A Non-Linear/Linear Instruction Set Extension for Lightweight Ciphers

Susanne Engels, Elif Bilge Kavun, Christof Paar and Tolga Yalcın
Horst Görtz Institute for IT-Security
Ruhr-Universität Bochum
Germany
Email: susanne.engels, elif.kavun, christof.paar, tolga.yalcin@rub.de

Hristina Mihajloska
Faculty of Computer Science and Engineering
Ss Cyril and Methodius University
Skopje, Macedonia
Email: hristina.mihajloska@finki.ukim.mk

Abstract—Modern cryptography today is substantially involved with securing lightweight (and pervasive) devices. For this purpose, several lightweight cryptographic algorithms have already been proposed. Up to now, the literature has focused on hardware-efficiency while lightweight with respect to software has barely been addressed. However, a large percentage of lightweight ciphers will be implemented on embedded CPUs – without support for cryptographic operations. In parallel, many lightweight ciphers are based on operations which are hardware-friendly but quite costly in software. For instance, bit permutations that accrue essentially no costs in hardware require a non-trivial number of CPU cycles and/or lookup tables in software. Similarly, S-Boxes often require relatively large lookup tables in software. In this work, we try to address the open question of efficient cipher implementations on small CPUs by introducing a non-linear/linear instruction set extension, to which we refer to as NLU, capable of implementing non-linear operations expressed in their algebraic normal form (ANF) and linear operations expressed in binary “matrix multiply-and-add” form. The proposed NLU is targeted for embedded microcontrollers and it is therefore 8-bit wide. However, its modular architecture allows it to be used in 16, 32, 64 and even 4-bit CPUs. We furthermore present examples of the use of NLU in the implementation of standard cryptographic algorithms in order to demonstrate its coding advantage.

Keywords—lightweight ciphers; instruction set extension; non-linear operation; linear operation; algebraic normal form

I. INTRODUCTION

Modern cryptography is a central tool for securing pervasive devices, ranging from smart phones over contactless smart cards to RFID tags. Security is needed for applications such as user and device identification, remote configuration, software update, function activation, IP protection, and secure communication. Motivated by the need to realize such services on cost and energy constrained platforms, numerous symmetric lightweight ciphers have been proposed over the last few years. Suggested algorithms include the block ciphers CLEFIA [1], Hight [2], KATAN, KTANTAN [3], Klein [4], mCrypton [5], LED [6], Piccolo [7], and PRESENT [8], and the stream ciphers Grain [9], Mickey [10], and Trivium [11]. Noticeably, there is also industry demand for lightweight ciphers, as can be seen by the recent adoption of CLEFIA and PRESENT in the ISO/IEC Standard 29192-2 [12]. Somewhat surprisingly, the focus in the literature has almost been entirely on hardware-efficiency, while lightweight with respect to software has barely been addressed. This is in contrast to the fact that the vast majority of pervasive devices is equipped with (small) embedded CPUs as opposed to dedicated ASICs.

At the same time, many lightweight ciphers are based on operations which are hardware-friendly but quite costly in software. For instance, bit permutations with zero costs in hardware require several number of CPU cycles and/or lookup tables in software, whereas S-Boxes of substitution layer often require relatively large lookup tables in software, e.g. T-tables in the case of Advanced Encryption Standard (AES) [13], [14]. This development has somewhat lead to the paradox situation that many of the “lightweight” ciphers are not particularly lightweight with respect to software implementation. We would like to note at this point that even if a cipher is implemented in hardware on an end-device, e.g., on an RFID tag, it is not uncommon that a corresponding embedded software implementation is needed on the other end of the communication link, e.g., within an RFID reader. In this work, we try to improve the situation of efficiently realizing lightweight ciphers on embedded CPUs.

Our approach is based on a non-linear/linear operation unit, denoted in the following as NLU. The NLU is capable of computing non-linear operations expressed in their algebraic normal form (ANF) and linear operations expressed in binary “matrix multiply-and-add” form, both of which are very common in modern (including lightweight) ciphers. The NLU discussed in this contribution targets small embedded microcontrollers and is, hence, 8 bits wide. (Note that 8-bit CPUs still hold a huge share of the worldwide microprocessor market.) The modular architecture of the NLU allows its adoption to 16, 32, 64 and even 4-bit CPUs, giving the flexibility to be used with different microcontrollers and microprocessors.

For demonstration purposes, we show that NLU can result in time-area product reductions between 20-70% for the widely used and standardized ciphers PRESENT, CLEFIA, SERPENT [15], and AES. Note that SERPENT and AES are not targeted to be lightweight, but they are widely used in many embedded applications.

