The BIG Cipher: Design, Security Analysis, and Hardware-Software Optimization Techniques

Anthony Demeri, Thomas Conroy, Alex Nolan and William Diehl

The Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA
Email:{demeri, tconroy, ajn96, wdiehl} @vt.edu

Abstract. Secure block cipher design is a complex discipline which combines mathematics, engineering, and computer science. In order to develop cryptographers who are grounded in all three disciplines, it is necessary to undertake synergistic research as early as possible in technical curricula, particularly at the undergraduate university level. In this work, students are presented with a new block cipher, which is designed to offer moderate security while providing engineering and analysis challenges suitable for the senior undergraduate level. The BIG (Block) (Instructional, Generic) cipher is analyzed for vulnerability to linear cryptanalysis. Further, the cipher is implemented using the Nios II microprocessor and two configurations of memory-mapped hardware accelerators, in the Cyclone V FPGA on the Terasic DE1 System-on-chip (SoC). Three distinct implementations are realized: 1) Purely software (optimized for latency), 2) Purely hardware (optimized for area), and 3) A hardware-software codesign (optimized for throughput-to-area ratio). All three implementations are evaluated in terms of latency (encryption and decryption), throughput (Mbps), area (ALMs), and throughput-to-area (TP/A) ratio (Mbps/ALM); all metrics account for a fully functional Nios II, 8 kilobytes of on-chip RAM, Avalon interconnect, benchmark timer, and any hardware accelerators. In terms of security, we demonstrate recovery of a relationship among 12 key bits using as few as 16,000 plaintext/ciphertext pairs in a 6-round reduced round attack and reveal a diffusion rate of only 43.3% after 12 rounds. The implementation results show that the hardware-software codesign achieves a 67× speed-up and 37× increase in TP/A ratio over the software implementation, and 5× speed-up and 5× increase in TP/A ratio compared to the hardware implementation.

Keywords: Cryptography**,** block cipher, linear cryptanalysis, FPGA, Nios II, Cyclone V, design optimization, undergraduate, education.

1 Introduction

In 2013, over 2.5 quintillion (10^{18}) bytes of data were created each day, encompassing many fields of personal data, such as social media postings, mobile banking information, cellular data, and purchase history [1]. Naturally, as the number of internet users continues to grow, totaling over 2.5 billion worldwide users as of 2016 [2], an increase in personal data generation is logical. In order to provide a means of secure

transmission for the growing quantity of sensitive data, a system of optimized encryption and decryption (secure from brute-force attacks, cryptanalysis, and physical implementation attacks) is mandated.

New cryptographic algorithms are historically introduced through government and industry collaboration (e.g., [5]), government-sponsored competitions and standardization efforts (e.g., [6],[7],[8]) or non-governmental, multi-national competitions (e.g., [9]). However, the core competencies and skill sets required to develop robust cryptographic algorithms, which are secure and efficient in hardware and software, are diverse, academically challenging, and require significant time and effort to mature.

Although the above skills can be developed piecemeal (e.g., through separate mathematics, engineering, and computer science curricula), a *unified educational approach*, which combines all of these disciplines as early as possible, could facilitate future implementation of cryptographic standards, which are *secure* and *efficient* from inception.

To demonstrate the "art of the possible," undergraduate computer engineering students implement a prototype educational block cipher in software, hardware, and an optimized hardware-software codesign approach. Implementations use the Nios II soft core microprocessor and are built with the Intel Quartus Prime Lite design suite, including Platform Designer, and the Nios II Software Build Tools (SBT) by Eclipse. Designs are instantiated in the Cyclone V FPGA, as part of the Terasic DE1 System-on-Chip (SoC). Implementation results are characterized in terms of latency (total encryption and decryption clock cycles), throughput (Mbps), memory (bytes), FPGA area (adaptive logic modules, or ALMs), and throughput-to-area (TP/A) ratio (Kbps/ALM).

We additionally perform a partial security analysis of the cipher by analyzing the confusion and diffusion of the cipher and using linear cryptanalysis to uncover a relationship of key bits. In this analysis, we uncover a relationship among 12 key bits by using plaintext/ciphertext pairs, which, if used to create other approximations, would allow more bits of the key to be recovered. We additionally argue the impracticality of key recovery for the 12-round or 18-round block cipher.

