Instruction Set Extensions for Enhancing the Performance of Symmetric-Key Cryptography

Sean O’Melia † Member, IEEE, and AJ Elbirt ‡ Member, IEEE

Abstract

Instruction set extensions for a RISC processor are presented to improve the software performance of the Data Encryption Standard (DES), Triple-DES, the International Data Encryption Algorithm (IDEA), and the Advanced Encryption Standard (AES) algorithms. The most computationally intensive operations of each algorithm are off-loaded to a set of newly defined instructions. The additional hardware required to support these instructions is integrated into the processor’s datapath. For each of the targeted algorithms, comparisons are presented between traditional software implementations and new implementations that take advantage of the extended instruction set architecture. Results show that utilization of the proposed instructions significantly reduces program code size and improves encryption and decryption throughput. Moreover, the additional hardware resources required to support the instruction set extensions increases the total area of the processor by less than 65%.

Keywords: symmetric-key, cryptography, software, FPGA

1 Introduction

With more than 188 million Americans connected to the Internet [12], information security has become a top priority. Many applications — electronic mail, electronic banking, medical databases, and electronic commerce — require the exchange of private information. For example, when engaging in electronic commerce, customers provide credit card numbers when purchasing products. If the connection is not secure, an attacker can easily obtain this sensitive data. In order to implement a comprehensive security plan for a given network to guarantee the security of a connection, Confidentiality, Data Integrity, Authentication, and Non-Repudiation must be provided [37].

Cryptographic algorithms used to ensure confidentiality fall within one of two categories: private-key (also known as symmetric-key) and public-key. Symmetric-key algorithms use the same key for both encryption and decryption. Conversely, public-key algorithms use a public key for encryption and a private key for decryption. In a typical session, a public-key algorithm will be used for the exchange of a session key and to provide authenticity through digital signatures. The session key is then used in conjunction with a symmetric-key algorithm. Symmetric-key algorithms tend to be significantly faster than public-key algorithms and as a result are typically used in bulk data encryption [37]. The two types of symmetric-key algorithms are block ciphers and stream ciphers. Block ciphers operate on a block of data while stream ciphers encrypt individual bits or bytes. Block ciphers are typically used when performing bulk data encryption and the data transfer rate of the connection directly follows the throughput of the implemented algorithm.

High throughput encryption and decryption are becoming increasingly important in the area of high-speed networking. Many applications demand the creation of networks that are both private and secure while using public data transmission links. These systems, known as Virtual Private Networks (VPNs), can demand encryption throughputs at speeds exceeding Asynchronous Transfer Mode (ATM) rates of 622 million bits per second (Mbps). Increasingly, security standards and applications are defined to be algorithm independent. Although context switching between algorithms can be easily realized via software implementations, the task is significantly more difficult when using hardware implementations. The advantages of a software implementation include ease of use, ease of upgrade, ease of design, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [37]. Conversely, cryptographic algorithms that are implemented in hardware are by nature more physically secure as they cannot easily be read or modified by an outside attacker when the key is stored in special memory internal to the device. As a result, the attacker does not have easy access to the key storage area and cannot discover or alter its value in a straightforward manner [37]. Although traditional hardware implementations lack flexibility, configurable hardware devices offer a promising alternative for the implementation of processors via the use of IP cores in Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) technology. To illustrate, Altera Corporation offers IP core implementations of the Intel 8051 microcontroller and the Motorola 68000 processor in addition to their own Nios®-II embedded processor. Similarly, Xilinx Inc. offers IP core implementations of the PowerPC processor in addition to their own MicroBlaze™ and PicoBlaze™ embedded processors. ASIC and FPGA tech-
nologies provide the opportunity to augment the existing datapath of a processor implemented via an IP core to add acceleration modules supported through newly defined instruction set extensions targeting performance-critical functions [13], [19], [39]. Instruction set extensions result in significant performance improvements versus traditional software implementations with considerably reduced logic resource requirements versus hardware-only solutions [4], [23], [24], [31].

What follows is an overview on the various methods of speeding up symmetric-key algorithms in software. It will be shown that advances in technology have fueled trends towards increased reconfigurability in embedded systems, resulting in instruction set extensions becoming an attractive option for performance-critical applications. A discussion of the target processor, the LEON2 RISC processor, will be followed by an analysis of the performance bottlenecks commonly encountered in software implementations of the targeted cryptographic algorithms. The proposed instructions will then be presented, followed by a description of the modifications to the LEON2 processor and its associated development tools. Finally, data on the logic utilization of the additional hardware as well as throughput data for the target algorithms achieved using the instruction set extensions will be presented to demonstrate the effectiveness of this acceleration method.

2 Previous Work

Most traditional methods for improving the throughput of pure software implementations of symmetric-key algorithms fall into one of two categories. One option is to construct memory-based look-up tables where results of some of the basic operations of the algorithm have been pre-computed and stored. The substitution boxes, or S-Boxes, of the DES and AES algorithms are commonly stored in look-up tables in software implementations. Look-up tables may also be used to combine operations used in the DES and AES algorithms. AES requires several complicated mathematical operations that are time-consuming on general-purpose processors. Therefore in some implementations, large look-up tables, called T-tables, are employed that combine several of these complex operations into a single table access [34]. Look-up table based implementations are viable for systems with large memory spaces and low access times. However, area-constrained systems suffer large performance penalties using this methodology and are typically not implemented in this manner [33], [34].