The remainder of our paper is structured as follows: Sec-
tion II gives an overall view of our instruction set extension model. In Section III, the unified hardware structure of non-
linear and linear units is explained in detail. Following that, the application of NLU on PRESENT, CLEFIA, SERPENT, and AES is given in Section IV. Section V discusses the achieved speed-up in software using NLU instructions and we finally conclude the paper in Section VI by also stating our future directions.

II. INSTRUCTION SET EXTENSION MODEL

Performance enhancement of software implementations of cryptographic algorithms is not entirely a new idea. There have already been several studies on this subject. However, most of the previous works rely on implementation of cipher specific instructions, which – in hardware – maps to plugging the specific cipher as a coprocessor into the main module. This approach, while providing the best performance improvement in terms of execution time, results in a considerable increase in microprocessor area. For example, in [16], the authors state the area increase in the total area of the processor to be less than 65%. Furthermore, such an approach limits the performance improvement(s) to a specific cipher. Other works try to address this issue by introducing relatively complex instructions utilization [17], [18], [19], [20].

However, none of the previous works targets lightweight cryptography, which requires realization of (lightweight) ciphers on resource-constrained devices. The resource limitation on such devices is two-fold: In terms of hardware, it should occupy the smallest possible silicon area; in terms of software, it should be realized with minimal lines of code. These two requirements usually contradict with each other. The smallest possible silicon area is achieved by implementing all cryptographic operations in software on a small-footprint embedded processor; while the least possible number of code lines can be achieved on dedicated hardware – either in the form of specific crypto-units, or coprocessors attached to an embedded processor. Unfortunately, smallest silicon area results in the maximum number of code lines and longest execution time, while the shortest code results in the highest silicon area.

In our case, we bridge the gap between these two approaches, which also comes with a compromise: We can neither reach the minimal silicon area nor the shortest code. Instead, we try to find a common ground by introducing minimum additional silicon area that results in a code size reduction (about 20-50%, which can be considered as “high” for most applications). We also target a generic solution that would apply to any (lightweight) cipher.

We take the standard arithmetic logic units (ALUs), which exist even in the simplest microprocessors, as our base approach. In fact, this is also the basis for floating point units (FPUs), where floating point instructions are realized on specific hardware modules. These modules are physically connected to the main databus as the ALU and logically connected to the instruction decoder, which forwards each floating point instruction to the FPU. The previous works on cryptographic instruction set extensions have also proposed additional “crypto” units that function exactly like the ALU or the FPU.

We propose a non-linear/linear operation unit – which we refer to as NLU – that can implement logical functions expressed in their algebraic normal form (ANF) in its non-linear mode, and binary matrix multiply-and-add operations in its linear mode. We design the NLU as a single input (source) and single output (destination) unit, where the additional operands (ANF or binary matrix coefficients are stored inside a configuration register). The choice of the operations relies on investigation of several different ciphers (both block and stream) [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] as well as newly introduced lightweight hash functions that use block-cipher-like internal permutations [21], [22], [23], [24]. The two common operations in all of these ciphers are substitution and permutation.

Substitution is the process of transforming $m$ bits input to an $n$ bits output. It can either be realized by table-lookup or by ANF expressions. For lightweight ciphers, both $m$ and $n$ are chosen as 4 bits, unlike the 4-to-6 transformation in the Data Encryption Standard (DES) [25] or the 8-to-8 transformation in AES. Therefore, we have decided to support only 4-bit substitutions with the NLU. In a lookup table based implementation, this means a maximum of 16 different 4-bit output words (for a bijective substitution) selected by 4-bit input, resulting in a total storage of $16 \times 4 = 64$ bits. In an ANF-based implementation, each output bit is expressed in terms of the $1^{st}$ to $4^{th}$ degree product terms of input bits masked with 16 bits of ANF coefficients, which again results in a total storage of $16 \times 4 = 64$ bits. In other words, both approaches result in the same storage cost. However, at the end of our area cost analyses, we have observed that ANF gives better area results, and might also provide additional flexibility for non-substitution type of operations, which might still require ANF form. Another important parameter regarding the substitution operation is the input/output width. While we have chosen to support 4-bit substitutions only, the input operand width is enforced by the target microprocessor architecture. For most of the lightweight embedded applications, this is either 8 or 16 bits. Since the operations within the substitution layer of a cipher can be executed in parallel, it is possible to put two or four of the 4-bit substitution boxes (commonly referred to as S-Boxes) within the same substitution module, where nibbles of the input operand are distributed to each of these S-Boxes, and their output nibbles are concatenated to form the output operand of 8 or 16 bits. It is also common to have 32-bit operands, especially in the high-end embedded processors, as well as 4 bits in the very low-end ones. In either case, only the number of S-Boxes inside the substitution module
will change (to eight or one, respectively). Apart from the configuration of coefficients, substitution requires an instruction with source and destination registers as operands.

