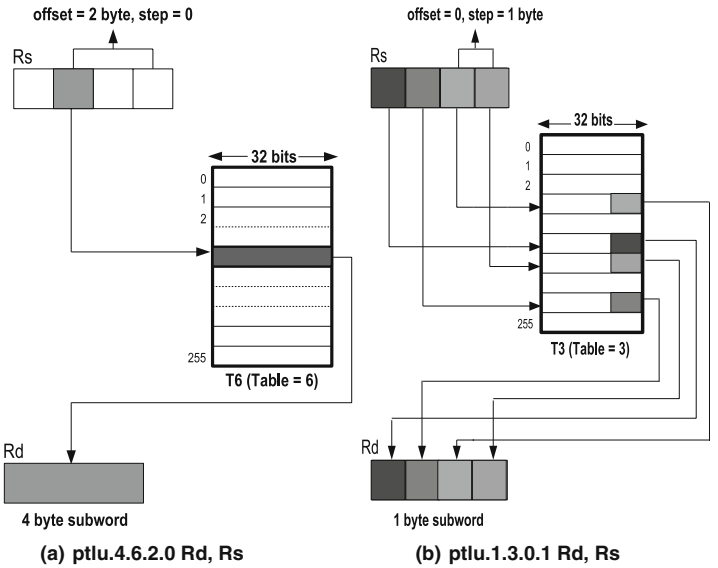

Fig. 9.11 A couple of examples of the *ptlu* instruction for accelerating table lookups.

long and so it can contain up to four byte-sized indices. The 4-bit *offset* field is used to select the first byte-sized index in *Rs*. The 4-bit *step* field gives the distance (in bytes) between two consecutive bytes in *Rs* that are used as indices when multiple lookups are performed.

For instance, in Figure 9.11 a couple of examples are shown. In 9.11a, a 4-byte access is shown using the byte 2 of *Rs* as index of T6, while in 9.11b, four parallel lookups in table T1 are made to read 1 byte per lookup and using the four bytes of *Rs* as indices. Note that the AES key expansion can take full advantage from this instruction to perform the byte substitution procedure.

The *ptlu* instruction is assumed to have 1 cycle latency, as it accesses a small on-chip memory. Using *ptlu*, the speedup of AES is 2.29x. Table 9.1 also shows that the other considered algorithms benefit from *ptlu* and deliver a significant speedup.

Table 9.1 Algorithm speedup using ptlu.

Algorithm	% Speedup
AES	2.29x
DES	1.28x
3DES	1.25x
RC4	1.92x
Blowfish	1.73x
MARS	1.40x
Twofish	1.61x

The authors highlight a possible advantage of *ptlu* instruction: its scalability to processors with different word sizes. In fact, as word size increases, more lookups can be performed simultaneously. For instance, if the word size is 128-bit, only 4 *ptlu* are needed to complete the 16 lookups of a single AES round. In case of 64 and 128-bit word sizes, the speedup for AES reaches 2.85x and 6.10x, respectively.

Tillich *et al.* in [55] analyze possible ISEs for AES on a 32-bit processor. Similar to [2], they propose byte-oriented and word-oriented instructions but with some differences. They propose byte-level *Sbox* and *MixCol* instructions, which calculate only one byte of the result. A specific immediate value of the instructions allows one to choose the source byte (*Sbox*), the destination byte and the encryption/decryption operation (for both *Sbox* and *MixCol*).

The *Sbox4* and *MixCol4* instructions extend this approach to 32-bit words and aim to parallelize four byte-operations. As shown in Figure 9.12, the *Sbox4* instruction substitutes all four bytes of the first source register and places them into the destination register. An optional byte-wise rotation can be performed on the result. The immediate value selects whether S-Box or inverse S-Box are used and specifies the rotation distance for the result. In this way, the key expansion step can be efficiently supported through this *sbox4* instruction. The *mixcol4* instruction calculates all four result bytes of the MixColumns or InvMixColumns operations, according to the immediate value.

The performance of the word-oriented ISE is better than the byte-oriented one but not as much as it could be expected due to the four-fold parallelism

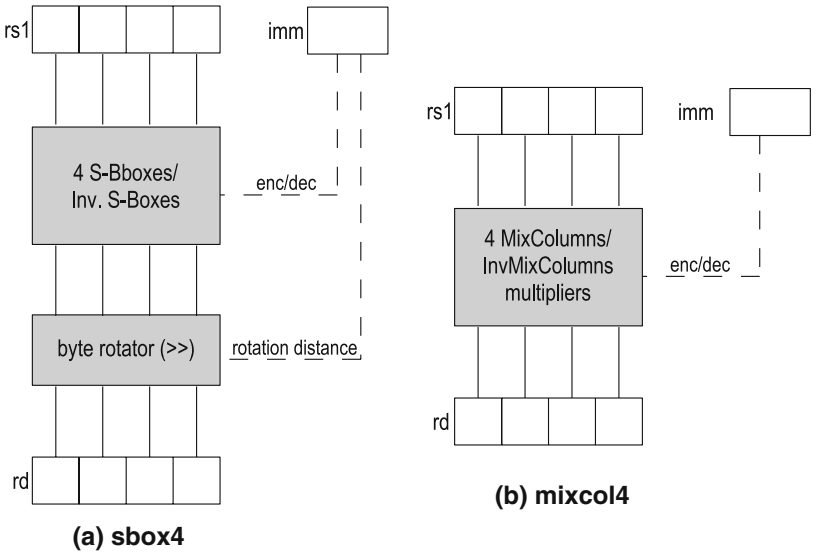

Fig. 9.12 Word-level *sbox* and *mixcol* instructions.

increase: 4.86x speedup vs. 1.74x speedup over the pure software implementation (no ISE). This is because the *Shift-Rows* AES operation becomes a bottleneck. Accelerating the most time-consuming parts of the algorithm caused other parts of the algorithm to become more important from the performance point of view.

The authors address this issue and propose a slight modification to the word-level circuits and instructions, so that they support a pair of input registers instead of only one. In this way, neglecting the details, the *Shift-Rows* transformation can be implicitly performed extracting 2 bytes from each of the two source registers. The speedup of this slightly improved ISE reaches 7.47x (8.35x with loop unrolling) over the non-extended software one and this is in line with expectations.

This experience should teach one that the ISE design and tuning is an iterative process that often has to be fed back from its own effects in order to obtain optimal results. The proposed approach does not rely on look-up tables and, therefore can be particularly suitable for constrained devices. The authors show that their ISE causes an 81% reduction in code size, while the well-known table-lookup approaches increase the code size by about 350–400%.

Tillich *et al.* in [54], explore the possibility of employing the ISEs for elliptic curve cryptography (ECC), assumed available on a given processor, to speed up AES. The considered ISE are:

- $gf2mul\ A,B$: 32-bit word-level polynomial multiplication, which generates a 63-bit result ($A \otimes B$) in the two 32-bit parts of an accumulator register $ACC.hi/ACC.lo$.
- $gf2mac\ A,B$: 32-bit word-level polynomial multiply and accumulate, which performs ($A \otimes B$), sums the result to $ACC.hi/ACC.lo$ and accumulates the outcome back to $ACC.hi/ACC.lo$.
- $shacr\ A$: shifts right the accumulator by 32-bit, so that $ACC.lo$ goes in A register and $ACC.hi$ goes in $ACC.lo$.

The MixColumns AES step requires to multiply two polynomials of degree 3 over GF(2^8), modulo x^4+1, where one polynomial is constant (specified by the algorithm) and the other is a column of the state. Essentially, each of such polynomials can be represented on a 4-byte word and the multiplication can be accelerated using $gf2mul$ instruction. After that operation, two more steps are needed: reduction of the polynomial coefficients, which live in GF(2^8), and polynomial reduction according to the modulus to obtain the correct 32-bit result.

The first operation can take advantage of $gf2mac$, while the second can benefit from *shacr*.