Another method for speeding up software implementations of cryptographic algorithms involves taking advantage of mathematical or structural properties of the particular algorithm. The Initial and Final Permutations of the DES algorithm have regular structures that make it possible to execute a series of matrix transformations and XOR operations as demonstrated in [29]. This translates into a sequence of instructions that is much smaller than the traditional sequence required to perform the Initial and Final Permutations. In previous work on improving the performance of the AES algorithm on 32-bit systems, it has been shown that transforming a block of plaintext from a column-oriented matrix to a row-oriented matrix reduces the number of instructions required to complete the cipher due to more efficient implementation of the Galois Field matrix multiplication operations [1].

In order to extend the cryptographic capabilities of an embedded system without modifying the main processor, a co-processor solution can be adapted. When there is data that must be encrypted or decrypted through the chosen symmetric-key algorithm, the main processor sends the data and key material to the co-processor, and the co-processor performs the algorithm, sending the processed data back over the interface to the main processor. Most co-processor solutions have tended to combine a number of different algorithms to provide a multi-faceted security solution. Co-processors have achieved high throughput values compared to traditional software implementations and therefore are much more capable of meeting demands for speed-critical network communications. However, this type of solution requires considerable overhead in terms of hardware area, data transfer latency, and processor interfaces [4], [5], [11], [15], [16], [21], [28], [35], [36], [40].

Previous work on instruction set extensions for generalized permutations are useful for improving the performance of permutations used in the DES algorithm. Two new instructions for general and dynamically specified permutations are presented in [38]. The input and a string of configuration bits are specified in the source operands and the result is stored in the destination register. Permutations of n bits required \( \lceil \log_2(n) \rceil \) issues of the custom instructions as well as several loads of configuration bits into registers. The MOSES platform, based on the Xtensa T1040, is a RISC-like processor designed to be easily extended with additional hardware and supporting instructions. Throughput improvement factors of 31.0 for DES, 33.9 for Triple-DES, and 17.4 for AES were reported for this architecture [31]. Similarly, instruction set extensions that perform the mathematical operations in the AES rounds using custom functional units integrated into the targeted processor’s datapath were investigated in [33]. These extensions minimize the number of memory accesses, usually by combining the SubBytes and MixColumns transformations into one T-table look-up operation to speed up algorithm execution. While T-table performance is dependent upon available cache size, these extensions have yielded performance improvements of up to a factor of 3.68 versus AES implementations without the use of the instruction set extensions [16], [18], [31], [36].

Several software implementations of the IDEA algorithm take advantage of advanced processor architectures that employ instruction parallelism or functional units for multimedia support. A four-way parallel implementation on a 166 MHz Pentium MMX processor [25] achieved a throughput of approximately 72 Mbps. Throughput values ranging from 421 Mbps to 550 Mbps have been achieved on the Itanium platform running at 733 MHz [14]. The performance evalu-
ations reported in [10] include a comparison of IDEA software implementations on processors with various word sizes, clock frequencies, and cache sizes. Execution times for IDEA encryption ranged from 2.555 µs on the 8-bit 4 MHz Atmega 103 to 9 µs on the 64-bit 440 MHz UltraSparc2⃝ with instruction and data cache sizes of 16 Kbytes. Fast multiplication capability was shown to be a major factor in the performance of the IDEA algorithm.

Implementations of IDEA on reconfigurable computing platforms and systems with co-processors have shown improved performance. An implementation on an SRC-6E platform [27] achieved throughputs of approximately 590 Mbps for end-to-end software time for bulk data processing. Comparisons have been made between the performance of IDEA on Digital Signal Processors (DSPs), cryptographic co-processors, and hardware implementations on FPGAs in a hardware-software co-design system that makes use of encryption in a mobile device. Reported performance figures ranged from 32 Mbps on the DEC SA-110 and 53.1 Mbps on the TI TMS320C6x DSP chips, to 180 Mbps using the VINCI cryptographic co-processor, to 528 Mbps with an FPGA-based implementation [26].

3 The LEON2 Processor

The LEON2 processor is a RISC CPU produced by Gaisler Research that is implemented in VHDL and is fully synthesizable. The model is highly configurable and the source code is freely available under the GNU General Public License which enables modifications and enhancements to the architecture. The LEON2 processor is based on the Scalable Processor Architecture (SPARC⃝) and features a fully synchronous design with a single clock, use of multiplexers for loading of pipeline registers, separate combinational and sequential processes, and record types for interconnection of component I/O signals.

All SPARC⃝ V8 instructions are implemented in the LEON2 processor architecture. Instructions are grouped according to the values of the various fields in the instruction opcode. Most of the available features of the LEON2 processor can be enabled, disabled, or adjusted. This research employed a basic configuration with no FPU, PCI, Ethernet, co-processor interface, or hardware multiplier or divider. To extend the LEON2 architecture beyond the scope of the standard model, additional VHDL code is required. The specific files that must be modified depend on the functionality to be added, but if the instruction set is to be extended, the module containing the SPARC⃝ V8 opcode constants must be updated.