We accomplish this functionality via the NNL (non-linear operation) instruction, which accepts source register and destination register as input and output arguments, respectively. 8-bit content of the source register is split into two 4-bit nibbles, on which the non-linear operation described by the ANF coefficients is performed in parallel. The two 4-bit results are then combined to form the 8-bit output for the destination register.

Permutation is the process of bit-shuffling in order to transpose bits across several S-Boxes, which therefore requires input/output widths larger than (almost always multiples of) S-Box widths. In our case of 4-bit S-Boxes, the permutation layer should be at least 8 bits wide. Again, this choice is enforced mainly by the target microprocessor’s datapath width. For example, with an operand width of 8 bits, it would be hard and infeasible to implement a 64-bit generic permutation block. We therefore decided to limit our permutation module also to 8 bits in width and implement it as multiplication with an $8 \times 8$ binary matrix. The choice of $8 \times 8$ is mainly due to the 64 bits storage required for the matrix coefficients, for which the storage required for the ANF coefficients can be reused. Furthermore, it is possible to implement multiplication with a larger matrix by means of repetitive “multiply-and-add”s with smaller matrices. In other words, the permutation layer can be realized with a matrix multiplication unit and an accumulator. Since loading coefficients for each of these smaller matrices requires additional clock cycles and code space, it also makes sense to make reuse of each small matrix. This can be made possible by implementing a first-in-first-out (FIFO) register of depth $d$ instead of a single register ($d = 1$) for the accumulator part of permutation unit. Choice of $d$ is yet another parameter for the design. Our investigation of several different ciphers showed us that most permutation operations require inputs from a maximum of 32 bits (8 S-Boxes), which means that choosing $d = 4$ would be sufficient for most applications. It should be possible to accumulate onto any desired part of the FIFO register, which means choosing from one of the four bytes of the 32 bits. Furthermore, the permutation operation could also be run in “multiply-only” mode. Therefore we need either a single instruction with an input and output operand, 1-bit for the mode selection (multiply-only or multiply-and-add), and 2 bits for the FIFO output selection in the “multiply-and-add” mode; or two separate instructions — one for multiply-only, and one for multiply-and-add.

In our design, we have opted for the latter case and implemented two new instructions — NMU (multiply) and NMA (multiply-and-add). Both instructions take the usual source and destination registers as input and output arguments, and perform binary matrix multiplication on the input byte. For NMU, this is the final result. In the case of NMA, the result of the multiplication is added onto the content of the $s^{th}$ register in the FIFO to obtain the final result, where $s$ is the second input argument for this instruction. In both cases, the final result is both sent to the destination register and pushed into the FIFO.

The configuration of the bits for the ANF or matrix coefficients is the last issue in the instruction set architecture. In parallel to our choice of 8-bit operands, we can perform this configuration by 8 bits at a time. It can be done either by writing 8 bits into a target portion of 64 bits selected via 3-bit address, or by shifting 8 bits into the configuration register. The latter option is a better choice when supported with the selection of number of bits to be shifted. In our case, this changes from 1 to 8 bits, selected by 3 bits, resulting in an instruction with an 8-bit immediate value for the configuration and 3 bits of length selection.

This is accomplished via the NLD (load NLU configuration) instruction. Its input arguments are the 8-bit input stream, $K$, and the 3-bit number of bits (to be shifted), $n$. NLD has no output arguments, as the NLU configuration register is the default target.

Details for the NLU instructions (supported by hardware choices) are given in Table I. These instructions assume an 8-bit microprocessor architecture with 16-bit instructions, which is very common for most of the embedded processors. However, they can easily be modified for larger (or smaller) instruction widths.

III. Unified Non-linear/Linear Hardware Unit

To realize the mentioned NLU (non-linear/linear unit) instructions, we implement a unified hardware unit performing non-linear and linear operations as explained in Section II. In this unified structure, we have non-linear and linear operational sub-units, a 64-bit configuration register to store the ANF/matrix coefficients, and also an accumulation module for the add part of the linear “matrix multiply-and-add” operation along with depth-4 FIFO registers. The overall block diagram of this structure is given in Figure 1.

