Block Cipher Speed and Energy Efficiency Records on the MSP430: System Design Trade-Offs for 16-bit Embedded Applications*

Benjamin Buhrow, Paul Riemer, Mike Shea, Barry Gilbert, and Erik Daniel

Mayo Clinic, Rochester, MN, USA
buhrow.benjamin@mayo.edu, riemer.paul@mayo.edu, shea.michael@mayo.edu, gilbert.barry@mayo.edu, daniel.erik@mayo.edu

Abstract. Embedded microcontroller applications often experience multiple limiting constraints: memory, speed, and for a wide range of portable devices, power. Applications requiring encrypted data must simultaneously optimize the block cipher algorithm and implementation choice against these limitations. To this end we investigate block cipher implementations that are optimized for speed and energy efficiency, the primary metrics of devices such as the MSP430 where constrained memory resources nevertheless allow a range of implementation choices. The results set speed and energy efficiency records for the MSP430 device at 132 cycles/byte and 2.18 μJ/block for AES-128 and 103 cycles/byte and 1.44 μJ/block for equivalent block and key sizes using the lightweight block cipher SPECK. We provide a comprehensive analysis of size, speed, and energy consumption for 24 different variations of AES and 20 different variations of SPECK, to aid system designers of microcontroller platforms optimize the memory and energy usage of secure applications.

Keywords: AES, SPECK, lightweight, encryption, MSP430, speed, energy, efficient, measurements, trade-offs

1 Introduction

Many lightweight block ciphers have been established in recent years in response to the growing use of resource-constrained electronic devices in a wide variety of embedded applications. Examples include TWINE [1], Piccolo [2], Lblock [3], LED [4], PRESENT [5], SIMON, and SPECK [6], in addition to the mainstay AES [7]. Lightweight block ciphers are largely targeted or optimized for small hardware implementations although some are specifically architected to admit software-friendly designs for microcontrollers (e.g., TWINE and SPECK).

Microcontroller based software applications occupy an interesting middle ground: resources are very constrained relative to general purpose 32- or 64-bit processors but they are abundant relative to devices like RFID tags or smart

* The final publication will be available at Springer via the Latincrypt 2014 proceedings

cards. For example, sensor nodes like the MicaZ [8] or TelosB [9] utilize microcontroller devices that offer enough ROM and RAM to implement large lookup tables or to unroll program loops. Further, the programmable nature of microcontrollers provides support for diverse applications, each with a different set of resource requirements, of which the block cipher is typically only a small part. Outside of choosing different block cipher algorithms, the ability to tailor a particular algorithm to the device resources at hand is desirable; for example when an algorithm provides exceptional security or energy efficiency.

Overall, many parameters of block ciphers are important to an embedded system designer such as size, security, speed, and energy efficiency, depending on the application. Varying a block cipher's implementation strategy within embedded devices is a relatively unexplored topic, and one that can provide much in the way of trade-off data to designers. Survey authors do a thorough evaluation over many different ciphers (e.g., [10], [11], and [12]), but in many cases one implementation strategy (e.g., small size versus high speed or C language versus assembly language [hereafter, abbreviated to assembler]) is chosen per cipher without much discussion. In this paper we quantitatively discuss this issue by measuring multiple implementations of two algorithms: AES and the lightweight block cipher SPECK. These were chosen for two primary reasons: both are expected to have good performance on the MSP430 [13] and each can be implemented in a variety of ways on 16-bit platforms that trade off size for speed. We focus on SPECK over other lightweight block ciphers because 1) it is a very recently proposed cipher and its implementation has not been fully explored on the MSP430 platform and 2) the wide range of block and key sizes in SPECK are interesting from the standpoint of configurablity. The intent is not to promote one algorithm as "better" or "worse" than the other - their designs are sufficiently different to preclude such a comparison. Nor is the intent to provide security analysis of either algorithm, other than by increasing key sizes. The goals are to provide system designers data and analysis over a wider range of size, speed, and energy efficiency than could be obtained from either block cipher alone, and to discuss efficient implementations of each algorithm.

The primary contributions of this paper are the presentation of a matrix of size, speed, and energy consumption data for 8 different implementation strategies of AES coupled with its 3 different key sizes and a first look at similarly thorough results for the entire family of 10 different SPECK parameterizations, for both C and assembler implementations. The thorough analysis across AES implementation strategies on the MSP430 presented here is unavailable elsewhere in the literature, to our knowledge. SPECK is a relatively new cipher and the implementations here represent the most thorough to date. Our fastest 128-bit implementations operate at 132 cycles/byte for AES-128 and 103 cycles/byte for SPECK-128, both setting speed records for 128-bit block ciphers on the MSP430. Measured energy of 2.18 μJ/block for AES and 1.44 μJ/block for SPECK fit in at numbers 5 and 6 in Guneysu's list of top energy efficient AES implementations for any platform [14], but notably the results are considerably more efficient than all other microcontroller platforms tested in that report.

In addition, we provide C and assembler implementation tactics for our AES designs as well as a detailed review of SPECK implementations in C that improve performance relative to conventional approaches. None of the implementation strategies used here for AES are new; however the 16-bit optimization of Gouvea [15] is fairly recent and led to the record speed and energy efficiency results. For all implementations we concentrate solely on encryption and omit decryption.

The remainder of the paper is organized as follows. In Section 2 we discuss related work, in Section 3 the algorithms of study and their implementation variations, in Section 4 efficient implementation details, in Section 5 the experimental setup, metrics, and results, and in Section 6 our conclusions.

2 Related Work

To a system designer choosing a block cipher for adoption in a microcontroller-based application, several relevant works exist. As previously mentioned, many lightweight block ciphers have been recently proposed and their authors typically offer performance results and/or implementation tactics although the scope of these efforts varies. Several surveys help distill the relative performance of these lightweight and other block ciphers. For example Eisenbarth et al. in [10] provides results of several ciphers on an 8-bit ATtiny45 device. The authors concentrate on small size as a design goal and provide energy consumption data but the results are of limited relevance to this study given the differences in target platform. Law et al. in [11] compares block ciphers on an MSP430F149 device. They adopt source code from public sources such as OpenSSL [16]. This approach ensures quality code, but fixes the implementation strategy to that of the public source that is not necessarily optimized for embedded devices. Cazorla et al. in [12] compare 12 lightweight and 5 conventional block ciphers on an MSP430F1611 device. The authors compare many ciphers, but understandably chose a single implementation for each and do not state any particular optimization goals. Didla in [17] investigates implementation tactics of AES in a MSP430F1611 device; however, all are variations of AES for 8-bit platforms. Finally, in [18] the authors compare AES with other block ciphers on both MSP430- and ATmega-based platforms. They address the variable key size of AES but otherwise choose a single implementation (unstated, but from their provided ROM size it appears to be a table-based one).