The LEON2 implementation can be targeted to any type of FPGA or ASIC technology. Pre-made packages enable use of technology-specific cells to directly instantiate or automatically infer the register files, caches, PCI FIFOs, and I/O pads. Functional verification and performance evaluation of programs built for the LEON2 architecture can be performed with the provided generic test bench. The VHDL source for the test bench is located in the /tbench/ sub-folder of the LEON2 directory structure. Software code is placed in the /source/ sub-folder in a format readable by the test bench VHDL code. The software can then be read and executed by the test bench. In order to facilitate the development of programs targeting the LEON2 processor, Gaisler Research has provided a series of compilers and simulators that may be chosen depending on the software environment. For standalone applications, the Bare C Compiler, based on the GNU Compiler Collection and GNU binutils, is recommended.

4 Evaluation of the Target Algorithms

Software implementations of DES tend to be significantly slower than hardware implementations. Bit-level manipulations such as those contained in the permutation, expansion, permuted choice, and Cyclic Left/Right Shift units do not map well to general-purpose processors. General-purpose processor instruction sets operate on multiple bits at a time based on the processor word size. Moreover, the DES S-Boxes do not use memory in an efficient manner. Software look-up tables would appear to be the obvious implementation choice for the DES S-Boxes. However, the DES S-Boxes have 6-bit addresses and 4-bit output bits while most memories associated with general-purpose processors use byte addressing with either 8-bit or 32-bit output data. As a result, many software implementations of DES exhibit throughputs that are at least a full order of magnitude slower than hardware implementations.

Even the best software implementations are only capable of throughputs in the range of 100–200 Mbps. Most of these implementations recommend storing the 32-bit left and right halves of the data stream as a 48-bit padded word within a 64-bit processor word and implementing the permutations and S-Boxes as precomputed look-up tables. Additionally, look-up table implementation for the S-Boxes is most effective when the size of the look-up tables is minimized, guaranteeing that the data will fit entirely in on-chip cache. Size minimization of the S-Box look-up tables is achieved by implementing each S-Box in its own look-up table. Finally, one key software optimization is the unrolling of software loops to increase performance. Even when software loops are too cumbersome to unroll, using loop counters that decrement to zero in place of loop counters that increment to a terminal count are shown to greatly increase the performance of software implementations of the DES algorithm. However, the unrolling of software loops must be done with great care such that the total data storage space does not exceed the size of the on-chip cache to avoid extreme performance degradation [3], [17], [30].

In terms of the core operations of IDEA, bit-wise XOR and addition are easily implemented with one instruction each in software. For reduction modulo 216, a processor such as the LEON2 that only performs arithmetic on 32-bit registers requires an additional logic instruction to mask out the bits that may overflow into the sixteen most significant bits of the destination register. However, the major performance bottleneck for a software implementation of IDEA
is multiplication modulo $2^{16} + 1$. Multiplication may require several clock cycles to complete (especially those without hardware multipliers), and the modular reduction, commonly implemented using the Low-High Lemma [22], requires additional execution time.

AES software performance bottlenecks typically occur in the SubBytes and MixColumns transformations, one or both of which are usually implemented via 8-bit to 8-bit look-up tables. Often most of the AES round transformations — SubBytes, ShiftRows, and MixColumns — are combined into large look-up tables termed T-tables. Such implementations require up to three T-tables whose size may be either 1 Kbytes or 4 Kbytes where the smaller tables require performing an additional rotation operation. The goal of the T-tables is to avoid performing the MixColumns and InvMixColumns transformations as these operations perform Galois Field field constant multiplication, an operation which maps poorly to general-purpose processors. However, the use of T-tables has a number of disadvantages. T-tables significantly increase code size, their performance is dependent on the memory system architecture as well as cache size, and their use causes key expansion for AES decryption to become significantly more complex. As an alternative to the use of T-tables, it is also feasible to have the processor perform all of the AES round transformations. Row-based implementations have been demonstrated to allow for greater efficiency in the implementation of the MixColumns and InvMixColumns transformations versus column-based implementations. However, the SubBytes transformation still remains as a bottleneck, requiring separate 256-byte look-up tables for encryption and decryption [1], [6], [33], [34].

5 Proposed Instruction Set Extensions

All of the proposed instruction set extensions comply with the SPARC® V8 instruction model using the Format 3 structure. All instructions that write to a register execute in one clock cycle except for the mmul16 instruction which requires two clock cycles. For those instructions that store data directly into registers contained in the hardware added to the datapath, the data is available at the start of the next cycle, after instruction execution has completed. Building the set of development tools from the source files is necessary when extending the instruction set of the LEON2 processor. The source code archive for BCC v1.0.29c includes specific modifications made to two different versions of GNU binutils to support the LEON2 processor. This research employs the v2.16.1 binutil. The file /opcodes/sparc-opc.c was edited to include op3 values for the instruction set extensions.

All added hardware modules are coded in VHDL and all inputs and outputs are read from and written to the LEON2 pipelined integer unit (IU) register file. None of the added logic circuits rely on external memory for their functionality. A new module was included with the VHDL source for the LEON2 processor architecture to provide an easy way to select specific extensions to be included in the architecture. For the AES S-Box extensions, the available options are no S-Boxes, one S-Box, and four S-Boxes. For all other types of extensions, setting the configuration variables to true includes extensions into the architecture while a value of false excludes them from the architecture. All of the added functional units that support the proposed instruction set extensions have been included in the IU. Component declarations were added for each of the added hardware units and instantiated as part of the arithmetic logic unit. The decode stage of the IU pipeline sets flags for the instruction set extensions and generates source register and immediate data. On the next clock cycle, the execute stage passes the input operands to the appropriate functional unit based on the instruction flag set. The result is then read from the functional unit when the instruction specifies a destination register to receive an output.