The push input is activated whenever an NLD instruction is issued. The immediate value part of the instruction is pushed from right to left in $n$ bits, which results in $t$ clock cycles (where $1 \leq t \leq 8$) to store a new 64-bit coefficient data. During the configuration phase, the register acts like a shift-register. Sel selects the number of most significant bits of the immediate value to be pushed. This is accomplished by an 8-to-1 barrel-shifter of 64 bits (not shown on the circuit schematic). After the configuration of the coefficient values, NLU can be operated in either non-linear or linear mode with respect to the issued instruction. The 64-bit coefficient data from the configuration register are connected to both the non-linear and linear units together with the 8-bit input data (operand). The output of the NLU is selected by the mode input — non-linear result or linear result. Mode is set
Table I

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLD Load NLU configuration</td>
<td>NLD n, K</td>
<td>CONF ← CONF ≪ K[MSB-n], if n&gt;0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONF ← CONF ≪ K, else</td>
</tr>
<tr>
<td>NNL NLU non-linear operation</td>
<td>NNL Rd, Rs</td>
<td>Rd(7:4) ← ANF[Rs(7:4)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rd(3:0) ← ANF[Rs(3:0)]</td>
</tr>
<tr>
<td>NMU NLU multiply operation</td>
<td>NMU Rd, Rs</td>
<td>Rd ← M × Rs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIFO ← FIFO ≪ M × Rs</td>
</tr>
<tr>
<td>NMA NLU multiply-and-add</td>
<td>NMA s, Rd, Rs</td>
<td>Rd ← M × Rs + FIFO(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIFO ← FIFO ≪ [M × Rs + FIFO(s)]</td>
</tr>
</tbody>
</table>

The non-linear unit consists of many AND and XOR gates, which actually express the S-Box in algebraic normal form (ANF). The input (masking) coefficient values define the ANF coefficients of the S-Box used in the cipher. In our case of 8-bit operand, we use two identical S-Boxes in parallel, thereby sharing the same ANF masking coefficients. The most significant 4 bits of the input data go into the first S-Box and the least significant 4 bits to the second S-Box. The outputs of the two S-Boxes are concatenated to form the 8-bit output in the same order as the input bits. For each S-Box output, the first 16 bits of the mask value define the first bit of the S-Box result, the second 16 bits define the second bit, and so on. Detailed schematic of the non-linear unit with all the masking coefficients ($m_0...63$) is shown in Figure 2.

The linear operations part performs bitwise operations expressed in “matrix multiply-and-add” form. The linear unit handles the matrix multiply step by multiplying the 8-bit input data with an $8 \times 8$ binary matrix, whose coefficients are taken from the 64-bit configuration register. This matrix multiplication can be expressed as:

$$
\begin{align*}
\text{out}_7 &= (m_{63} \land i_{7}) \oplus \ldots \oplus (m_{56} \land i_{4}) \\
\text{out}_6 &= (m_{55} \land i_{7}) \oplus \ldots \oplus (m_{48} \land i_{4}) \\
& \vdots \\
\text{out}_1 &= (m_{15} \land i_{7}) \oplus \ldots \oplus (m_8 \land i_{0}) \\
\text{out}_0 &= (m_7 \land i_{7}) \oplus \ldots \oplus (m_0 \land i_{0})
\end{align*}
$$

Figure 3 shows this multiplication scheme and Figure 4 presents the detailed circuit diagram of the linear matrix multiplication unit. An additional accumulation circuit performs the add step of “matrix multiply-and-add” structure. When acc is set (i.e. NMA instruction is issued), the selected FIFO register output is added to the matrix multiplication result. sro input (corresponding to the $n$ parameter of the NMA instruction) decides the register output to be taken. The output of each “multiply-only” or “multiply-and-add” operation is pushed into the FIFO to allow following accumulations via the mac input which is activated with each NMU or NMA instruction. The additional accumulation circuit can be seen in Figure 1 (on the right side of the figure), next to the linear unit.

To observe the impact of the NLU unit on area and cost, we implemented the unit as a synthesizable RTL code and synthesized in UMC 90 nm low-leakage Faraday library. Our implementation occupies 1752 gate equivalents (GE) and the (simulational) power consumption is 28.59 $\mu$W @100 KHz (simulational). Therefore, our design is very compact and cheap as an extension unit.
IV. APPLICATIONS

We demonstrate the effectiveness of the NLU in terms of code space and execution time by implementing two lightweight ciphers and two full size ciphers on a popular embedded microcontroller. As the lightweight cipher, we have chosen PRESENT and CLEFIA, which are the new ISO standards; and as the full size cipher, we have chosen SERPENT and AES (NIST encryption standard). We have implemented different versions of these ciphers on Atmel's AVR microcontroller [26] in assembly using both standard instruction set and the NLU instruction set extension to have a fair comparison. The implementation details for the ciphers are given in the following subsections.