Concerning speed records for AES on microcontroller devices, Hyncica in [19] presents optimized AES results of 172 cycles/byte for the MSP430 platform that is based on 32-bit table-based code ported from LibTomCrypt [20]. Gouvea [15] first presented the 16-bit lookup table strategy for AES in which they reported 180 cycles/byte on a MSP430 platform. On an AVR device the current speed record is described by Bos in [21], previously held by Poettering in [22].

Implementation and analysis results have begun to appear for SPECK. The designers present implementation results for SPECK on Atmels ATmega128 8-bit processor and the BLOC project [13] provides preliminary performance data on the MSP430. Cryptanalysis of SPECK can be found in [23], [24], and [25].

3 Algorithms and Implementation Variations

3.1 AES

The AES algorithm uses a substitution-permutation approach and operates on a block size of 128-bits organized as a 4x4 array of bytes [7]. Four basic transformations are iteratively applied over a variable number of rounds (depending on key size) to complete each block encryption. These operations are SubBytes, ShiftRows, MixColumns, and AddRoundKey. Of these, MixColumns is the most complex operation requiring multiplication of state bytes by constants over the Galois Field $GF(2)^8$. Most of the AES implementation variations in common use concern themselves with optimizing this transformation.

Daemen and Rijmen in [26] discuss implementation aspects for both 8-bit and 32-bit processors. In the 8-bit approach, SubBytes, ShiftRows, and AddRoundKey can be easily combined and executed byte-by-byte for each of the 16 input bytes and the MixColumns step can also be implemented efficiently. In this paper the 8-bit approach is implemented in 4 different ways. The first is optimized for speed by unrolling all transformations within each round. It was adapted from the implementation provided by Texas Instruments (TI) in [27]. The second also follows [27], but condenses the transformations into nested loops to reduce ROM size. The third and fourth variations further optimize for speed by introducing extra 256-byte lookup tables to speed up the field multiplications within MixColumns. In the sections below, these four 8-bit variations are referred to as 8-BIT-UNROLL, 8-BIT-LOOPED, 8-BIT-2T, and 8-BIT-2SBOX, respectively.

The 32-bit approach discussed by Daemen and Rijmen is also practical using the 16-bit instruction set of the MSP430. The matrix formulation of the round transformation can be used to define a set of four 256-entry 32-bit lookup tables, known as T-tables, for a total of 4096 precomputed and stored bytes. One iteration of the round function amounts to 16 table lookups and 16 XORs. Three of the T-tables are byte rotations of the first T-table, thus as a space/speed tradeoff, 1024 bytes of storage can be used together with cyclic 8-bit shifts of the single table. Both of these 32-bit variations are implemented; in the sections below they are referred to as 32-BIT-4T and 32-BIT-1T, respectively.

A new optimization was proposed by Gouvea [15] targeting 16-bit processors. This new approach is a variation of the 32-bit table lookup approach where 4 tables of 16-bit entries are defined such that each of the original 32-bit tables can be constructed by concatenating two of the 16-bit tables. This formulation reduces the memory requirement by a factor of 2. After initial tests showed that this variation was the best performing in terms of speed and energy consumption, it was also implemented in assembler. In the sections below these variations are referred to as 16-BIT-4T and 16-BIT-4T-ASM, respectively.

3.2 SPECK

SPECK is a family of lightweight block ciphers with a wide range of block and key size choices and hence is potentially interesting to system designers

desiring trade-offs between size, speed, and security. SPECK is a Feistel-like algorithm that uses the map $R_k : \mathrm{GF}(2)^N \times \mathrm{GF}(2)^N \rightarrow \mathrm{GF}(2)^N \times \mathrm{GF}(2)^N$, where $k \in \mathrm{GF}(2)^N$, defined by

$$R_k(x, y) = ((S^{-\alpha}x + y) \oplus k, S^{\beta}y \oplus (S^{-\alpha}x + y) \oplus k) \quad (1)$$

where $\oplus$ denotes bitwise XOR, $+$ denotes addition modulo 2^N, S^j, S^{-j} denote left and right circular shifts by j bits, and α, β are constants defined according to the block size chosen.

The family of SPECK algorithms is defined according to Table 4.1 in [6], reproduced here for convenience in Table 1. We implemented each of the 10 parameterizations of SPECK shown in Table 1 in both C and assembler. In the sections below we refer to these implementations by the version name with an -ASM or -C suffix for assembler or C, respectively.

Table 1. SPECK parameters

Block size $2n$	Key size mn	Word size n	Key words m	Rotation α	Rotation β	Rounds T	Version Name
32	64	16	4	7	2	22	32-BIT
48	72	24	3	8	3	22	48-BIT
	96		4			23	
64	96	32	3	8	3	26	64-BIT
	128		4			27	
96	96	48	2	8	3	28	96-BIT
	144		3			29	
128	128	64	2	8	3	32	128-BIT
	192		3			33	
	256		4			34	

4 Implementation Details

In all cases the interface to the block ciphers consists of two byte-pointer arguments to an array of bytes to be encrypted and to the expanded key, respectively. We use IAR Embedded Workbench version 5.51 as a development platform. The target device is the MSP430F5528.

4.1 AES 8-BIT

The first 8-bit version of AES, 8-BIT-UNROLL, is based on the implementation by TI for the MSP430 [27] that makes use of the efficient 8-bit implementation hints given in [26].

The 8-BIT-LOOPED version replaces the unrolled MixColumns step with a loop over the 4 columns of the state, and replaces the unrolled AddRoundKey step with another loop over the 16 bytes of the state.

The 8-BIT-2T version replaces each AES GF(2)^8 multiply-by-2, requiring test, branch, shift, and XOR instructions with a single table lookup. The goal with this version is to increase speed at the expense of program size, and increase side-channel timing attack resistance (see Section 4.5).