The SPARC® assembly instruction opcodes are defined in the /leon/sparcv8.vhd module. The code added to this module include the values for the op3 field of the new instructions. Provided in the LEON2 base package is a generic test bench with disassembly support. During functional simulation, assembly instructions are printed out to the simulation software’s console window as they are executed. The module /leon/debug.vhd contains the functions that handle display of the instruction strings, and disassembly support for the new instructions was added. When the op and op3 instructions match those of one of the instruction set extensions, the instruction name is printed followed by any applicable source/destination registers or immediate data for that instruction.

5.1 DES and Triple-DES

The desipl and desippr instructions produce the left and right halves of the Initial Permutation, respectively. Similarly, the desipl and desippr instructions produce the left and right halves of the Final Permutation, respectively. The left half of the input block must be located in the rs1 register and the right half must be located in the rs2 register. Inclusion of these instructions allows the Initial and Final Permutations to be completed in two instructions each. Traditional software implementations require a series of bit mask setup, shift, logical AND, and logical OR operations for each bit for a total of 256 instructions [38]. Even the improved permutation algorithm used in [20] requires 44 instructions to complete on a SPARC® V8 processor such as the LEON2.

The desdir instruction sets up the Key Generator to output round keys in either encryption or decryption order. This instruction also resets the round counter of the Key Generator according to the chosen direction to ensure that output of the round keys may be immediately carried out in the proper order. It is not necessary to re-load the master key after this instruction is executed. The desdir instruction is used in conjunction with the deskey and desf instructions. The deskey instruction loads the 64-bit master key. The left half of the master key must be contained in the rs1 register, and the right half in the rs2 register. The desf instruction takes the right half of a round output block stored in the rs1 register and
stores the output of the \( f \)-function into the \( rd \) register. The
round key is not specified here since the round key output of the
Key Generator is hard-wired to the \( f \)-Function unit’s
round key input. After completion of this instruction, the Key
Generator is signaled to generate the key for the next round.
Due to the logic of the Key Generator, the \( desf \) instruction
may not be followed by another \( desf \) instruction. However,
this is not expected to cause a performance bottleneck due to
the additional instruction required for swapping the values of the
left and right halves of the round input block. Implementation
of the \( desdir, deskey, \) and \( desf \) instructions removes the
need for storage of the sixteen round keys and S-Boxes
in memory. All round keys are generated on-the-fly in the
hardware added to the datapath. An implementation of the
DES algorithm using these instructions requires two
instructions for key scheduling and four instructions for each
of the sixteen rounds — one \( desf \), one bit-wise XOR, and two reg-
ister data transfers for swapping the left and right halves of the
round function output.

The Permutation Unit implements the Initial Permutation
and Final Permutation. The inputs are loaded from the source
registers specified in the permutation instruction. The inputs
are passed through two stages of 2-to-1 multiplexers. The
first stage selects the rearranged bits of either the Initial Per-
mutation or the Final Permutation output. The output of the
selected permutation is represented by the pair of 32-bit vec-
tors. The second stage of multiplexers sets the final output
to either the left half or the right half of the output. The Key
Generator unit was designed to work in conjunction with the
\( f \)-Function unit. All registers are sensitive to the rising edge
of the clock. The 64 key bits, which are loaded from the source
registers specified in the \( deskey \) instruction, are rear-
anged according to the \( PC-I \) mapping. A 64-bit master key
is loaded by issuing the \( deskey \) instruction. When this
instruction is in the execute stage, the key bits are loaded into
the \( C0 \) and \( D0 \) registers. When performing encryption, the
\( C \) and \( D \) registers are loaded with the values of \( C0 \) and \( D0 \)
rotated left by 1 bit since the first values of \( C_i \) and \( D_i \) used
in the encryption key schedule are \( C_1 \) and \( D_1 \). When per-
forming decryption, the \( C \) and \( D \) registers are loaded with
the exact values of \( C0 \) and \( D0 \) because the final \( C_i \) and \( D_i \)
values used in the encryption key schedule are the first values
used in the decryption key schedule. The \( f \)-Function unit’s
32-bit input port for \( R_{i-1} \) is loaded from the source register
operand specified in the \( desf \) instruction. The 48-bit input
port for the current round key is generated by the Key Gen-
erator unit. Expansion and permutation blocks are imple-
mented by rerouting the inputs and the S-Boxes are defined as
logic-based mappings. The output of the \( f \)-Function unit is
stored in the destination register specified in the \( rd \) field of the
\( desf \) instruction.