A. PRESENT Software Implementation

PRESENT is a 64-bit block cipher with 80 or 128-bit key size; and it is one of the most compact ciphers for hardware implementation. It is composed of a simple substitution-permutation (SP) network with 31 rounds. Its substitution layer is composed of sixteen 4-bit S-Boxes in parallel. The permutation layer is extremely simple: It is only bit shuffling, which has zero cost for hardware. However, the same arguments are not valid for software. The bit shuffling is a very costly operation when implemented with standard instructions. 4-bit S-Box also makes direct software implementation harder in non-4-bit processors. Programmers usually prefer to use 8-bit T-tables, where the 16 entries of the S-Box are repeated 16 times at the low nibbles of 256 bytes, whereas the high nibble changes once in every 16 entries. This implementation makes it possible to run two parallel S-Box operations at a time, while costing 256 bytes instead of the ideal case of 8 bytes for S-Box storage.

In an optimized implementation of the PRESENT cipher in AVR assembler [26], each 1-byte S-Box operation takes 6 instructions, one of which is a program memory load instruction with 3 cycles while others are all single cycle instructions. Therefore, total cost for the substitution layer (i.e. 64 bits) is 64 cycles and 48 lines of code. There is also the 256 bytes of T-table storage within the program memory. Permutation is much worse, where it takes 16 cycles to complete for each byte, resulting in a total of 128 cycles (plus overhead) per round for the permutation layer. Similar arguments are also valid for the key expansion, where the 80-bit key has to be rotated by 61 bits in every round. In total, this implementation occupies 660 bytes of program (flash) memory and takes 10792 cycles to complete encryption of a single 64-bit input block with an 80-bit key.

In our implementation of PRESENT using the NLU instruction set extension, it takes 8 cycles (and 8 lines) to load
the ANF coefficients into the NLU configuration register. Then it takes 1 cycle per byte for the S-Box operation, as shown in the following assembly code segment:

```Assembly
; load ANF bits for the PRESENT S-Box
NLD 0, 0x83
NLD 0, 0x92
NLD 0, 0x67
NLD 0, 0x0B
NLD 0, 0xDE
NLD 0, 0x43
NLD 0, 0x4A
NLD 0, 0x80

; perform non-linear S-Box operation
NNL r18, r18
NNL r19, r19
NNL r20, r20
NNL r21, r21
NNL r22, r22
NNL r23, r23
NNL r24, r24
NNL r25, r25
```

We also use NLU for the permutation layer - this time in linear mode. We first write the expressions for the bit permutations as follows (where \( y_i \) is the output bit and \( A_j \) is the input bit in big-endian format):

\[
\begin{align*}
y_{\{63...56\}} &= a_{\{63,59,55,51,47,43,39,35\}} \\
y_{\{55...48\}} &= a_{\{31,27,23,19,15,11,7,3\}} \\
y_{\{47...40\}} &= a_{\{62,58,54,50,46,42,38,34\}} \\
y_{\{39...32\}} &= a_{\{30,26,22,18,14,10,6,2\}} \\
y_{\{31...24\}} &= a_{\{61,57,53,49,45,41,37,33\}} \\
y_{\{23...16\}} &= a_{\{29,25,21,17,13,9,5,1\}} \\
y_{\{15...8\}} &= a_{\{60,56,52,48,44,40,36,32\}} \\
y_{\{7...0\}} &= a_{\{28,24,20,16,12,8,4,0\}}
\end{align*}
\]

In the next step, we convert these expressions to matrix form (shown only for the first output byte):

\[
\begin{bmatrix}
y_{63} \\
y_{62} \\
y_{61} \\
y_{60} \\
y_{59} \\
y_{58} \\
y_{57} \\
y_{56}
\end{bmatrix}
= \begin{bmatrix}
10000000 \\
00010000 \\
00001000 \\
00000100 \\
00000010 \\
00000001 \\
00000000 \\
00000000
\end{bmatrix}
\oplus \cdots \oplus
\begin{bmatrix}
a_{63} \\
a_{62} \\
a_{61} \\
a_{60} \\
a_{59} \\
a_{58} \\
a_{57} \\
a_{56}
\end{bmatrix}
\begin{bmatrix}
00000000 \\
00000000 \\
00000000 \\
00000000 \\
10000000 \\
00000000 \\
00000000 \\
00000000
\end{bmatrix}
\]