Considering that these extensions were not aimed specifically to AES, the performance increase from their usage is significant: +23% and +20% in the case of precomputed key schedule, for encryption and decryption, respectively.

The reader is highly encouraged to go into the details of the cited paper, as a useful exercise to fully understand the usage of the proposed approach.

9.3.3 ISE for ECC applications

Elliptic curve cryptography (ECC) was proposed independently by Victor Miller [44] and Neal Koblitz [33] in 1985 and is gaining interest as a viable alternative to "standard" public-key methods (like RSA), because of its shorter keys at the same security level. This can translate into faster implementations and reduced consumption of energy and bandwidth, which are crucial points, especially for embedded applications on constrained devices.

An elliptic curve for cryptography can be defined over a finite field (FF), *e.g.*, GF(p), GF(2^m), and is the set of points $P = (x,y)$ that satisfy the equation $y^2 + xy = x^3 + ax^2 + b$, $a,b,x,y \in FF, b \neq 0$, together with a point at infinity O. An addition operation defined on the curve points allows calculating integer multiples of a point: given a point P and an integer k, [k]P (*i.e.*, the scalar multiplication operation) produces another point Q on the same curve. Scalar multiplication can be naturally implemented through repeated doublings and additions of the point P, and it has security features similar to exponentiation in discrete-logarithm cryptosystems.

Elliptic curve point addition and doubling, in turn, can be calculated with a number of additions, multiplications, squarings, and inversions in the underlying binary finite field, through formulas operating on the coordinates of the involved points. For example, Figure 9.13 shows the formulas [43], needed to calculate EC addition and doubling in affine coordinates.

Handling finite-field elements in software requires multiprecision arithmetic because typical field sizes are hundreds of bits long. For instance, the sizes of EC binary fields delivering a security level similar to 1024, 2048 and 3076-bit RSA are 163, 233 and 283-bit, respectively.

Standard processors manage such big values (m-bit) as arrays of w-bit long words (where w typically is 8, 16, 32 or 64 bits, matching the word size of the processor). The number of required words is $\lceil m/w \rceil$.

Finite-field addition can be performed word by word, taking into account the carry propagation from each word to the immediately more significant one, in the case of GF(p). This can be tricky using a high-level language because there is no direct control over the carry propagation. In an assembler, according to the available ISA, it might be easier to obtain an efficient implementation. For instance, on Intel x86 processors, *ADC* instruction sums two operands plus the carry flag set by

$$
\begin{aligned}
x_3 &= \lambda^2 + \lambda + a + x_1 + x_2 \\
y_3 &= \lambda \cdot (x_1 + x_3) + x_3 + y_1 \\
\lambda &= \begin{cases} (y_2 + y_1)/(x_2 + x_1), & if P \neq Q \\ x_1 + y_1/x_1, & if P = Q \end{cases}
\end{aligned}
$$

Fig. 9.13 Elliptic curve point addition and doubling formulas, given $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$.

the previous instruction, allowing straightforward implementation of multiprecision additions. In case the result of an addition is bigger than the field prime, a reduction operation must be performed.

In GF(2^m), addition is simpler because it becomes a bitwise exclusive-OR and, therefore, it does not require to manage carries.

Field multiplication is quite expensive because, by nature, it *mixes* all the words of the two operands together and outputs a double-sized value. Multiplication can be done iteratively via repeated shifts and additions/XOR (*i.e.*, one per bit) to the partial product, as in the well-known schoolbook method for base 10 numbers. In each iteration, a number of instructions are needed to process the intermediate field values.

In the case of GF(p) fields, the algorithm can work at word level and be much more efficient. In fact, it can take advantage from the instructions and circuits available in the processor for integer multiplication (w-bit operands) so that the m-bit multiplication can be implemented via a number of w-bit multiplications. In the case of GF(2^m) this is not typically possible because there are almost no processors that natively support operations (*e.g.*, multiplication) on binary fields.

As a matter of fact, multiprecision modular multiplication, together with the modular inversion, may take most of the execution time of EC operations implemented in software [4, 25]. Therefore, their performance is crucial for the efficiency of high-level cryptographic protocols. Inversions are much more expensive computationally than multiplications but they can be almost totally avoided using a projective representation of the curve.

Concluding, ordinary processors are not particularly suited for efficient working on multiprecision operands and can be very inefficient in managing binary field GF(2^m) operations. For these reasons, specific ISEs have been studied to improve the ECC performance on these platforms.

9.3.3.1 Extended Instructions

Fiskiran and Lee in [16] do an extensive study on the performance of the elliptic curve public-key cryptosystem and, in particular, on the performance of the binary field GF(2^m) operations needed by ECC. The elements of a binary extension field are polynomials with coefficients from $\{0, 1\}$, having high degree (*e.g.*, 162, 192, 232). They highlight that the main bottleneck of performance is due to the fact that 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. The authors derived these results from an implementation of ECC on an Intel Pentium-II workstation. They used the Montgomery scalar multiplication algorithm described by Lopez *et al.* in [40] and which is the fastest method that does not require significant precomputations and/or storage.

They show that typical high-level application benchmarks for security protocols (EC Diffie-Hellman key-exchange protocol, EC digital signature generation/verification, and El-Gamal encryption/decryption) spend from about 94% up to 99% of the time in EC scalar multiplication k[P]. For this reason, the performance of this EC operation can be addressed in isolation as a good measure for overall application performance.

Using an instrumented version, through *gprof* GNU tool, of the scalar multiplication benchmark, the authors were able to break down the execution time of k[P] on the Pentium-II into the time components spent in the various code regions. In this way, it was possible to measure the time spent into squaring (with reduction), multiplication (with reduction), inversion and other modular operations. Table 9.2 shows these results and highlights that modular squaring and multiplication are the most important as they take 6.33% and 87.25% of the total execution time, respectively. Therefore, according to Amdahl's law, modular multiplication is the first target of possible optimizations for speeding up ECC.

The authors first analyze the intrinsic features of these field operations in order to understand the limits achievable through more complex processor microarchitectures. Using SimpleScalar [5], a cycle-accurate simulator for computer architecture, they simulate single-issue and multiple-issue processors (*i.e.*, capable of managing more than one instruction per cycle due to the parallelism in their pipeline). Results show that the critical operations (multiplication and squaring) implemented in software can have a speedup of more than 1.9x for a two-way execution core and more than 3.4x for a four-way one. This indicates the presence of a good number of independent instructions close to each other in the dynamic instruction flow of these benchmarks. The independent instructions are likely to originate from the operations on the various words that make up the field elements (*e.g.*, six 32-bit words for each 163-bit field element).

In addition, the authors show that adding one more load/store unit, to increase the memory bandwidth, brings almost no benefits to a two-way issue machine, while it improves slightly the performance of a four-way issue machine. Obviously, these results are very biased by the particular software implementation of the field operations. For instance, software implementations that rely on look-up tables of precomputed results are likely to benefit more from memory bandwidth increase. However, these results show that a relatively complex general-purpose processor can improve ECC performance significantly compared to a simple one.

Then, the authors investigate the effects of including a specific ISA support for word-level polynomial multiplication. They propose a *bfmul* instruction that works

Table 9.2 Execution time quote of finite-field operations in EC scalar multiplication $k[P]$.

Operation	% of Total Execution Time
Squaring (including reduction)	6.33
Multiplication (including reduction)	87.25
Inversion	1.51
Other	4.91

on two 32-bit operands and outputs the 64-bit product on a couple of special 32-bit registers *RH* and *RL*. The speedup of polynomial multiplication using this instruction on a single issue processor is almost 25x. The authors also analyze a simpler unit that can write only 32-bits of the result at a time. In this case, even if two instructions *bfmul.lo* and *bfmul.hi* are needed to obtain the lower and higher word of the result, respectively, the achieved speedup is still more than 24x. This indicates that even a narrower interface between the multiplier unit and the existing datapath is able to be effective.