5.2 IDEA

The \( mmul16 \) instruction computes
\( rs1 \cdot rs2 \text{ mod } 2^{16} + 1 \)
and stores the product in the \( rd \) register. Both source
operands must be in the lower sixteen bits of their respective
registers. The 16-bit product is stored in the lower sixteen
bits of the \( rd \) register. The \( Modal2^{16} + 1 \) Multiplication
unit is designed based on the adder-based modular multiplier
in [2]. The \( Modal2^{16} + 1 \) Multiplication unit first gener-
ates partial products \( \text{reduced modulo } 2^{16} + 1 \) as described
in Zimmerman’s investigation of efficient architectures for
arithmetic \( \text{modulo } 2^n \pm 1 \) [41]. Each partial product is
determined by the formula:

\[
PP_i = x_i \cdot y_{15-i} \cdot \ldots \cdot y_1y_{15-i} + \bar{x}_i \cdot 0 \cdot 01 \cdot 01 \ldots (1)
\]

where the vector \( 0 \cdot 01 \cdot 01 \ldots \) contains \( 16 - i \) zeros and \( i \)
one. In order to handle cases where \( x = 0 \) or \( y = 0 \), a
correction term \( k \) is defined:

\[
k = \begin{cases} 
2 & \text{if } x = 0 \text{ and } y = 0, \\
\bar{x} + 3 & \text{if } x = 0 \text{ and } y \neq 0, \\
\bar{y} + 3 & \text{if } x \neq 0 \text{ and } y = 0, \\
1 & \text{if } x \neq 0 \text{ and } y \neq 0.
\end{cases} (2)
\]

An intermediate sum \( s = k + \sum_{i=0}^{15} PP_i \) is then computed.
The final step is a reduction of \( s \text{ modulo } 2^{16} + 1 \). Defining
\( s_L \) to be the sixteen least significant bits and \( s_H \) to be the
remaining high-order bits of \( s \), the result of the multiplication
is \( s \text{ mod } (2^{16} + 1) = s_L + 2^{16}s_H \). This result may be
reduced such that \( s \text{ mod } (2^{16} + 1) \equiv s_L + s_H + 2 \).
Using the Low-High Lemma for reduction \( \text{modulo } 2^{16} + 1 \)
assuming that \( t = s_L + s_H + 1 \) [22]:

\[
s \text{ mod } (2^{16} + 1) = \begin{cases} 
(s_L + s_H + 2) \text{ mod } 2^{16} & \text{if } t < 2^{16} \\
(s_L + s_H + 1) \text{ mod } 2^{16} & \text{if } t \geq 2^{16}. 
\end{cases} (3)
\]

Additions are implemented in the \( Modal2^{16} + 1 \) Mul-
tiplication unit with a carry-propagate adder tree. A generic
model is specified in a VHDL source file separate from the
multiplier source file. The generic component design allows
for adjustable width of the inputs.

5.3 AES

The \( aessb, aessbs, aessb4, \) and \( aessbs4s \) instructions per-
form the SubBytes and InvSubBytes operations on either
one or four of the bytes in the \( rs1 \) register. The instruction
operands determine if the SubBytes operation or the InvSub-
Bytes operation is to be performed. In the case of the single
S-Box instructions, the operand also specifies which of the
four bytes in the \( rs1 \) register is to be operated upon. The
\( aessbs \) instruction provides the additional functionality of
allowing the user to specify the destination byte in the \( rd \)
register while the \( aessbs4s \) instruction allows the user to spec-
ify the number of bytes to left-shift the 4-byte result prior to
storage in the destination register. The \( gfmkld \) instruction is
used to load one of the sixteen constants into the \( 4 \times 4 \) con-
stant matrix of the Galois Field fixed field constant matrix
multiplier. The constants are loaded row by row, beginning
with row zero and proceeding in order to row three. Each
row is loaded beginning with the constant from column zero.
and proceeding in order to the constant in column three. Due to the logic that has been added to the multiplier for inclusion into the LEON2 processor datapath, instances of the gfinkld instruction may not be issued consecutively. The gffmmul instruction performs the Galois Field fixed field constant matrix multiplication on the input in the rs1 register and stores the result in the rd register.

The SubBytes and InvSubBytes S-Boxes are implemented as logic-based mappings in hardware. The dir signal selects the S-Box output. The Galois Field Fixed Field Constant Multiplier unit performs the MixColumns or InvMixColumns operation. The architecture of this multiplier is described in [9], [8]. The MixColumns and InvMixColumns operations are a matrix multiplication over the Galois Field GF(2^8) on each column of the state by a 4 × 4 fixed field constant matrix. This means that a total of sixteen multiplications in the Galois Field GF(2^8) must be performed to complete the entire operation. Each matrix must then be reduced modulo \( m(x) = x^8 + x^4 + x^3 + x + 1 \), the irreducible polynomial specified for AES. To accomplish the multiplication and modular reduction simultaneously, the operation can be represented as an 8 × 8 matrix multiplication over the Galois Field GF(2).