Writing this for all the bytes, we get:

\[
\begin{align*}
Y_7 &= M_{60}A_7 \oplus M_{61}A_6 \oplus M_{62}A_5 \oplus M_{63}A_4 \\
Y_6 &= M_{60}A_6 \oplus M_{61}A_5 \oplus M_{62}A_4 \oplus M_{63}A_3 \\
Y_5 &= M_{10}A_7 \oplus M_{11}A_6 \oplus M_{12}A_5 \oplus M_{13}A_4 \\
Y_4 &= M_{10}A_6 \oplus M_{11}A_5 \oplus M_{12}A_4 \oplus M_{13}A_3 \\
Y_3 &= M_{20}A_7 \oplus M_{21}A_6 \oplus M_{22}A_5 \oplus M_{23}A_4 \\
Y_2 &= M_{20}A_6 \oplus M_{21}A_5 \oplus M_{22}A_4 \oplus M_{23}A_3 \\
Y_1 &= M_{30}A_7 \oplus M_{31}A_6 \oplus M_{32}A_5 \oplus M_{33}A_4 \\
Y_0 &= M_{30}A_6 \oplus M_{31}A_5 \oplus M_{32}A_4 \oplus M_{33}A_3
\end{align*}
\]

As seen in the matrix form, an output byte is obtained by sum of multiplications of four input bytes with four different matrices, where each matrix is a 2-row shifted version of the neighboring matrix. In total, we need 16 different matrices, which can be obtained from each other in a sequence of row shifts. In the case of NLU, each byte written into the configuration register corresponds to left shift of the coefficients by one byte, which in fact is a 1-row shift upwards. Therefore we start with multiplication of the rightmost byte, and continue towards the leftmost byte by consecutive multiply-and-adds. However, since each byte pair uses the identical four matrices, we can also make use of this property by performing rightmost multiplications of two bytes. We then continue with the multiply-and-adds in pairs, using the second register output from the accumulator FIFO. The resultant assembler code for the permutation layer is given as follows (only initialization and permutation for the first two bytes are shown):

```Assembly
; state is in registers r18 to r25
NLD r18, r18
NLD r19, r19
NLD r20, r20
NLD r21, r21
NLD r22, r22
NLD r23, r23
NLD r24, r24
NLD r25, r25

; write NLU to FIFO
NLD 0, 0x0
NLD 0, 0x0
NLD 0, 0x0
NLD 0, 0x0
NLD 0, 0x0
NLD 0, 0x0
NLD 0, 0x80
NLD 0, 0x40
```
As seen in the code, it takes 16 cycles in average for permutation of two bytes, or 8 cycles per byte, which is only half of the implementation without NLU instructions. After applying similar use of NLU for the key expansion block as well, we end up with 406 bytes of program memory and 6017 cycles to complete encryption of a single 64-bit input block. The reduction in both code space and execution time is considerable with respect to a raw assembly implementation.

B. CLEFIA Software Implementation

CLEFIA is a 128-bit block cipher with 128, 192, or 256-bit key size. It is based on a 4-branch generalized Feistel structure, which uses two different diffusion functions and two different 8-bit S-boxes. On the key schedule part, the same diffusion functions are used to generate the intermediate key, \( L \), from the input key, \( K \). Both keys, \( K \) and \( L \), are then expanded to round keys using a permutation called DoubleSwap function. The fastest and the most compact version of the cipher, the 128-bit key version, uses 18 rounds for both encryption and decryption.

In our comparison, we have only implemented fast and compact versions of the 128-bit keyed encryption function with minimal optimization in both standard assembly and NLU instruction set extension cases. The 8-bit S-Boxes are most efficiently implemented as lookup tables in either case. However, the situation is the opposite for diffusion and permutation functions. Since both of them are based on linear combinations of shifted bytes, they are perfect examples to utilize the linear mode operation of the NLU.

For example, the basic operation of the diffusion function is multiplication by 2 over finite field \( GF(2^8) \) defined by the primitive polynomial \( p(x) = x^8 + x^4 + x^3 + x^2 + 1 \). It is repeated several times inside both diffusion operations. In the standard assembly, it is implemented as a subroutine which takes 15 or 17 cycles (depending on the most significant bit of the input byte), including the subroutine calls. It is also possible to reduce the cycle counts to 7 or 9 at the cost of repeating the corresponding program code several times, adding hundreds of bytes to the occupied program memory.