In addition, the authors highlight that, using a *rev* instruction, which reverses the bit order within a word (*i.e.*, the most significant bit becomes the least significant one and so on), it is possible to substitute the *bfmul.hi* instruction by three *rev* instructions (two of them can be executed in parallel), a *bfmul.lo* and a shift instruction. In this way, the circuit can be further simplified because the circuit for bit reversing is far simpler than the one for *bfmul.hi*, as it does not need any logic gate but only wiring. Note that, for a number of DSP processors, the *rev* instruction comes for free because it is already part of their standard ISA. This is the code fragment that can substitute the *bfmul.hi* instruction:

```
rev t1, a       # t1, t2 are temporary variables
rev t2, b       # two rev instructions are independent
bfmul.lo t1, t1, t2
slli t1, t1, 1  # 1-bit logical shift left
rev t, t1
```

This approach allows a speedup of more than 17x over the single-issue machine, which is significantly lower than in the case of the other more complex approaches, but it is still a very good solution for designs where the hardware modifications have to be kept as small as possible.

In the same paper, the authors also address a possible extension for the the squaring operation in binary fields. Recall that the square of a binary polynomial has the same bits as the operand, but with zeroes interleaved between each pair of them. This can be easily accomplished by the *shuffle* instruction inspired by the DSP world. Such instruction reads one bit at a time alternatively from the two source registers and inserts them into the destination register. In this way, if the first register holds the polynomial coefficients and the second one holds zero, the output result is the squaring of the polynomial.

Again, some issues might come into the game because of the width of the output. In order to maintain the *shuffle* instruction with one-word output, it must work on half-word operands. For instance, *shuffle.lo* and *shuffle.hi* instructions could be provided so that the low and the high half-words could be processed, separately. Actually, the *shuffle.hi* can be emulated with a pair of preliminary shifts on the operands and using the *shuffle.lo*, reduce the overall circuit complexity.

The speedup of finite field squaring using the *shuffle* instruction is 3.8x over the software implementation.

Fiskiran and Lee further investigate the usage of complex processor microarchitectures when the above-mentioned extensions are available. A two-way issue processor, with either one or two load/store units and one or two multiplier units,

deliver about 1.75x speedup, which is slightly lower, but still in line with the results of the unmodified processor. In comparison, a four-way issue processor, with one or two load/store units and one or two multipliers, gives a speedup of about 2.2x, which is significantly lower than in the case of the software version on the unmodified processor. This means that the software that uses the ISE has a lower instruction-level parallelism (ILP) than the original one and, because of that, complex wide-issue general-purpose architectures are less beneficial in this case. However, the adoption of *bfmul* and *shuffle* instructions allow one to speed up the application-level benchmark (EC point multiplication) by more than 7.5x for a single issue processor and up to more than 22x for a four-issue processor with four ALUs, two load/store units, and two multiplier units. The adoption of the simpler variants of *bfmul* and *shuffle* do not affect these results too much.

The possible benefits from the usage of a specific optimized operation for word-level polynomial multiplication were first highlighted by Koç *et al.* in [34], where this operation was named MULGF.

Großshädl et al. in [22] highlight a proposal for fast ECC over binary fields GF(2^m) on a possible 16-bit smart card architecture. The contribution highlights the difficulty of implementing polynomial multiplication on the limited general-purpose architecture of a smart card. The authors propose the integration of a word-level polynomial multiplier unit within the existing datapath of the processor. They highlight that the overhead in hardware complexity can be very limited because the polynomial multiplier can share the same circuit as the integer one if a dual-field adder (DFA) [48] is used.

In fact, the DFA, shown in Figure 9.14 for only one bit addition, can be employed within the multiplier circuit so that the partial-product accumulation can be done using integer additions, when *fsel=1*, or polynomial additions (*i.e.*, XOR operations), when *fsel=0*. The authors highlight that the selection logic of a dual-field adder increases the area and the delay compared to a standard full adder, but when it works in polynomial mode, two NAND gates are forced to one and only the two XOR gates contribute to the dynamic power consumption. In this way, if the power consumption due to leakage is negligible compared to dynamic power, the DFA consumes significantly less power when working in polynomial mode than in integer mode.

Given the availability of the word-level multiplier for polynomials, and the corresponding new instruction *MULGF2*, word-level algorithms for multiprecision multiplication, squaring and modulo reduction can be employed in place of the bit-oriented ones. In this way, the machine parallelism can be fully exploited also in polynomial computations. For instance, as shown in Algorithm 1, the word-level schoolbook pencil-and-paper method for polynomial multiplication translates into a number of *MULGF2* instructions, marked as $\otimes$, and word-level *XOR* instructions, marked as $\oplus$. Any iteration of the inner loop carries out an operation of the form $(\tilde{u},\tilde{v}) \leftarrow \tilde{r}_{i+j} \oplus (\tilde{a}_j \otimes \tilde{b}_i) \oplus \tilde{u}$. The tuple $(\tilde{u},\tilde{v})$ is a double-precision quantity representing $u(t) \cdot t^w + v(t)$ (*i.e.*, a polynomial of degree 2w-1, where w is the processor word width).

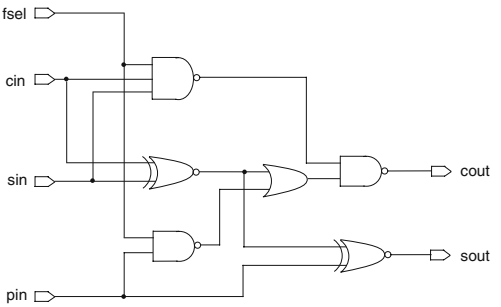

Fig. 9.14 Structure of a dual-field adder.

Algorithm 1 Pencil-and-paper method.

Input: Binary polynomials $a(t) = (\tilde{a}_{s-1}, ..., \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, ..., \tilde{b}_0)$ consisting of s word each
Output: Product $r(t) = a(t) \otimes b(t) = (\tilde{r}_{2s-1}, ..., \tilde{r}_0)$.

1: $r(t) \Leftarrow 0$
2: **for** i from 0 by 1 to $s-1$ **do**
3: $\quad \tilde{u} \Leftarrow 0$
4: $\quad$ **for** j from 0 by 1 to $s-1$ **do**
5: $\quad\quad (\tilde{u}, \tilde{v}) \Leftarrow \tilde{r}_{i+j} \oplus (\tilde{a}_j \otimes \tilde{b}_i) \oplus \tilde{u}$
6: $\quad\quad \tilde{r}_{i+j} \Leftarrow \tilde{v}$
7: $\quad$ **end for**
8: $\quad \tilde{r}_{s+i} \Leftarrow \tilde{u}$
9: **end for**
10: **return** $r(t)$

The Comba multiplication method [7] is shown in Algorithm 2. Comba's method forms the product $r(t)$ by computing each word $\tilde{r}_i$ of the result at a time, starting with the least significant word $\tilde{r}_0$. The partial products $\tilde{a}_j \oplus \tilde{b}_i$ are processed by columns instead of by rows as in the schoolbook pencil-and-paper method.

Comba's method for long integer multiplication can deliver performance advantages on processors having a multiply/accumulate unit (*e.g.*, digital signal processors) [18] because each word of the result is calculated by repeated multiply/accumulate (MAC) operations. Another potential advantage of Comba's method originates from keeping the running sum $(\tilde{u}, \tilde{v})$ in a register pair because, in this way, no store instructions are needed during the multiply/accumulates for computing each result word. Conversely, each iteration of the inner loop of schoolbook algorithm requires three memory accesses in order to load the values of $\tilde{a}_j$ and $\tilde{r}_{i+j}$, and write back the result to $\tilde{r}_{i+j}$.

In other words, Comba's method eliminates the write-back operation by changing the order of partial-product generation/accumulation such that each word of *r(t)* is computed completely before passing to the next one.

However, possible drawbacks of Comba's method can originate from the reversed addressing of the words of the operands *a(t)* and *b(t)*, along with the more complicated loop control. Nevertheless, on processors that support an

Algorithm 2 Comba's method for binary polynomials.