The constants in the inner matrix are determined by the constant factor in the multiplication and the polynomial \( m(x) \). Consider the representative Galois Field GF(2^8), used by AES in the MixColumns and InvMixColumns transformations. Note that \([A_3:A_0]\) are the input bytes and \([B_3:B_0]\) are the output bytes [7]:

\[
\begin{bmatrix}
B_0 \\
B_1 \\
B_2 \\
B_3
\end{bmatrix} =
\begin{bmatrix}
K_{00} & K_{01} & K_{02} & K_{03} \\
K_{10} & K_{11} & K_{12} & K_{13} \\
K_{20} & K_{21} & K_{22} & K_{23} \\
K_{30} & K_{31} & K_{32} & K_{33}
\end{bmatrix}
\begin{bmatrix}
A_0 \\
A_1 \\
A_2 \\
A_3
\end{bmatrix}
\]

The core operation in this fixed field multiplication is an 8-bit inner product that must be performed sixteen times, four per row. The four inner products of each row are then combined via a bit-wise XOR operation to form the final output word. For a known primitive polynomial \( p(x), k(x) \) (representing the 8-bit constant), and a generic input \( a(x) \), the resultant polynomial equation takes the form \( b(x) = a(x) \times k(x) \mod p(x) \) where each coefficient of \( b(x) \) is a function of \( a(x) \). This results in an 8-bit × 8-bit matrix

representing the coefficients of \( b(x) \) in terms of \( a(x) \) [7]. An 8-bit × 8-bit matrix must be generated for each \( K_{xy} \), resulting in a total of sixteen matrices. Note that this analysis holds true for Galois Fields other than GF(2^8) with corresponding adjustments to the mapping matrix.

6 Analysis of Results

Functional verification was performed using twelve test vectors for each algorithm. Performance testing measured the execution cycles required to perform one iteration of the target algorithm. Each algorithm was tested in both non-feedback (Electronic Code Book) and feedback (Cipher Block Chaining) modes of operation for both encryption and decryption. The LEON2 processor implementation was synthesized targeting the Xilinx Virtex-4 XC4VLX25 FPGA using the Xilinx ISE 8.1i tools.

6.1 Code Size

The following tables present executable code sizes for implementations of the target algorithms with different combinations of instruction set extensions. For the DES and Triple-DES algorithms, the permutation instructions decrease the total code size by up to a factor of 1.3 but have no effect on the key schedule as permutations are not used in the computation of the round keys. Instructions supporting the round key generation and round function have a much more pronounced impact on code size. When these instructions are used, all of the lengthy permutation routines and memory-based S-Boxes are no longer needed. Encryption and decryption code size is reduced by up to a factor of 4.0 in the case of DES and a factor of 3.5 in the case of Triple-DES with the use of these instructions alone. Key scheduling is handled via two instructions, making the respective code size a small percentage of the remaining program code needed to implement encryption and decryption. When all of the instruction set extensions are implemented, DES code size is reduced by factors of up to 31.2 in non-feedback mode and 21.1 in feedback mode and Triple-DES code size is reduced by factors of up to 15.9 in non-feedback mode and 13.0 in feedback mode.

Note that the decryption key schedule code size data for IDEA includes that of the encryption key schedule. This is because the decryption keys are determined from the encryption keys. The key schedule for decryption sees a slight decrease in code size with the use of the mmul16 instruction because of the need to compute multiplicative inverses mod \( 2^{16} \). The addition of the mmul16 instruction significantly decreases the code size of encryption and decryption by factors of up to 2.8 in non-feedback mode and 2.4 in feedback mode. Note that due to the absence of a hardware multiplier in the LEON2 integer unit, the multiplication is performed by a library function.

The AES instruction set extensions aessbs and aessbs4 yielded either equivalent or reduced code size for both column-oriented and row-oriented implementations versus the aessb and aessb4 instruction set extensions. Using one S-Box led to the largest reduction in code size for column-oriented implementations by up to a factor of 1.8 for encryption and 1.6 for decryption. In the case of the row-oriented implementations, the use of four S-Boxes led to the largest reduction in code size by up to a factor of 2.6 for encryption and 2.2 for decryption. However, the use of one S-Box results in significantly reduced key scheduling code size for row-oriented implementations.

When only the gffmmul instruction is incorporated, code size decreases by up to a factor of 1.3 for encryption and 1.8 for decryption in the column-oriented implementations. The original code used to implement the InvMixColumns operation for decryption requires many more operations than the MixColumns operation used by encryption. Only four
instances of the \texttt{gfmmul} instruction are needed to perform both operations – one for each column of the AES state. The use of the \texttt{gfmmul} instruction results in up to a factor of 1.1 increase in code size for encryption and 1.1 decrease in code size for decryption in the row-oriented implementations. This occurs because the MixColumns and InvMixColumns operations operate on the columns of the AES state, requiring additional instructions to rearrange the bytes prior to being processed by the \texttt{gfmmul} instruction in the row-oriented implementations. The effect of the byte rearrangement is that column-oriented implementations require significantly smaller code space than the row-oriented implementations when all of the instruction set extensions are implemented. For column-oriented implementations, code size is reduced by up to a factor of 3.6 for encryption and 4.8 for decryption using one S-Box while for row-oriented implementations, code size is reduced by up to a factor of 2.1 for encryption and 2.3 for decryption using four S-Boxes.

The following tables present the number of clock cycles required to complete a full iteration of each algorithm. For the DES and Triple-DES algorithms, the permutation instructions alone have virtually no impact on the execution cycles for both DES and Triple-DES. However, the instructions supporting round key generation and the round function have a significant impact on execution cycles, yielding speedups for DES by a factor of up to 17.3 in non-feedback mode and 16.5 in feedback mode and speedups for Triple-DES by a factor of up to 17.3 in non-feedback mode and 17.0 in feedback mode. Implementation of all of the instruction set extensions yields speedups for DES by a factor of up to 32.6 in non-feedback mode and 30.2 in feedback mode and speedups for Triple-DES by a factor of up to 32.8 in non-feedback mode and 32.0 in feedback mode.