On the other hand, the whole multiplication is only a single NLU instruction for NLU, following a one-time initialization of the internal product matrix (via \( \text{NLD} \) instruction). Similar approach is also used for the DoubleSwap function. The resultant NLU based code is 1912 bytes vs. 2170 and 3046 bytes in the compact and fast standard codes, respectively. Execution time is reduced to 15268 cycles from 28684 cycles in the fast version and 42124 cycles in the subroutine-based compact version.

C. SERPENT Software Implementation

SERPENT is an interesting case study for several reasons: It is the cipher from which PRESENT has originated. Therefore, they have several similarities. However, SERPENT is defined for bit-slice implementation. Therefore, its code can be optimized for different targets: Either for code space (compactness) or for execution time. In the compact case, S-Boxes are implemented as lookup tables, which requires un-bitslicing and re-bitslicing state bits before and after S-Boxes. Such an implementation is much slower than a bit-sliced implementation.

Since NLU instruction set is optimized for ANF based implementations, it is not directly applicable to a bit-sliced cipher. However, in the case of SERPENT, un-bitslicing operation is equivalent to the permutation layer of PRESENT, only with a larger block size – 128 bits instead of 64. Therefore, PRESENT permutation layer code can be reused together with substitution layer codes. Finally, re-bitslicing has to be done, which is the inverse permutation of PRESENT and can be implemented using matrix multiply-and-adds.

We have applied this strategy in re-writing the SERPENT code, resulting in substantial gains both in code space and execution time. We have also re-written the bit-sliced linear layer of SERPENT using NLU linear operations, which provided nominal gain. As a result, we have come up with a 10% faster software implementation, which is only one third of the fastest SERPENT code. It is also more than twice as fast as the most compact implementation, but slightly larger in code size.

D. AES Software Implementation

Since AES is the most widely used and still unbroken block cipher, we have also investigated the possibility of implementing AES using NLU instructions. In this case, as in CLEFIA, we were not able to implement the S-Boxes more efficiently than lookup tables. However, there is space for improvement for the implementation of MixColumns diffusion operator.
In an optimized implementation of the AES cipher, MixColumns operation takes 117 instructions, 16 of which are 3-cycle program memory load instructions, while all the others are single cycle instructions. Therefore, total execution time for the MixColumns diffusion operator is 149 cycles, whereas it requires a total of 492 bytes of program memory – 236 bytes for 118 lines of code and 256 bytes for an auxiliary lookup table.

In our implementation using the NLU instruction set extension, the same operation requires only 112 lines of code (224 bytes of program memory), which also corresponds to 112 cycles of execution time. In order to implement MixColumns via NLU instructions, we use the linear mode of NLU. Let’s recall the expressions for MixColumns:

\[
Y_0 = \{02\}A_0 \oplus \{03\}A_1 \oplus A_2 \oplus A_3
\]
\[
Y_1 = A_0 \oplus \{02\}A_1 \oplus \{03\}A_2 \oplus A_3
\]
\[
Y_2 = A_0 \oplus A_1 \oplus \{02\}A_2 \oplus \{03\}A_3
\]
\[
Y_3 = \{03\}A_0 \oplus A_1 \oplus A_2 \oplus \{02\}A_3
\]

This can also be expressed in matrix form as follows (shown only for the first output \(Y_0\)):

\[
\begin{bmatrix}
97^* \\
96 \\
95 \\
94 \\
93 \\
92 \\
91 \\
90 \\
99 \\
98 \\
97 \\
96 \\
95 \\
94 \\
93 \\
92 \\
91 \\
90 \\
99 \\
98 \\
97 \\
96 \\
95 \\
94 \\
93 \\
92 \\
91 \\
90
\end{bmatrix}
\begin{bmatrix}
01000000 \\
00100000 \\
00010000 \\
10001000 \\
10010100 \\
10011100 \\
10011010 \\
00001010 \\
00000110 \\
00000011 \\
00000001 \\
10000000 \\
10000010 \\
10000100 \\
10000110 \\
10000011 \\
00000000 \\
00000001 \\
10000001 \\
10000000
\end{bmatrix}
\begin{bmatrix}
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^* \\
0^*
\end{bmatrix}
\]

The matrix form representation allows us to implement it using the linear mode of NLU. The most code line consuming part is the initializations of the matrices with coefficients for multiplication by 2 and 3. Re-initialization with two matrices (×2 and ×3) for every column corresponds to eight initializations, which is optimized by reusing the last initialized matrix. Only initialization for the first column requires both matrices, while the other columns require only a single matrix initialization.