Input: Binary polynomials $a(t) = (\tilde{a}_{s-1}, ..., \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, ..., \tilde{b}_0)$ consisting of s word each
Output: Product $r(t) = a(t) \otimes b(t) = (\tilde{r}_{2s-1}, ..., \tilde{r}_0)$.

```
 1: (ũ, ṽ) ⇐ 0
 2: for i from 0 by 1 to s − 1 do
 3:     for j from 0 by 1 to i do
 4:         (ũ, ṽ) ⇐ (ũ, ṽ) ⊕ (ã_j ⊗ b̃_{i−j})
 5:     end for
 6:     r̃_i ⇐ ṽ
 7:     ṽ ⇐ ũ, ũ ⇐ 0
 8: end for
 9: for i from s by 1 to 2s − 2 do
10:     for j from i − s + 1 by 1 to s − 1 do
11:         (ũ, ṽ) ⇐ (ũ, ṽ) ⊕ (ã_j ⊗ b̃_{i−j})
12:     end for
13:     r̃_i ⇐ ṽ
14:     ṽ ⇐ ũ, ũ ⇐ 0
15: end for
16: r̃_{2s−1} ⇐ ṽ
17: return r(t)
```

auto-increment/decrement addressing mode, the first drawback might be canceled because the computation of the addresses of $\tilde{a}_j$ and $\tilde{b}_{i-j}$ comes for free.

The effects on performance of the memory behavior of Comba's method depend on the features and speed of the memory subsystem (*i.e.*, cache, if any, buses and RAM chips) of the target processor. It is very interesting to observe that the actual performance of different algorithms can be highly affected by the features of the execution platform: the availability of special instructions, as well as, the organization and speed of the memory system. Therefore, the selection, and the tuning, of the most performing algorithm must take into account very precisely the architecture of the system that will be running the specific implementation of the algorithm.

Großshädl et al. in [56] use ECC as a case study for analyzing when ISE can change algorithm design. The main results of the work are the following. First, ISE can reverse the relative interest of different algorithm versions. Second, automatic exploration of the best ISE can be viable for an algorithm designer in this field. In fact, they show that the considered automatic exploration tool was able to derive similar results as the ones obtained through manual exploration and simulation, even if with less accuracy in the performance estimates.

The authors present possible alternative algorithms for polynomial multiplications both in GF(p) and GF(2^m) finite fields and show how, in both cases, specific ISEs can help to boost performance.

For their experiments, the authors use the MIPS32 architecture, which has a multiply-and-accumulate integer instruction (*MADDU*) that can be very useful in the GF(p) multiplication algorithms. The *MADDU* instruction was aimed to DSP-like computations and operates as follows: multiplies two 32-bit words, adds the product to the 64-bit value in the concatenated *HI*/*LO* register pair, and writes back

```
L1: lw      $t0, 0($t1)     #load A[j]
    addiu   $t1, $t1, 4     #increment A pointer
    multu   $t0, $t4        #A[j]*B[i]
    lw      $t2, 0($t3)     #load Z[i+j]
    maddu   $t5, $t7        #add old U to product
    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    #if j!= s branch to L1
```

Fig. 9.15 Implementation of the inner loop of schoolbook multiplication in GF(p).

the result to *HI*/*LO*. Using the *MADDU* instruction, the core-loop of the schoolbook pencil-and-paper method ($a \cdot b + z + u$, where a, b, z, u are 32-bit values) can be mapped on only 11 instructions as in Figure 9.15. The main residual problem in this method is that the two additions of 32-bit values to the 64-bit product might generate carry bits and so they must be managed as double-precision additions, with increased complexity.

The authors describe that if the instruction set could be augmented with an instruction that is able to perform multiply, accumulate and the two additions (named *MADDL* as in [20]), the inner loop could be made even faster. This extension can increase slightly the area and the delay of the circuit but could shrink the code shown in Figure 9.15 from 11 down to only 7 instructions. Apart from the details that can be found in [20], if load/stores are assumed to hit in the data cache in both cases, all instructions can be assumed to run in one cycle. In this way, *MADDL* instruction can enable a significant speedup (1.57x) over the code that uses *MADDU*.

For the inner loop of Comba's method on GF(p) fields, the *MADDU* instruction appears to provide exactly the functionality needed by the concatenated multiply/accumulate operations. However, multiple accumulations on the 64-bit HI/LO register pair cannot be done without possible overflow. For this reason, even using *MADDU* instruction, the inner loop of Comba's takes no less than 18 cycles. However, another simple custom instruction extension could be very beneficial in this case. If the accumulator of the *MADDU* (*HI*/*LO* pair) is widened, for instance *HI* register is extended to 40-bit, up to 256 *MADDU* could be performed without risk of overflow. This could be enough for ECC applications and would not increase significantly the complexity and the delay of the circuit. Großshädl et al. highlight that a careful implementation of the Comba core loop, employing this modification, could be run in only 6 clock cycles. In this way, the possibility of customizing the instruction set according to the algorithm features allows the slowest algorithm with unmodified ISA (Comba) to be the absolute fastest when a custom-tailored ISA is employed.

On GF(2^m) fields, the authors perform a similar analysis and show that, using the standard ISA, the inner loop of the polynomial multiplication can execute in only 10 cycles. It seems in line with GF(p), but the difference is huge because the inner

loop on GF(2^m) is performed many more times because of the lack of word-level instructions for polynomials. The number of iterations is a function of the number of bits of the operands instead of the number of words, as shown in Algorithm 3 for the shift-and-XOR multiplication algorithm.

Several improvements on this method have been proposed and, essentially, they rely on precomputed look-up tables which limit the number of operations. The look-up table must be calculated online because it contains multiples of $a(t)$ and, therefore, the table length has to be tuned taking into account the trade-off between the pre-computation overhead and the resulting performance benefits. This trade-off is certainly dependent on the particular processor features. For instance, in [26, 41] the best trade-off is shown to be a 16-entry table (*i.e.*, 16 multiples of $a(t)$), which allows it to process $b(t)$ four bits at a time. In this way, only 1/4 of the multiprecision shifts are needed, where each one is a four-bit shift.

The usage of a look-up table causes the memory traffic to increase in the inner loop and increases the overall memory footprint of the algorithm.

Algorithm 3 Pencil-and-paper method.

Input: Binary polynomials $a(t) = (\tilde{a}_{s-1}, ..., \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, ..., \tilde{b}_0)$ consisting of s words each
Output: Product $r(t) = a(t) \otimes b(t) = (\tilde{r}_{2s-1}, ..., \tilde{r}_0)$.

```
1: r(t) ⇐ 0
2: for i from 0 by 1 to s − 1 do
3:     ũ ⇐ 0
4:     for j from 0 by 1 to s − 1 do
5:         (ũ, ṽ) ⇐ r̃_{i+j} ⊕ (ã_j ⊗ b̃_i) ⊕ ũ
6:         r̃_{i+j} ⇐ ṽ
7:     end for
8:     r̃_{s+i} ⇐ ũ
9: end for
10: return r(t)
```

If a word-level approach is also used in GF(2^m), the performance of the multiplication algorithm becomes strictly related to the efficiency of the MULGF2 operation (*i.e.*, multiplication of two 32-bit polynomials, obtaining a 63-bit result), which must be implemented in software using the standard processor ISA. The authors of [56], highlight that the sole MULGF2 operation can be emulated in no less than 190 cycles on their MIPS32 processor, in the case of a GF(2^{191}) binary field.

The hardware implementation of the MULGF2 operation is possible on a polynomial multiplier or even on a unified dual-field multiplier [48], able to manage integers and polynomials at the cost of a small increase in area and delay. The performance benefits are impressive because the hardware MULGF2 can be implemented to run in only one or two cycles, against 190 of the software emulation.

Table 9.3 shows the experimental results of the performance of the multiplication algorithm, as well as of the EC scalar multiplication operation, for both standard ISA and algorithm-specific ISE.