The use of the \texttt{mmul16} instruction significantly decreases the IDEA execution cycle count. While key scheduling for encryption is unaffected, the use of the \texttt{mmul16} instruction results in a decryption key scheduling speedup by a factor of 7.8. Similarly, the use of the \texttt{mmul16} instruction results in speedups of IDEA by up to a factor of 7.7 in non-feedback mode and 7.1 in feedback mode.

The AES instruction set extensions \texttt{aessbs} and \texttt{aessb4s} yielded either equivalent or reduced execution cycles for both column-oriented and row-oriented implementations versus the \texttt{aessb} and \texttt{aessb4} instruction set extensions. Using one S-Box led to the largest speedup for column-oriented implementations by up to a factor of 1.4 for encryption and 1.2 for decryption. Using four S-Boxes led to the largest speedup for row-oriented implementations by up to a factor of 1.9 for encryption and 1.6 for decryption. However, the use of one S-Box reduced the key scheduling execution cycles for row-oriented implementations.

Incorporating only the \texttt{gfmmul} instruction yields speedups of up to a factor of 1.8 for encryption and 3.0 for decryption in the column-oriented implementations. The original code used to implement the InvMixColumns operation for decryption requires many more operations, and thus cycles, versus the MixColumns operation used by encryption. The use of the \texttt{gfmmul} instruction results in up to a factor of 1.1 decrease in performance for encryption and 1.04 speedup for decryption in the row-oriented implementations. This occurs because the MixColumns and InvMixColumns operations operate on the columns of the AES state, requiring additional cycles to rearrange the bytes prior to being processed by the \texttt{gfmmul} instruction in the row-oriented implementations. The effects of the byte rearrangement is such that the speedups of the column-oriented implementations are significantly larger than the speedups of the row-oriented implementations when all of the instruction set extensions are implemented and row-oriented implementations perform better when the \texttt{gfmmul} instruction is combined with one S-Box instead of four S-Boxes. Column-oriented implementations yield speedups by a factor of 4.0 for encryption and 6.6 for decryption using one S-Box while row-oriented implementations yield speedups by a factor of 1.4 for encryption and 1.7 for decryption.

### Table 1. DES and Triple-DES code size (bytes)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Base Code</th>
<th>Permute ISEs</th>
<th>Key and f-Function ISEs</th>
<th>All ISEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES Key Sched</td>
<td>1524</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>DES Enc CBC</td>
<td>1524</td>
<td>237</td>
<td>170</td>
<td>197</td>
</tr>
<tr>
<td>DES Dec CBC</td>
<td>1524</td>
<td>237</td>
<td>170</td>
<td>197</td>
</tr>
<tr>
<td>Triple-DES Enc</td>
<td>1524</td>
<td>1194</td>
<td>1069</td>
<td>1200</td>
</tr>
<tr>
<td>Triple-DES Dec</td>
<td>1524</td>
<td>1194</td>
<td>1069</td>
<td>1200</td>
</tr>
<tr>
<td>Triple-DES Enc</td>
<td>2096</td>
<td>1656</td>
<td>1632</td>
<td>1672</td>
</tr>
<tr>
<td>Triple-DES Dec</td>
<td>2096</td>
<td>1656</td>
<td>1632</td>
<td>1672</td>
</tr>
<tr>
<td>Triple-DES Enc</td>
<td>2128</td>
<td>1656</td>
<td>1632</td>
<td>1672</td>
</tr>
<tr>
<td>Triple-DES Dec</td>
<td>2128</td>
<td>1656</td>
<td>1632</td>
<td>1672</td>
</tr>
</tbody>
</table>

### Table 2. IDEA code size (bytes)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Base Code</th>
<th>Modulo $2^{16} + 1$ Multiplication ISE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc Key Sched</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>Dec Key Sched</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>Enc CBC</td>
<td>796</td>
<td>796</td>
</tr>
<tr>
<td>Dec CBC</td>
<td>796</td>
<td>796</td>
</tr>
</tbody>
</table>

### Table 3. InvMixColumns code size (bytes)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Base Code</th>
<th>InvMixColumns ISE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES Enc CBC</td>
<td>576</td>
<td>276</td>
</tr>
<tr>
<td>DES Dec CBC</td>
<td>576</td>
<td>276</td>
</tr>
<tr>
<td>Triple-DES Enc</td>
<td>576</td>
<td>276</td>
</tr>
<tr>
<td>Triple-DES Dec</td>
<td>576</td>
<td>276</td>
</tr>
</tbody>
</table>

### Table 4. DES and Triple-DES execution cycles

<table>
<thead>
<tr>
<th>Operation</th>
<th>Base Code</th>
<th>Permute ISEs</th>
<th>Key and f-Function ISEs</th>
<th>All ISEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES Key Sched</td>
<td>2011</td>
<td>2011</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>DES Enc CBC</td>
<td>2011</td>
<td>2011</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>DES Dec CBC</td>
<td>2011</td>
<td>2011</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Triple-DES Enc</td>
<td>8730</td>
<td>8730</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Triple-DES Dec</td>
<td>8730</td>
<td>8730</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Triple-DES Enc</td>
<td>12648</td>
<td>12648</td>
<td>752</td>
<td>752</td>
</tr>
<tr>
<td>Triple-DES Dec</td>
<td>12648</td>
<td>12648</td>
<td>752</td>
<td>752</td>
</tr>
</tbody>
</table>