The 8-BIT-2SBOX version precomputes the 256-byte table 2Sbox = $2 \otimes$ Sbox[a], where $\otimes$ denotes multiplication in the Galois Field, for each input byte a. To see how this is effective, recall that the MixColumns step computes a vector-matrix multiplication, for example,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \begin{pmatrix} \text{Sbox}[a_0] \\ \text{Sbox}[a_1] \\ \text{Sbox}[a_2] \\ \text{Sbox}[a_3] \end{pmatrix} \tag{2}$$

Also recall that in multiplication over GF(2)^8 we have that $3 \otimes \text{SBox}[a_x] == (2 \otimes \text{SBox}[a_x]) \oplus \text{SBox}[a_x]$. Therefore multiplication of Sbox[a_x] by 1, 2, and 3 can be done in a straightforward way by application of Sbox and 2Sbox. Once the row shifts are folded into each column's matrix-vector multiplication, clever ordering of the resulting systems of equations yields a very efficient implementation, as realized by Pottering in [22]. We adapted his 8-bit AVR assembler code for the MSP430.

4.2 AES 32-BIT

The two 32-bit versions of AES use the T-table approach as described by Daemen and Rijmen in section 4.2 of [26]. On 8-bit architectures this approach may not be practical because each of the 32-bit table lookups would require 4 byte-lookups. In total, 64 byte-lookups and 64 XORs per round would be required, which is comparable to the instruction count of a non-table-lookup approach. However on 16-bit architectures 32 word-lookups and 32 XORs per round are required, thus the overall instruction count is reduced quite a bit compared to 8-bit approaches.

In our implementation, the input array of bytes to be encrypted is used directly as the state matrix. The state bytes are used to index the table lookups and the resulting new columns are stored in four temporary 32-bit variables. These are XORed with a 32-bit pointer aliased to the expanded key byte-array. The results are stored back into the state matrix using a 32-bit pointer aliased to the state byte-array. Aliasing the pointer allows access to the same data bytes at different granularity without use of temporary storage. Reducing temporary storage is an important strategy in fast designs, (discussed further in Section 4.5). Pointer aliasing is accomplished using casting, e.g.:

```
state32 = (uint32_t *)state;
```

32-BIT-1T is very similar to the above, except T1, T2, and T3 are replaced with T0 and macros to perform left circular byte shifts by 1, 2, and 3 bytes respectively.

4.3 AES 16-BIT

As described by Gouvea in [15], the 32-bit lookup tables can be reduced to 16-bit lookup tables such that concatenations of two of the 16-bit tables can produce any of the original 32-bit tables. The pointer aliasing approach used in the 32-bit case applies similarly to the 16-BIT-4T version. Of note, by directly using 16-bit operations on 16-bit data types the compiler is no longer relied upon to synthesize 16-bit instructions from source code using 32-bit operations on 32-bit data types. Compilers are not perfect in this regard, so in addition to reducing the memory footprint by a factor of two, this approach was found to be faster as well. There is no equivalent byte-rotation-based memory/speed tradeoff in the 16-bit approach as there is in the 32-bit approach.

After initial testing, the 16-BIT-4T version of AES was found to be the fastest and lowest energy of all of the AES implementation variations tested. A complete listing of 16-BIT-4T can be found in Listing A in the Appendix. To further enhance the performance, 16-BIT-4T was also implemented in assembler, using the IAR generated assembler as a starting point. In the generated assembler, the compiler was unable to utilize the 12 general purpose registers (R4-R15 [28]) of the MSP430 efficiently enough and thus required temporary data to be loaded from and stored to the stack (in RAM). As discussed in Section 4.5, accessing temporary data in RAM is detrimental to speed.

The following register scheme in our assembler version avoids storing temporary data to RAM, thus increasing the speed and further reducing the energy consumption of the 16-BIT-4T-ASM code version. Registers R4 through R11 are used to hold the 8 temporary 16-bit column results, R12 and R13 hold pointers to the state array and expanded key respectively, R14 is the loop counter, and R15 is used to compute offsets into the tables. An excerpt of the resulting assembler round function is shown in the Appendix, Listing B.

It is likely that other AES versions implemented in assembly language would also see performance improvements. For instance, an assembler implementation of the 32-BIT-4T version could likely be made identical to the 16-BIT-4T-ASM version, speed-wise, since in assembler the only difference would be different offsets into the larger 32-bit tables. Based on the preceeding reasoning, we limit our AES assembly language analysis to a single implementation variation; we do not anticipate that 32-bit table-based variations re-implemented in assembly language would exceed the performance of 16-BIT-4T-ASM.

4.4 SPECK

The round function of SPECK is very succinct; therefore all implementations fully unroll the round function, leaving a single loop over the required number of rounds. Beyond decisions like unrolling or not, or inlining or not, Beaulieu in [6] provides no guidance on implementation of the round function. In this section implementations in C and assembly language are discussed.

SPECK performs all operations modulo n, the word size. Whenever the C data type (e.g., uint32_t or uint64_t) is larger than the 16-bit processor word size,

then the compiler must translate XOR, addition (+), and shift operations within C code into multi-precision operations over the native 16-bit instructions of the MSP430 [28]. For example, let $X = \{X_3, X_2, X_1, X_0\}$ and $Y = \{Y_3, Y_2, Y_1, Y_0\}$ be 64-bit integers composed of 16-bit words X_i and Y_i ($0 <= i < 4$). Adding $X + Y$ in C would ideally result in the following sequence of operations in a 16-bit instruction set:

```
add X0, Y0        ; add X0 to Y0
adc X1, Y1        ; add X1 and previous carry to Y1
adc X2, Y2        ; add X2 and previous carry to Y2
adc X3, Y3        ; add X3 and previous carry to Y3
```

C implementations may be preferred by developers (e.g., to simplify the coding effort), and our particular compiler generated efficient multi-precision code for the XOR and addition operations within SPECK. However, the generated code for the circular shifts was not very efficient. Since at least one compiler appears to have difficulty generating efficient code for the SPECK round function, we also implement the round function in assembler to quantify the trade-off.