2 The BIG Cipher Specification

The BIG (Block) (Instructional, Generic) cipher is a block cipher, which operates on 128-bit blocks and allows for the encryption of plaintext or decryption of ciphertext, using a 128-bit secret key. This cipher requires a specific number of encryption or decryption rounds to be used, based on the desired level of security, such as 12 rounds for moderate security or 18 rounds for high security. This cipher is designed as a Feistel structure [3], which implies separate operations on 64-bit left (high) and right (low) operands. This approach lends itself to parallelization, provided appropriate resources are available. The Feistel approach is motivated by previous block cipher designs, such as DES [5], SIMON and SPECK [10], and TWINE [11].

The BIG cipher is designed as an educational block cipher to prepare computer engineers for the challenges of implementing symmetric cryptography in hardware, software, and hardware-software codesign. For example, it contains 4-bit non-linear substitutions (S-Boxes), a composite permutation (including rotation by r bits, where $2^n \nmid$

r), and round constant addition (which is not aligned with bytes, half-words, or words), which increase complexity in software and favor a hardware approach. Additionally, substitutions and constant generations are derived through finite field arithmetic (in GF(2^4) and GF(2^8), respectively), which provide the possibility of low-area hardware implementations. Finally, the 128-bit block and key size of the cipher is designed to provide a nominal example of the implementation costs of a high-security block cipher, and to highlight the realistic communication challenges of integrating hardware accelerators with large data words (e.g., an Avalon memory-mapped slave device) with master peripherals on smaller data buses (e.g., a 32-bit Nios II processor).

2.1 Basic Operation

Strings of plaintext (PT), ciphertext (CT), and secret key (K) are 128-bits (16 bytes), where the order of a string W, consisting of PT, CT, or K, is represented as:

$$W = W_0 \,||\, W_1 \,||\, W_2 \,||\, ... \,||\, W_{15}, \tag{1}$$

where W_i is the *i*th byte of the relevant string, and W is further split into an upper half W_{H} and a lower half W_{L}, where $W = W_{\mathrm{H}} \,||\, W_{\mathrm{L}}$, and both halves are 64 bits (8 bytes long). The BIG encryption operation 'ENC' is represented as Algorithm 1 and displayed in Figure 1.

Algorithm 1. The BIG cipher encryption algorithm

$CT = ENC(PT, K)$
  $W_H = PT[127:64]$
  $W_L = PT[63:0]$
  $for\ i = 0\ to\ NUM\_ROUNDS - 1$
    $W_H = W_H \oplus RK_{H_i}$
    $W_H = SubBytes(W_H)$
    $W_L = AddRoundConstant(W_L, i)$
    $W_L = W_L \oplus RK_{L_i}$
    $W_L = Perm_2(Perm_1(W_L))$
    $tmp = W_H$
    $W_H = W_H \oplus W_L$
    $W_L = tmp$
  $CT = W_H \,||\, W_L$
  $return\ CT$

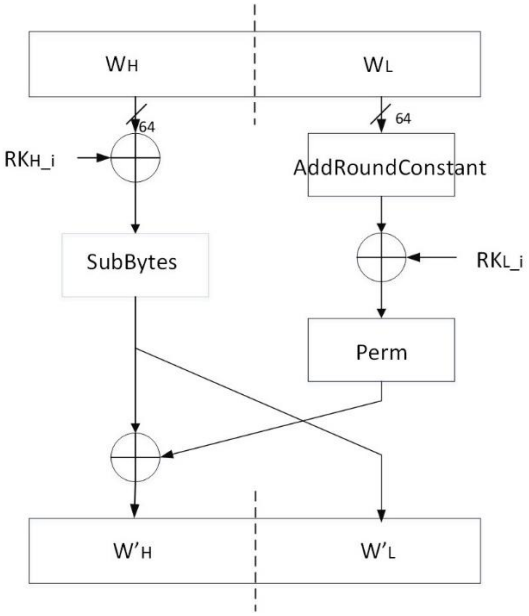

Figure 1. The BIG encryption operation 'ENC', where the flow of data is logically represented from top to bottom and encompasses the ith encryption round.

Similarly, the BIG decryption operation 'DEC' is represented as Algorithm 2 and displayed in Figure 2.