Table 9.3 Execution time (cycles) of the various multiplication algorithms on GF(p) and GF(2^m) for the standard ISA and with algorithm-specific ISE.

Operation	GF(p) Schoolbook	GF(p) Comba	GF(2^m) Shift-XOR	GF(2^m) Word-level
Field mul. (std ISA)	629	827	2758	7848
Field mul. (with ISE)	485	**441**	2151	456
Point mul. (std ISA)	2160k	2840k	4050k	10420k
Point mul. (with ISE)	1670k	1470k	3280k	**870k**

The table shows very interesting results for ECC on both prime and binary fields at the same security level: GF(p), with $p = 2^{192} - 2^{64} - 1$, and GF(2^{191}). The fastest multiplication operation on the field is achieved for GF(p) with the described ISE (441 cycles), even if the result for GF(2^m) and custom ISE is very close to it (456 cycles).

However, the results for the EC scalar multiplication k[P], which is significant at application level, show a different trade-off. GF(2^m) results significantly outperform GF(p) ones (870 vs. 1470 kilo cycles), indicating the complex interaction between architectural features of the target system, the instruction set architecture and its extensions, the field-level, and the EC-level algorithms.

In a previous paper [23], Großshädl et al. highlight the useful ISEs for prime (GF(p)) and binary (GF(2^m)) finite fields and present some additional instructions to the ones that we have already described. For instance, on GF(p), they propose *M2ADDU Rs, Rt* which is a slightly modified multiply-and-accumulate (*MADDU*) which doubles the partial product before accumulation. The additional complexity is negligible because doubling on integers requires only a left shift. This instruction is very useful in multiple-precision squaring of integers.

The *ADDAU Rs, Rt* adds the two input registers and accumulates the result on the *HI*/*LO* register pair. This is useful to support multiple-precision addition and reduction modulo a generalized Mersenne prime.

The *SHA* performs a 32-bit right shift of the *HI*/*LO* accumulator, in order to move the value in *HI* into the *LO* register.

For GF(2^m), they propose *MULGF2 Rs, Rt* and *MADDGF2 Rs, Rt*, which perform the word-level polynomial multiplication and multiply-and-accumulate, respectively.

As a training exercise, the reader is encouraged to study the paper [21], which presents a study on possible ISEs in the case of optimal extension fields (OEF). The approach has similarities with the other ones already discussed but has also some differences originating from the specific features of OEFs.

A similar study has been conducted by Eberle et al. in [10] on an 8-bit ATmega128 8 MHz processor. The authors show that simple extensions of the available datapath suffice to efficiently support ECC over GF(2^m) and, in addition, to outperform GF(p). The extensions include a dual-field multiplier for both integers and

polynomials using the carry-save adder (CSA) tree structure. The proposed ISEs comprise *MULX* and *MULACCX* instructions, which are designed so as to maintain compatibility with existing instructions (*i.e.*, two operands only), and the existing datapath (*i.e.*, up to two source operands can be read and two destination operands can be written by a functional unit).

- *MULX* R_d, R_r instruction performs the polynomial multiplications between R_d and R_r and puts the doubleword result into the R1:R0 register pair.
- *MULACCX* R_d, R_r instruction performs a multiply-accumulate operation with extended carry. In particular, $R_d \leftarrow bits[7:0]((R_r \otimes R_c) \oplus XC \oplus R_d)$, and $XC \leftarrow bits[15:8]((R_r \otimes R_c) \oplus XC \oplus R_d)$, where R_c is an implicit architectural register and XC is a non-architectural register. Note that R_c must be loaded prior to executing *MULACCX*.

Figure 9.16 shows in grey the words that can be fruitfully managed by a *MULACCX* instruction. In fact, the basic idea is to reuse in the following *MULACCX* the XC part of the result obtained by the previous one. In particular, the instruction would be used in this way: *MULACCX* R_d, R_r, where $R_d = B_n$, $R_r = c_{n+p}$, would calculate $c_{n+p} = (a_p \otimes b_n) \oplus (bits[15:8](a_p \otimes b_{n-1}) \oplus c_{n+p})$, where R_c holds a_p and XC holds $bits[15:8](a_p \otimes b_{n-1})$. The example in the figure highlights when $p = 1$ and $n = 3$.

Note that the XC register is not explicitly accessible to the programmer. When XC must be saved and restored, for instance because of a system call, its value can be retrieved using a dummy *MULACCX* $R_d = 0$, $R_r = 0$ instruction to move XC value to R_d. The dual operation can be performed by a similar technique (the reader should look for it as a useful training exercise).

The performance originating from such ISEs are significant: 0.40 and 0.29 seconds for EC scalar multiplication on GF(2^{163}) using *MULX* and *MULACCX*,[1]

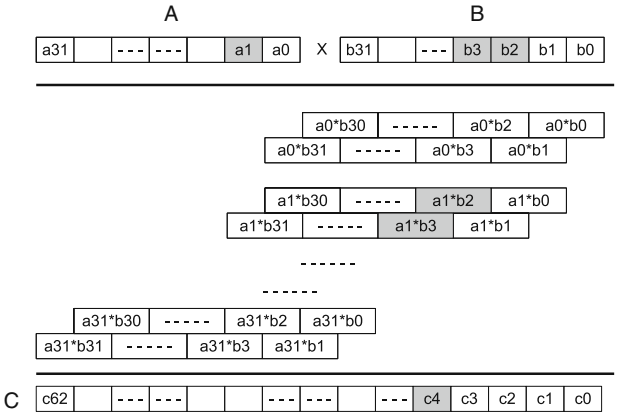

Fig. 9.16 Multiple-precision multiplication in GF(2^m). *MULACCX* instruction can speed up the management of the data highlighted in grey.

[1] executing in one cycle.

respectively, compared to 4.14 seconds of the unmodified ISA. Moreover, the performance of k[P] for GF(p) on a 160-bit prime is 0.81 seconds.

As a comparison, RSA performance for the same security level (1024-bit), are 0.43 and 10.99 seconds for public-key and private-key operations, respectively. Moreover, switching to higher security level, 2048-bit RSA and 233-bit ECC, the speedup given by the ISE over the standard ISA is almost the same (about 14x) but the performance gap between ECC and RSA widens. In fact, in this case, ECC with *MULX* and *MULACCX* runs in 1.12 and 0.81 seconds, respectively, against 10.98 seconds of the unmodified ISA, while RSA public-key and private-key timings are 1.94 and 83.26 seconds, respectively.

Bartolini et al. in [1] analyze the performance of some ECC applications (Diffie-Hellman key-exchange, digital-signature algorithm, El-Gamal encryption/decryption) on a 32-bit ARM-based Intel XScale processor. The performances are investigated through SimpleScalar, [5] a cycle-accurate simulator of the target processor which allows one to analyze the behavior of all internal modules of the processor, as well as of the memory hierarchy (*i.e.*, caches, memory bus, RAM). A specifically modified version of the simulator allowed to highlight the time spent in each of the finite-field, elliptic-curve and other operations.

Figure 9.17 shows that 32-bit word-level polynomial multiplication (*mr_mul*2) takes up a significant fraction of the time on the considered benchmarks. In particular, from 18% in the digital signature benchmarks (*ecDSsign* and *ecDSver*), which, however, spend more than 64% of the time in hash generation and file reading, up to 54% in Diffie-Hellman key-exchange protocol (*ecDH*).

Operation *mr_mul*2 is the software procedure that performs a 32-bit word-level polynomial multiplication, similar to the MULGF2 operation cited by Koç and Acar in [34]. The procedure is optimized through loop unrolling and by the use of a small look-up table that speeds up the shift-and-XOR approach. However, on the given architecture and with the particular library (MIRACL [53]), it takes about 400 dynamic instructions (roughly 500 cycles), which correspond to about 12 instructions per bit. The library is able to support different operand sizes and, therefore, is not fully optimized for a particular key size.