The SPECK round function in most cases requires both a left circular shift (LCS) by 3 bits and a right circular shift (RCS) by 8 bits. The LCS can be implemented in an efficient way in assembler using three one bit circular shifts, as follows, where 64 bits of data are stored in the four 16-bit registers R4 through R7 :

```
; assembly language 1-bit LCS
; each instruction takes one clock cycle
; (in register addressing mode)
rla r4            ; shift first 16-bit word
rlc r5            ; shift with carry second 16-bit word
rlc r6            ; shift with carry third 16-bit word
rlc r7            ; shift with carry second 16-bit word
adc r4            ; rotate final carry back to first word
```

The 8-bit RCS can be performed in assembler in an efficient way using swap-byte and XOR operations, as shown in Listing C on a 64-bit word held in registers R4-R7. Equivalent LCS and RCS operations in C were not as efficient. For example the RCS implemented as $x = (((x) << 56)|((x) >> 8))$, did not use an extra temporary register and final swap-byte/XOR, as in Listing C, instead using two AND operations (in immediate mode), a swap-byte, and an OR operation. (The immediate addressing mode is slower than the register addressing mode on the MSP430, two clock cycles versus one [28].)

For developers who do not want to proceed to assembler there unfortunately may be limited options to optimize the multi-precision LCS/RCS operations. Typically there is not enough direct access to machine status words, for example to access/modify carry flags, or access to specialized instructions like "rotate left through carry" (rlc), from within high level languages such as C. Listing D in the Appendix shows the full implementation of SPECK-128 in C. The listing

illustrates different ways to implement LCS and RCS that resulted in an 8% speedup over versions that used the C language methods shown above.

4.5 MSP430 features and capabilities

Several features of the MSP430 family of microcontrollers have direct bearing on the implementation results presented in Section 5. Chiefly, these are 1) the instruction set, 2) the addressing modes, and 3) the register set. The MSP430 Family User Guide [28] provides detailed information on all of these features. In this section, we offer comments on specific use of several of the features as they pertain to SPECK and AES implementations.

Instructions on the MSP430 allow operations on either bytes or 16-bit words (via .b or .w suffixes). The swap bytes (swpb) instruction is very useful to SPECK implementations (for the RCS operation). Byte operations to registers clear the most significant byte of the word; this effect is also used during the RCS operation (Appendix, Listing C).

The addressing modes of the MSP430 include register modes (operations on data held in processor registers) and several memory modes (operations on data held in processor memory, RAM or ROM). Operations on data held in memory are generally much slower than data held in registers. For example, to XOR two words held in processor registers takes one clock cycle, but to XOR two words held in memory takes five or six clock cycles, depending on the specific addressing mode employed. As such, whenever possible AES and SPECK code is structured to attempt to minimize loading from and storing to memory. Unfortunately there are only 12 general purpose registers (designated R4 through R15) in which to hold data. A consequence of the limited register set is that temporary variables must be used very sparingly in C code. As the compiler encounters "larger" numbers of temporary variables (e.g., function locals) it will utilize stack memory (physically stored in RAM) to hold them. Accessing these temporary values will therefor incur a speed penalty due to slower memory addressing modes on the MSP430. We do not attempt to quantify "larger" in this study, since detailed examination of the compiler is not our goal (and will be different, for other compilers). However, as the number of temporary variables grows it becomes more difficult for the compiler to avoid temporary use of RAM-based stack.

The MSP430's addressing modes have the advantage that memory accesses are constant time. There are no cache hierarchy effects or interactions with other concurrently running processes to worry about [29]. AES versions that use table lookups thus do not have key- or input-dependent timing variability and appear to have resistance to timing attacks on the MSP430. In our suite of implementations, the only AES versions that do not use table lookups are 8-BIT-UNROLL and 8-BIT-LOOPED. In these implementations, the computation of multiply-by-2 over $GF(2)^8$ depends on the input (a branch containing an extra instruction may or may not be taken). We have not investigated the feasibility of a timing attack on these AES implementations. The SPECK round function involves no branches and on the MSP430 takes constant time. Based on the constant time property of the round function, we expect SPECK to be resistant

to timing-based side-channel attacks on the MSP430, although this has not been investigated.

5 Results and Discussion

5.1 Experimental Setup and Procedure

The 8 variations of AES were evaluated for each of the 3 AES key sizes along with the 20 variations of SPECK (10 in C and 10 in assembler). The metrics for each test were speed of encryption, code size, and energy consumption. Speed was measured using the IAR debugger and function profiler tools in simulation mode. (Speed was also independently verified using timing information obtained from the measured waveforms described below, running released code.) Code size is provided by the IAR linker, broken down into CODE, DATA, and CONST segment sizes. Since all CONST segment data is stored in ROM along with the CODE segment, below we have grouped CODE and CONST together as a total ROM size, reported along with total RAM size (DATA segments). Energy consumption was calculated by first measuring the voltage drop across a 10 ohm resistor in series with the MSP430 digital voltage supply, V_{dvcc}, on a custom evaluation board (nominally $V_{dvcc} = 2.85$ V). Voltage drop was measured using a National Instruments PXI-1024Q chassis, PXI-8108 controller, and PXI-4071 7 digit, 26-bit digitizer. Custom MATLAB scripts then converted the voltage to current and performed integration of the current waveforms over the encryption time-period to get charge, Q. Finally, energy is calculated as $E = QV_{dvcc}$.

In every case key expansion was performed and all round keys were stored in RAM (code to perform key expansion is included in our ROM figures; however, we omit key expansion speed results). (In most cases the key expansion speed is within a factor of 2 of the number of cycles for a block encryption.) Stack utilization also consumes RAM; stack usage was determined by careful examination of compiler generated code.

5.2 Results and Discussion

The results are shown in Figures 1 through 5 below. Figure 1 shows the speed data for each algorithm, arranged right-to-left from fastest to slowest. Figure 2 through Figure 5 are presented in the same x-axis order as Figure 1, i.e., all results are sorted according to speed. Figure 2 through Figure 5 show energy consumption per byte, ROM size, RAM size, and a combined metric, the code size $\times$ cycle count product normalized by block size [10]. In all figures smaller bars are better. In the SPECK charts, "Small Key" refers to the smaller of the key options for each block size shown in Table 1. Similarly, "Large Key" refers to the larger of the key options. The 256-bit key only applies to the 128-bit block size.