Implementation using NLU instructions reduces execution time of the MixColumns operation by 24%, and its code size by 55%. In the overall AES encryption and decryption, this corresponds to a total reduction of 12% (2406 vs. 2739 cycles) and 10% (3246 vs. 3579 cycles) in execution time, respectively. Program memory size reduction for the combined encryption and decryption is 11% (1402 vs. 1570 bytes).

V. RESULTS AND DISCUSSION

The performance that can be achieved via NLU unit is shown in Table II with a comparison to traditional software implementations of PRESENT, CLEFIA, SERPENT and AES.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Clock Cycles</th>
<th>Flash Memory Utilization</th>
<th>Time-Area Product (TAP) (cycles-bytes)</th>
<th>TAP Gain (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENT (LUT)</td>
<td>10792</td>
<td>660 bytes</td>
<td>2.7 × 10^6</td>
<td>0</td>
</tr>
<tr>
<td>PRESENT (NLU)</td>
<td>6017</td>
<td>406 bytes</td>
<td>2.4 × 10^6</td>
<td>66</td>
</tr>
<tr>
<td>CLEFIA (compact)</td>
<td>42124</td>
<td>2170 bytes</td>
<td>9.1 × 10^6</td>
<td>0</td>
</tr>
<tr>
<td>CLEFIA (fast)</td>
<td>28684</td>
<td>3046 bytes</td>
<td>87.4 × 10^6</td>
<td>4</td>
</tr>
<tr>
<td>CLEFIA (NLU)</td>
<td>15268</td>
<td>1912 bytes</td>
<td>29.2 × 10^6</td>
<td>68</td>
</tr>
<tr>
<td>SERPENT (ANF)</td>
<td>49314</td>
<td>7220 bytes</td>
<td>356.0 × 10^6</td>
<td>0</td>
</tr>
<tr>
<td>SERPENT (LUT)</td>
<td>100338</td>
<td>2620 bytes</td>
<td>278.6 × 10^6</td>
<td>22</td>
</tr>
<tr>
<td>SERPENT (NLU)</td>
<td>45431</td>
<td>2960 bytes</td>
<td>134.5 × 10^6</td>
<td>62</td>
</tr>
<tr>
<td>AES (LUT)</td>
<td>3159</td>
<td>1570 bytes</td>
<td>4.96 × 10^6</td>
<td>0</td>
</tr>
<tr>
<td>AES (NLU)</td>
<td>2826</td>
<td>1402 bytes</td>
<td>3.96 × 10^6</td>
<td>20</td>
</tr>
</tbody>
</table>

Table II

Performance Comparison

For SERPENT, we have two different implementations: One is using ANF representations for the substitution part and implemented in bitsliced form, and the other is again implemented in bitsliced form except for S-Box operation. In the latter, we have a lookup table based substitution layer as in the case of PRESENT. Our NLU-based SERPENT design is even faster than the ANF-based SERPENT implementation and uses approximately the same amount of memory with LUT-based implementation. To compare our AES results, we use an optimized software implementation, which uses lookup tables for both S-Boxes and MixColumns operation. We were only able to optimize the MixColumns operation, which alone reduced both the code size and execution time. Depending on the implementation (both encryption and decryption or encryption-only), code size reduction is between 10-18%, while execution time reduction is around 10% for both cases.

We also list the time-area (execution time and program memory) products for each case in order to highlight the NLU instruction set extension advantage. The execution times are given as average times. RAM usage is independent of implementation in all four ciphers.

VI. CONCLUSION AND FUTURE WORK

In this work, we were able to improve the software implementations of lightweight block ciphers on small CPUs. We came up with a non-linear/linear instruction set extension, NLU, that can implement non-linear and linear operations. Using the new four NLU instructions, we show that we achieve time-area product reductions of 20-70% for the
widely used ciphers PRESENT, SERPENT, CLEFIA, and AES. The area overhead for the extension unit is only 1.8 KGE – almost the same as the area of a parallel PRESENT core designed in the same fashion.

The evaluation of NLU instructions with several different ciphers on various embedded microcontroller architectures is in progress. Among these, 32-bit microcontrollers are predicted to be dominating the embedded design domain in the future. Further work on 32-bit microprocessors can be the scope of another study.

REFERENCES


[26] ATmega8 Datasheet. Atmel AVR.