Apart from *mr_mul*2, the other operations highlighted in Figure 9.17 are:

- reduce2: GF(2^m) modulo reduction
- mr_sqr2 and square2: word-level and GF(2^m) squaring, respectively
- karmul2: manages Karatsuba algorithm for GF(2^m) polynomial multiplication
- mr_bottom4: a base case of Karatsuba algorithm
- add2: GF(2^m) addition, *i.e.* XOR operation over m-bit polynomials
- numbits: bit count
- hash and file read: hash algorithm and file-reading activities
- other: other functions for the management of finite-field values

The figure highlights that, among the cryptographic operations, the most time-consuming one is *mr_mul*2, and then, with a notably smaller weight, the modular reduction, word-level squaring and the multiprecision management of Karatsuba multiplication.

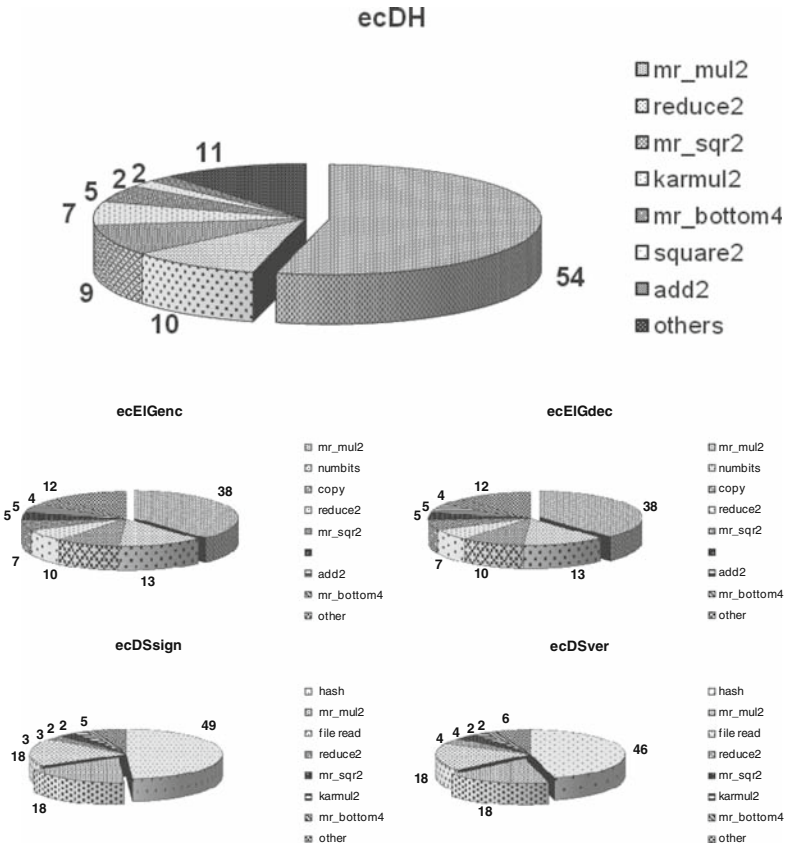

Fig. 9.17 Execution time breakdown for various EC benchmarks. Legend items are positioned clockwise on the pies and the word-level 32-bit polynomial multiplication (*mr_mul*2) is highlighted.

The authors evaluate the effects of including a 32-bit word-level polynomial multiplication instruction, named *MULGF*, to support *mr_mul*2 in hardware through a specific multiplier unit. A specific multiplier could be used, simpler than the integer one, or a dual-field multiplier [48] for both polynomials and integers. The latency of the multiplier is modeled conservatively, equal to the integer multiplier: three cycles, which is reasonable because the polynomial multiplication circuit can be quicker than the integer one because of the lack of carry propagation. The software procedure for word-level polynomial multiplication was substituted by the *MULGF* instruction throughout the ECC library using assembler inlining, and the GCC cross-compiler was modified for managing the additional *MULGF* instruction. In this way, the compiler is able to apply a number of optimizations to the extended instruction flow (*e.g.*, instruction scheduling, register allocation, generation of machine code).

Table 9.4 Effects of the adoption of *MULGF* instruction on both execution time and on the number of executed instructions.

Benchmark	Execution time %	Dyn. instruction %
ecDH	-54	-55
ecElGenc	-39	-48
ecElGdec	-35	-37
ecDSsign	-17	-19
ecDSver	-17	-19

In this way, the 400 dynamic instructions for executing *mr_mul*2 are collapsed into only one *MULGF* instruction and, correspondingly, the 500 cycles of the software *mr_mul*2 are distilled into the *MULGF* latency of three cycles.

For GF(2^{233}), Table 9.4 shows that the resulting improvement in execution time is more significant for Diffie-Hellman (54% in number of instructions and 55% in execution time) and El-Gamal algorithms (48% and 37% less instructions for encryption and decryption, respectively, corresponding to 39% and 35% less execution time), where 32-bit polynomial multiplication is used more. The improvement for digital signature algorithm is less evident (19% in instruction number and 17% in execution time for the same key length) because of the included hash time.

Apart from the plain performance speedup, the authors investigate on the origins of the performance and highlight the effects of the ISE on the instruction-level parallelism that the processor is able to exploit. Figure 9.18 shows the cycles per instruction (CPI) performance metric in the case of a 1Byte I-Cache and D-Cache organization. The CPI is split into the cycles spent for actual execution of the instructions (CPI-processor) and the cycles spent in waiting for memory (CPI-memory): operands (load/stores) or instructions (fetch). In addition to reducing the number of dynamically executed instructions, the ISE allows one to reduce the CPI, which means that the processor executes the remaining instructions faster. In particular, the CPI-memory quote is significantly reduced due to a more efficient behavior of the caches. This is due to the smaller footprint of the algorithm in terms of executed instructions (*i.e.*, less instruction fetches) and to the reduced number of spills from registers to memory in the computation of the *mr_mul*2 operation (*i.e.*, less load/stores).

Kumar and Paar in [35] evaluate the usage of a 163-bit *full-width* coprocessor for performing modular multiplication on a simple 8-bit processor. They propose that the additional hardware is closely coupled with the ALU of the processor, reducing the interface circuitry and aiming for efficient implementation. The circuit is designed to deliver high performance on GF(2^{163}) multiplications, but it cannot be as flexible as word-level ISE approaches, which can easily accommodate to different-sized fields.

The proposed extension is able to directly access the data-RAM so that the main processor can avoid transfer of data to/from memory for using the unit. The proposed 163-bit multiplier unit takes 42 cycles to execute, which become 193 cycles including all the set up and control overhead.

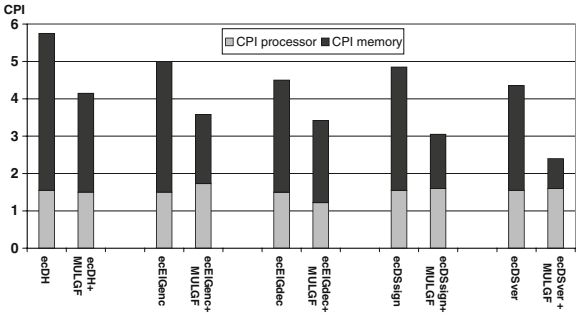

Fig. 9.18 Cycles Per Instructions (CPI) for the considered EC benchmarks for original and MULGF extended instruction set.

The unit is managed through an extended instruction which triggers the functional unit to load the operands (21 8-bit words) directly from memory, to execute the multiplication and to store back the result to memory. The address of the operands and of the result should be prepared into three fixed processor registers. During the multiplication, the processor must poll the unit for understanding when the operation has finished. An alternative is to implement and program the unit to raise an interrupt upon completion.

The proposal delivers a speedup of more than 30x over the original software implementation and allows the EC scalar multiplication (k[P]) to execute the operation in 169ms on a 4 MHz 8-bit microcontroller. In addition, the proposed ISE shrinks down the code and data size of the k[P] to 2048 and 273 bytes, respectively, whereas the original application required 8208 and 358 bytes, respectively.