For AES, the fastest and most energy efficient C implementation is the 16-BIT-4T variation at 152 cycles/byte and 2.46 μJ/block. The speedup obtained

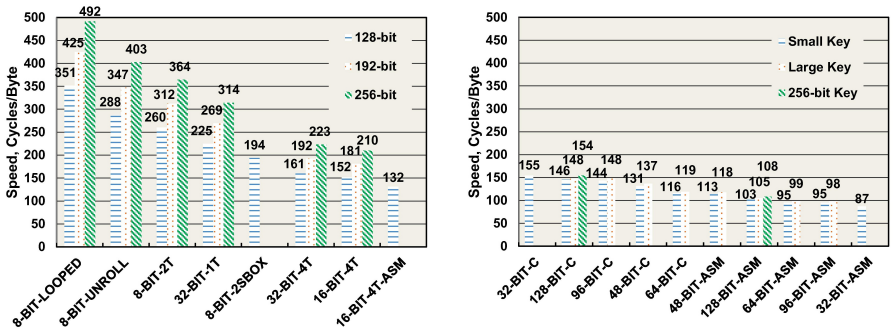

Fig. 1. Speed of AES (left) and SPECK (right) (Block encryption only) (44520)

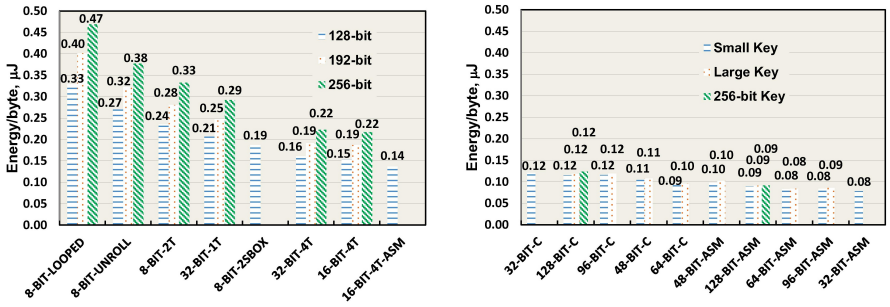

Fig. 2. Energy consumption per byte of AES (left) and SPECK (right) (44522)

over Gouvea's implementation [30] is due to the avoidance of storing the state matrix in temporary stack space. This was accomplished via the pointer aliasing technique discussed in Section 4.3: aliasing the input state array (uint8_t *), used to index the lookup tables, with a word-array pointer (uint16_t *), used for assignments to the state matrix. The 16 extra stack bytes in Gouvea's round function implementation cause more data movement to and from RAM that in addition to adding instructions, incurs the memory addressing mode cycle-count penalty discussed in Section 4.5.

The assembler implementation 16-BIT-4T-ASM gives a further 14% speedup and 12% decrease in energy usage over the C implementation, to 132 cycles/byte and 2.18 μJ/block. This improvement is again a direct consequence of improving register utilization (and thus reducing the memory addressing mode cycle-count penalty). The register utilization scheme that was employed is discussed in Section 4.3.

Of the lighter weight 8-bit AES versions, 8-BIT-2SBOX is the fastest at 194 cycles/byte but has the disadvantage of needing an assembler implementation to realize its performance. The 8-BIT-2T version is 25% slower but much simpler to implement. The combined metric shows that 8-BIT-LOOPED provides very

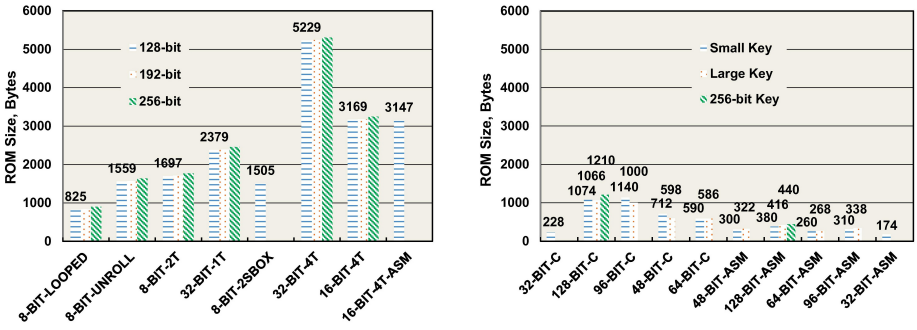

Fig. 3. ROM usage of AES (left) and SPECK (right) (Including key schedule code) (44521)

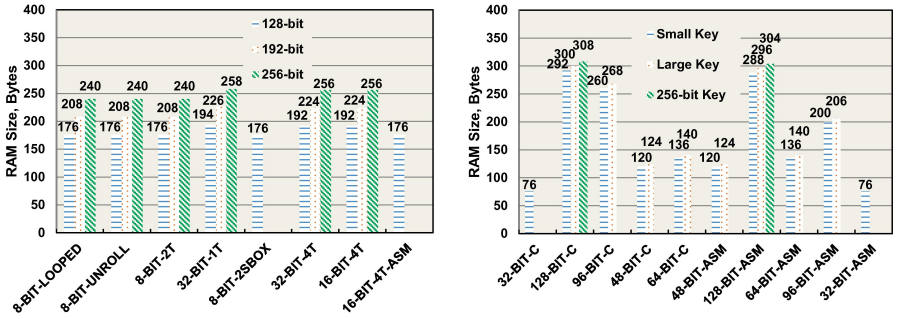

Fig. 4. RAM usage of AES (left) and SPECK (right) (44523)

good overall performance due to its reasonable throughput and small code size, 8-BIT-2SBOX is exceptional for the same reason, and 16-BIT-4T is also good due to its high speed. Figure 4 shows that table-driven C versions of AES consume slightly more RAM (beyond that required to hold the expanded key) because of temporary storage to the stack; however the 16-BIT-4T-ASM version does not require stack as discussed in section 4.3. The ROM size of AES increases slightly for 256-bit key versions because the compiler optimizes away the portion of the AES key schedule that is only valid for 256-bit keys (see, for example, section 5.2 of [7]). Energy consumption generally tracks speed quite well. Slightly varying average current levels for different implementations (not shown here) is a second order effect, confirming the observations of other authors [11].