Table 7 details execution cycle counts for some combinations of extensions are compared with results published by the ISEC project [33]. Their work includes the use of logic-based mappings as a choice of hardware implementation for the S-Boxes. The \texttt{abox} and \texttt{abox4} instructions from the ISEC project are equivalent to the \texttt{aessbs} and \texttt{aessb4} i-
3.6 Algorithm Throughput Comparisons

Table 8 shows the component usage of the Xilinx Virtex-4 XC4VLX25 FPGA by each added functional unit and the total utilizations of the LEON2 processor with combinations of instruction set extensions for each targeted algorithm. Due to the number of storage bits needed for matrix configuration and combinational logic used to compute the matrix product, the Galois Field matrix multiplier is the largest of the added hardware units. The implementation with all of the proposed instruction set extensions leads to a total area increase over the baseline configuration by approximately 63%. The modulo $2^{16} + 1$ multiplier for IDEA was the single largest contributor to decreases in the implementation’s maximum operating frequency. A purely combinational design of the multiplier had large path delays due to the carry-propagate adder structure, yielding a maximum operating frequency of only 72 MHz. The multiplier was modified to create a two cycle instruction implementation to decrease the effect on the processor clock frequency. With all extensions implemented, the LEON2 processor implementation resulted in a clock frequency of approximately 117 MHz, a 10% decrease compared to the baseline implementation.
target algorithms using the proposed instruction set extensions when performing encryption in non-feedback mode. The results are analyzed with respect to the hardware resources required for each implementation to determine the hardware cost associated with improving the execution time of the targeted algorithms using the proposed instruction set extensions. Throughput data is presented in terms of Megabits per second (Mbps), hardware usage is presented in terms of FPGA configurable logic block (CLB) slices, and throughput/area (T/A) ratios are presented in bits per second per slice.

The data shows that the instruction set extensions yielded the best improvement for DES in terms of T/A ratio, with the ratio increasing by a factor of 25.6. This increase is a result of nearly all of the DES functionality being off-loaded to the added hardware. As expected, T/A ratios for Triple-DES were approximately 1/4 of the corresponding values for DES because both algorithms require the same hardware in order to be implemented using the instruction set extensions. The T/A ratio for Triple-DES increased by a factor of 25.8 when instruction set extensions are used, matching the increase evidenced by DES. In the case of IDEA, the instruction set extensions had a significant impact upon performance at a reasonable hardware cost, with the measured T/A ratio increasing by a factor of 5.9. For AES, the instruction set extensions used in conjunction with the column-oriented implementations yield larger increases in throughput versus the instruction set extensions used in conjunction with the row-oriented implementations. As a result, the column-oriented implementations yielded a greater increase in T/A ratio and are superior to the row-oriented implementations when using the proposed instruction set extensions.

7 Conclusions

Instruction set extensions for improving software implementations of symmetric-key algorithms have been proposed. Existing literature on the subject of enhancing the performance of symmetric-key algorithms was discussed, followed by detailed descriptions of the targeted processor and the targeted cryptographic algorithms. Descriptions of the custom instructions were given along with the functional units that implement the underlying logical and arithmetic operations. The results show that the proposed instructions have a significant positive effect on program code size for all of the targeted algorithms, shrinking the number of code bytes by up to a factor of 31.2. Execution time for all algorithms was also improved, with demonstrated speedups by factors of up to 32.8. The instruction set extensions required only a 63% increase in the logic utilization of the LEON2 processor on the chosen FPGA device while decreasing the maximum clock frequency by approximately 10%. All of the targeted algorithms evidenced an increase in T/A ratio, increasing by a factor of up to 25.8. Finally, it was demonstrated that column-oriented implementations of AES are superior to row-oriented implementations for all evaluation metrics when using the proposed instruction set extensions.

8 Acknowledgement

We would like to thank Stefan Tillich and Johann Großchädler from the Graz University of Technology for their useful discussions regarding their instruction set extensions targeting the LEON2 processor.

References

Table 9. Throughput to area ratios

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Throughput</th>
<th>Core Code</th>
<th>T/A Ratio</th>
<th>With IDEA</th>
<th>Throughput</th>
<th>Core Code</th>
<th>T/A Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triple-DES</td>
<td>0.64</td>
<td>4563</td>
<td>200.11</td>
<td>55.43</td>
<td>10.55</td>
<td>55.43</td>
<td>1.92</td>
</tr>
<tr>
<td>IDEA</td>
<td>0.72</td>
<td>4563</td>
<td>258.34</td>
<td>55.43</td>
<td>10.55</td>
<td>55.43</td>
<td>2.07</td>
</tr>
<tr>
<td>AES-Columns</td>
<td>1.24</td>
<td>4563</td>
<td>707.57</td>
<td>42.89</td>
<td>5411</td>
<td>5411</td>
<td>20.07</td>
</tr>
<tr>
<td>AES-Rows</td>
<td>1.24</td>
<td>4563</td>
<td>707.57</td>
<td>42.89</td>
<td>5411</td>
<td>5411</td>
<td>20.07</td>
</tr>
</tbody>
</table>