Batina et al. in [29] propose to extend an 8-bit 8051 processor with a full-width multiply-and-accumulate unit for GF(2^{83}) which is interfaced toward the processor through an 8-bit bus. The unit width is suitable for hyper-elliptic curve cryptography (HECC) on GF(2^{83}) binary field.

The unit presented here is equipped with three full-width registers for holding modular multiplication operands (A, B) and result (C). Such registers are write (A,B) and read (C) in 8-bit chunks.

The unit is able to perform full-width multiplication and addition operations, as well as move operations between C and B register. In this way, multiply and accumulate can be performed in two steps (multiplication and accumulation) reusing part of the values already present in the unit.

The performance time for executing a modular multiplication with the new unit is 28.2 K-cycles. Modular addition is again 28.2 K-cycles, even if the plain circuit takes 83 cycles for a multiplication and 1 cycle for the addition. This is because much of the time is spent in I/O operations to transfer the operands to/from the unit. In software, the corresponding results are 650 K-cycles and 38 K-cycles, respectively. The multiply-and-accumulate operation using the new unit takes advantage of the already available multiplication result and runs in only 30.5 K-cycles. Also, in HECC, modular multiplication is the most time-consuming operation and

therefore, ISEs for modular multiplication can be very useful, especially in constrained devices.

It is interesting to highlight the proposal from Grabbe et al. [19], which is not exactly an ISE approach but a processor proposal for finite-field arithmetic and ECC. In fact the authors propose a very long instruction word (VLIW) processor with a number of full-width units able to work on 233-bit field values: adder, multiplier and squarer.

The adder takes 1 cycle to process two or three operands. The multiplier latency is 9 cycles but it is pipelined so that every two cycles a new operand pair can be fed to the unit. The latency of the squarer is one cycle.

Two independent register files hold four 233-bit registers each. We will not go into the details of the architecture of the processor here, because it goes beyond the scope of this chapter, but we want to highlight a design choice that allowed one to introduce some high-level instructions for ECC.

In fact, the proposed processor is able to manage the high-level EC operations through specific instructions which are executed via a microcoded approach. EC point addition, doubling, inversion, and the scalar multiplication are supported natively.

Obviously, there is no hardware circuit that executes such instructions as a whole. Each of these instructions is executed through a sequence of finite-field operations, supported by the processor functional units, which are orchestrated according to a microprogram stored in a control unit.

In other words, the processor supports two kinds of instructions, the ones directly executed by a specific unit (*e.g.*, field squaring, multiplication and addition) and others, more complex, which are executed through a *program* that uses the available processor units.

This can be an interesting approach to raise the level of abstraction of the operations managed by the processor autonomously. Certainly, this also leads to a less flexible architecture because the microcode that implements the high-level algorithms is fixed.

However, the availability of the lower-level instructions, allows one to program explicitly alternative algorithms also for EC operations, even if in a less efficient way than with microcode.

The performance results of this full-width microcoded approach are impressive: 31 and 42 clock cycles to execute 233-bit EC-doubling and EC-addition, respectively, which translate into about 12 K-cycles for an average EC scalar multiplication. The microcode implements the EC scalar multiplication using the add-and-double technique.

9.4 Exercises

1. Discuss the benefits of elliptic curves for public-key cryptography and show an instruction-set extension suited for a cryptographic system based on this approach.

2. Analyze the performances of the pencil-and-paper and Comba's methods for polynomial multiplication discussed in Section 9.3.3. The idea is to implement the algorithms in a high-level language and then inspect the code. Assume that assignments and additions/subtractions on integers and logic operations (*e.g.*, XOR) take 1 cycle, memory accesses (read/writes) take 3 cycles. Then, consider the following variant: up to 8 integer values (variables) can be maintained in the processor registers without the need of update/reread the memory location of the corresponding variable up to when the register has to be reassigned a new value. This should refine the estimation of the required memory access time.

9.5 Projects

1. Implement a multiprecision addition and multiplication procedure for 1024-bit integers and measure the performance through repeated random testing: generate random numbers for the operands, *e.g.* using the random number generator of the adopted high-level language (*e.g.*, outputting 32-bits at a time) to fill the 1024-bit operands.
2. Implement a simple symmetric block-cipher that uses a 128-bit symmetric key and works as follows. The plain text is encrypted 128-bit (block) at a time doing an XOR operation with the key. The decryption process is done exactly in the same way, using the same key. The key for encrypting a block is obtained by the key of the previous block multiplying it by 3 and getting the result mod 2^{128}. The first key is the initial key. Small plaintexts and the last block of bigger ones should be managed properly. Implement the solution in a high-level language and then try to apply some optimizations using the assembler on the target machine. For instance, exploit the carry flag and add with carry instructions for multiprecision additions.
3. Implement the AES block-cipher from the documentation[2] using a high-level language and analyze the performance of the main high-level operations. A possible way is to evaluate the performance of each of them separately and then count the number of times they are called in an AES encryption/decryption. In this way, an estimate can be drawn for the time spent in each operation by AES. Then, discuss the performance benefits from possible ISEs that accelerate specific operations.
4. Implement a multiprecision polynomial library GF(2^{163}), adopting the NIST irreducible polynomial ($p(x) = x^{163} + x^7 + x^6 + x^3 + 1$) and implementing addition (XOR), reduction, squaring and multiplication. Try to implement the field operations relying on word-level operations (*e.g.*, 163-bit multiplication, *fieldMult*, done in 32-bit chunks, *wordMult*.) which are implemented in specific functions. Measure the time spent in the various word-level functions when executing the high-level field operations and analyze the word-level function that take up most

[2] see, e.g., http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

of the time and thus would deserve a specific ISE. Consider the following relative number of invocations for the high-level library functions, normalized to 100 overall invocations: modular multiplication 20, modular squaring 60 and addition 20 of the time.

References

1. S. Bartolini, I. Branovic, R. Giorgi, and E. Martinelli. A performance evaluation of arm isa extension for elliptic curve cryptography over binary finite fields. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pp. 238–245, 27-29 Oct. 2004. 10.1109/SBAC-PAD.2004.5.
2. G. M. Bertoni, L. Breveglieri, F. Roberto, and F. Regazzoni. Speeding up AES by extending a 32-bit processor instruction set. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pp. 275–282, Sept. 2006. 10.1109/ASAP.2006.62.
3. I. Branovic, R. Giorgi, and E. Martinelli. A workload characterization of elliptic curve cryptography methods in embedded environments. *ACM SIGARCH Computer Architecture News*, 32(3):27–34, June 2004. ISSN 0163-5964. http://doi.acm.org/10.1145/1024295.1024299.
4. M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the nist elliptic curves over prime fields. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pp. 250–265, London, UK, 2001. Springer-Verlag. ISBN 3-540-41898-9.
5. D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997. ISSN 0163-5964.
6. J. Burke, J. McDonald, and T. Austin. Architectural support for fast symmetric-key cryptography. *SIGPLAN Not.*, 35(11):178–189, 2000. ISSN 0362-1340. http://doi.acm.org/10.1145/356989.357006.
7. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.
8. Counterpane Internet Security Inc. *The blowfish encryption algorithm*, 1993. http://www.counterpane.com/blowfish.html.
9. J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2.
10. H. Eberle, A. Wander, N. Gura, Sheueling Chang-Shantz, and V. Gupta. Architectural extensions for elliptic curve cryptography over gf(2/sup m/) on 8-bit microprocessors. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pp. 343–349, 23-25 July 2005. 10.1109/ASAP.2005.15.
11. H. Eberle, N. Gura, S. C. Shantz, V. Gupta, L. Rarick, and S. Sundaram. A public-key cryptographic processor for rsa and ecc. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pp. 98–110,

Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2226-2. http://dx.doi.org/10.1109/ASAP.2004.6.