For SPECK, the fastest C implementation is the 64-BIT block size version at 116 cycles/byte (0.74 μJ/block) and the fastest assembler implementation was the 32-BIT version at 87 cycles/byte (0.31 μJ/block). Of note, all of the assembler implementations are faster than the fastest C implementation. The speed-up from C to assembler is due to two factors. First, in assembly language the multi-precision rotation operations can be implemented more efficiently (as

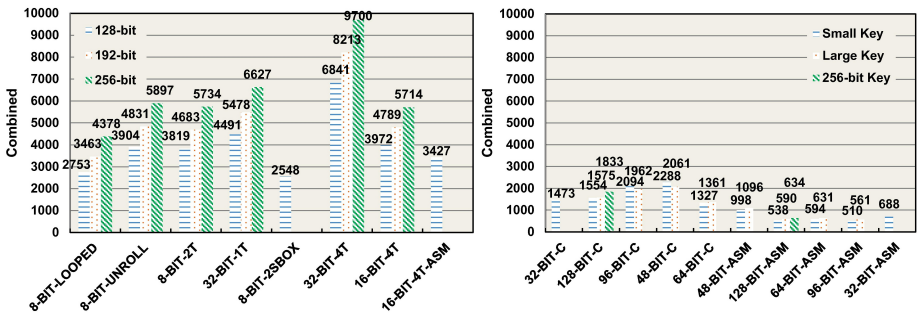

Fig. 5. Combined metric of AES (left) and SPECK (right) (code size cycle count / block size) (44524)

shown in Section 4.4). And second, in the C implementation of the larger block size versions of SPECK (96- and 128-bit) there is inefficient register utilization. The two temporary variables x and y (shown in Listing D) fill 8 of 12 available general purpose registers. This was enough to cause some temporary storage to stack (RAM) with an associated speed penalty. The assembly implementations were able to avoid use of extra RAM and the associated speed penalty.

Larger key sizes for similar versions of SPECK use incrementally more code and energy. However the effect is much less pronounced than in AES. In AES, the number of extra rounds increases from 10 to 14 when going from 128-bit keys to 256-bit keys (a 40% increase) while in SPECK the number of rounds increases from 32 to 34 (just over 6%). Block size has a much stronger impact than key size on both ROM and RAM, as seen in Figure 3 and Figure 4.

Although we chose to limit our in-depth study to the two tailorable block ciphers AES and SPECK, related work provides figures on AES and other block ciphers for the MSP430 platform that can be compared to our results. Related work is summarized in Table 2. In Table 2, implementations are first sorted by block size and then by the combined metric, to facilitate comparison of overall performance between ciphers of similar block size. Note that comparisons such as these can be difficult to interpret due to differing measurement conditions or techniques. Our measurement conditions are stated at the beginning of this section; we have indicated known differences between our approach and the various references as footnotes to Table 2. In cases where the reference indicated that the authors implemented both encrypt and decrypt, but provided only one code size result, the ROM for encrypt only is estimated by dividing the reported ROM by 2.

Table 2. Related work comparison

Algorithm-key size	Block size, bits	Reference	Speed cycle/byte bytes	ROM size, bytes	RAM size,	Energy per blk, μJ	Combined[5]
SPECK-128	128	This work	103	380	288	1.44	538
AES-128	128	This work	132	3147	176	2.18	3427
AES-128	128	[15]	180	2904	NA	NA	4084
RC6-128	128	[11][1]	1120	917	54	19.654	8496
AES-128	128	[11][1]	204	6555	60	3.584	10543
AES-128	128	[19]	172	12400	NA	NA	16662
AES-128	128	[12][2]	1891	2230	19	NA	33225
Camellia-128	128	[11][1]	393	11769	85	6.894	36395
AES-128	128	[18][3]	765	4500	1800	28.16	37652
CLEFIA-128	128	[12][2]	6134	4780	180	NA	237693
SPECK-96	64	This work	96	260	136	0.66	594
Skipjack-80	64	[18][3]	350	3750	40	2.63	20727
XXTEA-128	64	[18][3]	2340	1900	200	17.48	76781
TWINE-128	64	[12][2]	5125	1108	23	NA	90568
Piccolo-128	64	[12][2]	4562	1255	91	NA	95945
Lblock-80	64	[12][2]	5369	1784	13	NA	150751
LED-128	64	[12][2]	21382	1132	41	NA	391892
PRESENT	64	[12][2]	45573	4814	142	NA	3529059
SPECK-64	32	This work	87	174	76	0.31	680

[1] ROM and RAM figures do not include expanded key bytes; all metrics include overhead in Output Feedback mode of operation;

[2] Includes key expansion in encryption speed (separated key expansion speed data not provided in [12])

[3] TelosB Data (8 MHz Clock) used to compute speed from provided timing numbers in [18]

[4] Computed using average current of 2.93 mA, voltage of 2.994 V, and clock of 8 MHz, per [11]

[5] code size (bytes) $\times$ cycle count (cycles/byte) product normalized by block size (bits) [10]

6 Conclusions

We have implemented and measured 24 different variations of AES and 20 different variations of the new lightweight block cipher SPECK on the low power MSP430 platform, in both C and assembler. Many of these implementations represent records for speed and energy efficiency among lightweight and traditional block ciphers on that device, e.g. 132 cycles/byte and 2.18 μJ/block for AES and 103 cycles/byte and 1.44μJ/block for SPECK, both with 128-bit block and key sizes. The 32-bit block size of SPECK with a 64-bit key produced even lower numbers at 87 cycles/byte and 0.31 μJ/block. We provide implementation tactics for both AES and SPECK in both C and assembler for the 16-bit MSP430 platform. Finally, we provide a thorough analysis of measured results across algorithm, implementation strategy, and key size to aid system designers needing to incorporate block ciphers into their designs.

References

1. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: Twine: A lightweight block cipher for multiple platforms. In Selected Areas in Cryptography - SAC 2012. LNCS, vol. 7707, pp. 339-354. Springer, Heidelberg (2013).
2. Shibutani, K., Isobe, R., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo:An ultra-lightweight blockcipher. In Cryptographic Hardware and Embedded Systems - CHES 2011. LNCS, vol. 6917, pp. 342-357. Springer, Heidelberg (2011).
3. Wu, W., Zhang, L.: Lblock: A lightweight block cipher. In Applied Cryptography and Network Security - ACNS 2011. LNCS vol. 6715, pp. 327-344. Springer, Heidelberg (2011).
4. Guo, J., Peyrin, T., Poschmann, A., Robsha, M.: The led block cipher. In Cryptographic Hardware and Embedded Systems - CHES 2011. LNCS, vol. 6917, pp. 326-341. Springer, Heidelberg (2011).
5. Bogdanov, A., Knudson, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In Cryptographic Hardware and Embedded Systems - CHES 2007. LNCS, vol. 4727, pp. 450-466. Springer, Heidelberg (2007).
6. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, June 2013. http://eprint.iacr.org/2013/404
7. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard (AES), 2001. http://www.csrc.nist.gov/publications/ps/ps197/ps-197.pdf.
8. MICAz wireless measurement system. http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf
9. TelosB Platform. http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf
10. Eisenbarth, T., Gong, Z., Guneysu, T., Heyse, S., Indesteege, S., Kerckhof, S., Koeune, F., Nad, T., Plos, T., Regazzoni, F., Standaert, F.-X., van Oldeneel tot Oldenzeel, L.: Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices. In AFRICACRYPT 2012. LNCS, vol. 7374, pp. 172-187. Springer, Heidelberg (2012).

11. Law, Y. W., Doumen, J., Hartel, P.: Survey and benchmark of block ciphers for wireless sensor networks. ACM Transactions on Sensor Networks (TOSN), Volume 2, Issue 1, pp. 65-93. ACM, New York (2006).
12. Cazorla, M., Marquet, K., Minier, M.: Survey and benchmark of lightweight block ciphers for wireless sensor networks. In SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavk, Iceland, 29-31 July, 2013, pages 543-548. SciTePress, 2013.
13. BLOC project performance evaluations, June 2014. http://bloc.project.citi-lab.fr/library.html
14. Guneysu, T.: Implementing AES on a bunch of processors. ECRYPT AES day, Bruges, Belgium, 2012. https://www.cosic.esat.kuleuven.be/ecrypt/AESday/slides/AES-DAY-Gueneysu.pdf
15. Gouvea, C., Lopez, J.: High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller. In Progress in Cryptology LATINCRYPT 2012. LNCS, vol. 7533, pp. 288-304. Springer, Heidelberg (2012).
16. OpenSSL Cryptography and SSL/TLS toolkit. http://www.openssl.org/
17. Didla, S., Ault, A., Bagchi, S.: Optimizing AES for Embedded Devices and Wireless Sensor Networks. In Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities (TridenCOM), Article No. 4, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, Belgium (2008).
18. Lee, J., Kapitanova, K., Son, S. H.: The price of security in wireless sensor networks. Computer Networks vol. 54, no. 17, pp 2967-2978. Elsevier, New York (2010)
19. Hyncica, O., Kucera, P., Honzik, P., Fiedler, P.: Performance Evaluation of Symmetric Cryptography in Embedded Systems. In Proceedings of the 6th International Conference on Intelligent Data Acquistion and Advanced Computing Systems: Technology and Applications, pp. 277-282, Prague (2011)
20. St. Denis, T., LibTomCrypt (source code). http://libtom.org/?page= features&newsitems=5&whatfile=crypt
21. Bos, J., Osvik, D., Stefan, D., Canright, D.: Fast Software AES Encryption. In Proceedings of the 17th international conference on fast software encryption, FSE 2010. LNCS, vol. 6147, pp. 75-93. Springer, Heidelberg (2010).
22. Poettering, B.: AVRAES: The AES block cipher on AVR controllers, 2006. http://point-at-infinity.org/avraes/.
23. Abed, F., List, E., Wenzel, J., Lucks, S.: Differential Cryptanalysis of round-reduced Simon and speck. In Fast Software Encryption, FSE. 2014. To appear in LNCS.
24. Biryukov, A., Roy, A., Velichkov, V.: Differential Analysis of Block Ciphers SIMON and SPECK. In Fast Software Encryption, FSE. 2014. To appear in LNCS.
25. Dinur, I.: Improved Differential Cryptanalysis of Round-Reduced Speck. In Selected Areas in Cryptography (SAC), August 2014. To appear in LNCS.
26. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer, Berlin, 2002.
27. Kretzschmar, U.: AES software support for encryption and decryption. MSP430 Systems. http://www.ti.com/litv/zip/slaa397a
28. MSP430 Family, Instruction Set Summary. http://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf
29. Bernstein, D. J.: Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2005).
30. Gouvea, C.: Authenticated Encryption on the MSP430 (source code). http://conradoplg.cryptoland.net/software/authenticated-encryption-for-the-msp430/

7 Appendix A Code Listings:

Listing A C code for 16-bit AES using four lookup tables T0, T1, T2, and T3:

```
// ===========================================================
void encrypt_aes(uint8_t *s, uint8_t *expanded_key) {
    int round;
    uint16_t *state16;   // 16-bit alias of input array
    uint16_t *key16;     // 16-bit alias of input array
    uint16_t tmp[8];     // 16-bit temporary columns
    uint8_t buf1;
    uint8_t buf2;

    state16 = (uint16_t *)s;
    key16 = (uint16_t *)(expanded_key);

    state16[0] ^= key16[0]; state16[1] ^= key16[1];
    state16[2] ^= key16[2]; state16[3] ^= key16[3];
    state16[4] ^= key16[4]; state16[5] ^= key16[5];
    state16[6] ^= key16[6]; state16[7] ^= key16[7];

    key16 += 8;

    for (round = 1; round < NR; round++) {
        // use the state array to access byte indices to the tables
        tmp[0] = T0[s[0]] ^ T2[s[5]] ^ T1[s[10]] ^ T3[s[15]];
        tmp[1] = T1[s[0]] ^ T3[s[5]] ^ T0[s[10]] ^ T2[s[15]];

        tmp[2] = T0[s[4]] ^ T2[s[9]] ^ T1[s[14]] ^ T3[s[3]];
        tmp[3] = T1[s[4]] ^ T3[s[9]] ^ T0[s[14]] ^ T2[s[3]];

        tmp[4] = T0[s[8]] ^ T2[s[13]] ^ T1[s[2]] ^ T3[s[7]];
        tmp[5] = T1[s[8]] ^ T3[s[13]] ^ T0[s[2]] ^ T2[s[7]];

        tmp[6] = T0[s[12]] ^ T2[s[1]] ^ T1[s[6]] ^ T3[s[11]];
        tmp[7] = T1[s[12]] ^ T3[s[1]] ^ T0[s[6]] ^ T2[s[11]];

        state16[0] = tmp[0] ^ key16[0];
        state16[1] = tmp[1] ^ key16[1];
        state16[2] = tmp[2] ^ key16[2];
        state16[3] = tmp[3] ^ key16[3];
        state16[4] = tmp[4] ^ key16[4];
        state16[5] = tmp[5] ^ key16[5];
        state16[6] = tmp[6] ^ key16[6];
        state16[7] = tmp[7] ^ key16[7];
```

```
        key16 += 8;
    }

    //substitution and shift using Bytes
    // row 0
    s[ 0]  = sbox[s[ 0]];
    s[ 4]  = sbox[s[ 4]];
    s[ 8]  = sbox[s[ 8]];
    s[12]  = sbox[s[12]];
    // row 1
    buf1 = s[1];
    s[ 1]  = sbox[s[ 5]];
    s[ 5]  = sbox[s[ 9]];
    s[ 9]  = sbox[s[13]];
    s[13]  = sbox[buf1];
    // row 2
    buf1 = s[2];
    buf2 = s[6];
    s[ 2]  = sbox[s[10]];
    s[ 6]  = sbox[s[14]];
    s[10]  = sbox[buf1];
    s[14]  = sbox[buf2];
    // row 3
    buf1 = s[15];
    s[15]  = sbox[s[11]];
    s[11]  = sbox[s[ 7]];
    s[ 7]  = sbox[s[ 3]];
    s[ 3]  = sbox[buf1];

    state16[0] ^= key16[0]; state16[1] ^= key16[1];
    state16[2] ^= key16[2]; state16[3] ^= key16[3];
    state16[4] ^= key16[4]; state16[5] ^= key16[5];
    state16[6] ^= key16[6]; state16[7] ^= key16[7];

    return;
}
```