12. A. J. Elbirt. Fast and efficient implementation of AES via instruction set extensions. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pp. 396–403, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2847-3. http://dx.doi.org/10.1109/AINAW.2007.182.
13. Federal Information Processing Standards Publication 46-1. *Data encryption standard (DES)*, 1988.
14. Federal Information Processing Standards Publication 46-3. *Data encryption standard (DES) - tdea*, 1999.
15. Federal Information Processing Standards Publication 197. *Specification for the advanced encryption standard (AES)*, 2001.
16. A. M. Fiskiran and R. B. Lee. Evaluating instruction set extensions for fast arithmetic on binary finite fields. In *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004*, pp. 125–136. IEEE Computer Society, 2004. ISBN 0-7695-2226-2.
17. A. M. Fiskiran and R. B. Lee. Performance scaling of cryptography operations in servers and mobile clients. In *Proceedings of the Workshop on Building Block Engine Architectures for Computer Networks (BEACON)*, 2004.
18. J. R. Goodman. *Energy scalable reconfigurable cryptographic hardware for portable applications*. PhD thesis, Massachusetts Institute of Technology, Deptartment of Electrical Engineering and Computer Science, 2000.
19. C. Grabbe, M. Bednara, von zur Gathen, J. Shokrollahi, and J. Teich. A high performance vliw processor for finite field arithmetic. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 6pp., 22–26 April 2003. 10.1109/IPDPS.2003.1213351.
20. J. Großschädl and G.-A. Kamendje. Optimized RISC architecture for multiple-precision modular arithmetic. In *International Conference on Security in Pervasive Computing, LNCS*, 2003.
21. J. Großschädl, S. S. Kumar, and C. Paar. Architectural support for arithmetic in optimal extension fields. In *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 111–124, 2004. 10.1109/ASAP.2004.1342463.
22. J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m). In E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, editors, *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468. IEEE Computer Society Press, 2003. ISBN 0-7695-1992-X.
23. J. Großschädl and E. Savaş. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pp. 133–147. Springer Verlag, 2004. ISBN 3-540-22666-4.

24. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. http://dx.doi.org/10.1109/WWC.2001.15.
25. D. Hankerson, J. López, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *International Workshop on Cryptographic Hardware and Embedded Systems - CHES*, pp. 1–24, 2000.
26. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 038795273X.
27. Y. Hilewitz and R. B. Lee. Performing advanced bit manipulations efficiently in general-purpose processors. In *IEEE Symposium on Computer Arithmetic*, pp. 251–260, 2007.
28. Y. Hilewitz, Z. Jerry Shi, and R. B. Lee. Comparing fast implementations of bit permutation instructions. In *Proceedings of the 38th Annual Asilomar Conference on Signals, Systems, and Computers*, pp. 1856–1863, "November" 2004.
29. A. Hodjat, L. Batina, D. Hwang, and I. Verbauwhede. Hw/sw co-design of a hyperelliptic curve cryptosystem using a microcode instruction set coprocessor. *Integr. VLSI J.*, 40(1):45–51, 2007. ISSN 0167-9260. http://dx.doi.org/10.1016/j.vlsi.2005.12.011.
30. Intel. *IA-64 Architecture Software Developer's Manual*, May 1999.
31. Intel. *Ia-32 intel architecture software developers manual volume 1: Basic architecture*, 2004.
32. Intel. *Intel SSE4 programming reference*, July 2007.
33. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48: 203–209, ISSN 0025–5718 1987.
34. Ç. K. Koç and T. Acar. Montgomery Multiplication in GF(2^k). *Des. Codes Cryptography*, 14(1):57–69, 1998. ISSN 0925-1022. http://dx.doi.org/10.1023/A:1008208521515.
35. S. S. Kumar and C. Paar. Reconfigurable instruction set extension for enabling ecc on an 8-bit processor. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *FPL*, volume 3203 of *Lecture Notes in Computer Science*, pp. 586–595. Springer, 2004. ISBN 3-540-22989-2.
36. X. Lai. *On the Design and Security of Block Ciphers*. Hartung-Gorre Verlag, 1992.
37. R. B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989.
38. R. B. Lee. Subword parallelism with MAX-2: Accelerating media processing with a minimal set of instruction extensions supporting efficient subword parallelism. *IEEE Micro*, 16(4):51–59, August 1996. ISSN 0272-1732.
39. R. B. Lee, Z. Shi, and X. Yang. Cryptography efficient permutation instructions for fast software. *IEEE Micro*, 21(6):56–69, 2001.
40. J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In *CHES: International Workshop on Cryptographic Hardware and Embedded Systems, CHES, LNCS*, 1999.

41. J. López and R. Dahab. High-speed software multiplication in f2m. In *INDOCRYPT '00: Proceedings of the First International Conference on Progress in Cryptology*, pp. 203–212, London, UK, 2000. Springer-Verlag. ISBN 3-540-41452-5.
42. J. P. McGregor and R. B. Lee. Architectural enhancements for fast subword permutations with repetitions in cryptographic applications. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '01)*, pp. 453–461, Washington - Brussels - Tokyo, September 2001. IEEE. ISBN 0-7695-1200-3.
43. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Norwell, MA, USA, 1994. ISBN 0792393686. Foreword By-Neal Koblitz.
44. V. S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, pp. 417–426, Santa Barbara, California, USA, August 1985.
45. National Institute of Standrds and Technology. *Fips-197: Advanced encryption standard*, November 2001. http://csrc.nist.gov/publications/fips/.
46. National Institute of Standrds and Technology. *Fips-180-2: Secure hash standard*, August 2002. http://csrc.nist.gov/publications/fips/.
47. C. Paar. The future of the art of cryptographic implementations. In *Position Statement for the STORK Workshop*, Brussels, Nov. 2002.
48. 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 *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 277–292, London, UK, 2000. Springer-Verlag. ISBN 3-540-41455-X.
49. Princeton Architecture Laboratory for Multimedia and Security (PALMS). *Pax project*, 2003. http://palms.ee.princeton.edu/PAX.
50. C. E. Shannon. Communication theory of secrecy systems. *Bell Systen Technicl Journal*, 28:656–715, October 1949.
51. Z. Shi and R. B. Lee. Bit permutation instructions for accelerating software cryptography. In *ASAP '00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 138, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0716-6.
52. Z. Shi, X. Yang, and R. B. Lee. Arbitrary bit permutations in one or two cycles. In *ASAP '03: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 237. IEEE Computer Society, 2003. ISBN 0-7695-1992-X.
53. S. Software. *MIRACL: Multiprecision Integer and Rational Arithmetic C/C++ Library*, 1988. http://www.shamus.ie/.
54. S. Tillich and J. Großschädl. Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography. In Marina Gavrilova, Youngsong Mun, David Taniar, Osvaldo Gervasi, Kenneth Tan, and Vipin Kumar, editors, *Computational Science and Its Applications - ICCSA 2005*, volume 3481 of *Lecture Notes in Computer Science*, pp. 665–675. Springer, 2005.
55. S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In Louis Goubin and Mitsuru Matsui,

editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10–13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pp. 270–284. Springer, 2006.

56. A. K. Verma, L. Pozzi, P. Ienne, S. Tillich, and J. Großschädl. When instruction set extensions change algorithm design: A study in elliptic curve cryptography. In *4th Workshop on Application-Specific Processors (WASP 2005)*, p. 29, Jersey City, NJ, USA, September 2005.
57. L. Wu, C. Weaver, and T. Austin. Cryptomaniac: a fast flexible architecture for secure communication. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 110–119, New York, NY, USA, 2001. ACM Press. ISBN 0-7695-1162-7. http://doi.acm.org/10.1145/379240.379256.
58. X. Yang and R. Lee. Fast subword permutation instructions using omega and flip network stages. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, pp. 15–22, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0801-4.
59. X. Yang, M. Vachharajani, and R. Lee. Fast subword permutation instructions based on butterfly networks. In *Proceedings of SPIE, Media Processor*, pp. 80–86, January 2000.
60. P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995.