Listing B Assembler code snippet for 16-bit AES round function:

```
// ============================================================
; column 0
mov.b   @R12,R15       ; state[0]
rla.w   R15            ; address into 16-bit T0
mov.w   T0(R15),r4     ; table lookup into temp column
mov.w   T1(R15),r5     ; table lookup into temp column
mov.b   0x5(R12),R15   ; state[5]
```

```
rla.w   R15             ; address into 16-bit T2
xor.w   T2(R15),r4      ; accumulate table lookup into temp column
xor.w   T3(R15),r5      ; accumulate table lookup into temp column
mov.b   0xA(R12),R15    ; state[10]
rla.w   R15             ; address into 16-bit T1
xor.w   T1(R15),r4      ; accumulate table lookup into temp column
xor.w   T0(R15),r5      ; accumulate table lookup into temp column
mov.b   0xF(R12),R15    ; state[15]
rla.w   R15             ; address into 16-bit T3
xor.w   T3(R15),r4      ; accumulate table lookup into temp column
xor.w   T2(R15),r5      ; accumulate table lookup into temp column

; other columns similar (omitted)

; key add
xor.w   @R13+,r4        ; xor key bytes 0-1 with temp bytes
mov.w   r4,0x0(R12)     ; update state bytes 0-1
xor.w   @R13+,r5        ; xor key bytes 2-3 with temp bytes
mov.w   r5,0x2(R12)     ; update state bytes 2-3
xor.w   @R13+,r6        ; xor key bytes 4-5 with temp bytes
mov.w   r6,0x4(R12)     ; update state bytes 4-5
xor.w   @R13+,r7        ; xor key bytes 6-7 with temp bytes
mov.w   r7,0x6(R12)     ; update state bytes 6-7
xor.w   @R13+,r8        ; xor key bytes 8-9 with temp bytes
mov.w   r8,0x8(R12)     ; update state bytes 8-9
xor.w   @R13+,r9        ; xor key bytes 10-11 with temp bytes
mov.w   r9,0xA(R12)     ; update state bytes 10-11
xor.w   @R13+,r10       ; xor key bytes 12-13 with temp bytes
mov.w   r10,0xC(R12)    ; update state bytes 12-13
xor.w   @R13+,r11       ; xor key bytes 14-15 with temp bytes
mov.w   r11,0xE(R12)    ; update state bytes 14-15
```

Listing C Assembler code for multi-word Right Circular Shift by 8 bits:

```
// ============================================================
                ; register contents (Byte numbers):
                ; R4        R5        R6        R7        R9
mov.b r4, r9    ; B0    B1  B2    B3  B4    B5  B6    B7  B0  00
swpb  r9        ; B0    B1  B2    B3  B4    B5  B6    B7  00  B0
swpb  r4        ; B1    B0  B2    B3  B4    B5  B6    B7  00  B0
swpb  r5        ; B1    B0  B3    B2  B4    B5  B6    B7  00  B0
swpb  r6        ; B1    B0  B3    B2  B5    B4  B6    B7  00  B0
swpb  r7        ; B1    B0  B3    B2  B5    B4  B7    B6  00  B0
xor.b r5, r4    ; B1^B3 00  B3    B2  B5    B4  B7    B6  00  B0
xor   r5, r4    ; B1    B2  B3    B2  B5    B4  B7    B6  00  B0
xor.b r6, r5    ; B1    B2  B3^B5 00  B5    B4  B7    B6  00  B0
```

```
xor   r6, r5    ; B1    B2  B3    B4  B5    B4  B7    B6  00  B0
xor.b r7, r6    ; B1    B2  B3    B4  B5^B7 00  B7    B6  00  B0
xor   r7, r6    ; B1    B2  B3    B4  B5    B6  B7    B6  00  B0
xor.b r9, r7    ; B1    B2  B3    B4  B5    B6  B7^00 00  00  B0
xor   r9, r7    ; B1    B2  B3    B4  B5    B6  B7    B0  00  B0
```

Listing D C language code for SPECK-128:

```
// ===============================================================
void encrypt_speck(uint8_t * pointer, uint8_t * expanded_key) {
    uint64_t *p64 = (uint64_t*)pointer;
    uint64_t *k64 = (uint64_t*)expanded_key;
    uint64_t  y;
    uint64_t  x;

    // Copy values to be encrypted into x and y
    y = p64[0];
    x = p64[1];

    for (int round = 0; round < SPECK_T; round++) {
        uint64_t t;
        uint8_t y8;

        // RCS, addition with y, and key addition
        t = x >> 8;
        x = x << 56;
        x = x + t + y;
        x = x ^ k64[round];

        // LCS and XOR with x
        t = y << 3;
        y8 = (uint8_t)(y >> 56);
        y8 >>= 5;
        y = (t | y8) ^ x;
    }

    // Copy encrypted values back into ram
    p64[0] = y;
    p64[1] = x;
}
```