Algorithm 2. The BIG cipher decryption algorithm

$PT = DEC\,(CT, K)$
  $W_H = CT[127:64]$
  $W_L = CT[63:0]$
  $for\ i = NUM\_ROUNDS - 1\ down\ to\ 0$
    $tmp = W_L$
    $W_L = W_H \oplus W_L$
    $W_L = {Perm_1}^{-1}({Perm_2}^{-1}(W_L))$
    $W_H = InvSubBytes(tmp)$
    $W_H = W_H \oplus RK_{H_i}$
    $W_L = W_L \oplus RK_{L_i}$
    $W_L = AddRoundConstant(W_L, i)$
  $PT = W_H \,||\, W_L$
  $return\ PT$

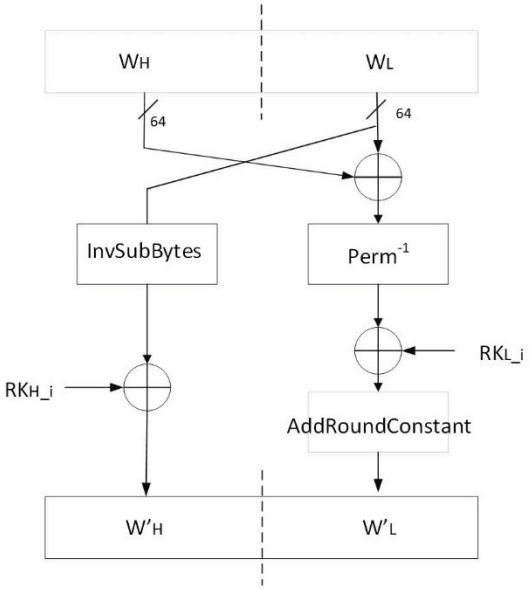

Figure 2. The BIG decryption operation 'DEC', where the flow of data is logically represented from top to bottom and encompasses the ith decryption round.

Given the high-level encryption and decryption algorithms, we present the individual implementations for each submodule.

2.2 SubBytes Specification

The SubBytes operation is represented as a string of sixteen 4-bit S-Boxes, where $SubBytes(Y)$ is defined as:

$$S = SubBytes(Y) = S_0 \,||\, \dots \,||\, S_{15} = Sbox(Y_0) \,||\, \dots \,||\, Sbox(Y_{15}), \tag{2}$$

where S_i and Y_i are 4-bit nibbles of S and Y, respectively, and $i = 0, 1, 2, \dots, 15$.

$Sbox$ is defined as

$$Sbox(Y) = AY^{-1} + b, \tag{3}$$

where the binary matrix A is represented as

$$A = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0, \end{matrix}$$

Y^{-1} is the multiplicative inverse of Y modulo $P(x) = x^4 + x + 1$, and b is the column vector constant $(0\ 0\ 1\ 1)^T$. The resulting substitution is shown in Table 1.

Table 1. S-Box Lookup Table

Y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S	C	9	D	2	5	F	3	6	7	E	0	1	A	4	B	8

2.3 InvSubBytes Specification

The InvSubBytes operation is represented as a string of sixteen 4-bit S-Boxes, where $InvSubBytes(v)$ is defined as:

$$\sigma = InvSubBytes(v) = InvSbox(v_0) \,||\, \dots \,||\, InvSbox(v_{15}), \tag{4}$$

where σ_i and v_i are 4-bit nibbles of σ and v, respectively, and $i = 0, 1, 2, \dots, 15$.

$InvSbox$ is defined as

$$InvSbox(v) = [A^{-1}(v + b)]^{-1}, \tag{5}$$

where the binary matrix A^{-1} is represented as

$$A^{-1} = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0, \end{matrix}$$

$[\,\cdot\,]^{-1}$ is the multiplicative inverse of $[\,\cdot\,]$ modulo $P(x) = x^4 + x + 1$, and b is the column vector constant $(0\ 0\ 1\ 1)^T$. The resulting substitution is shown in Table 2.

Table 2. Inverse S-Box Lookup Table

V	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
σ	A	B	3	6	D	4	7	8	F	1	C	E	0	2	9	5

The choices of S-Boxes used in SubBytes and InvSubBytes are motivated by inversions in a finite field modulo an irreducible polynomial (e.g., [12]), and S-Box constructions presented in [13], [4], and [11]. Additionally, the choice of a 4-bit S-Box purposely increases the complexity required for software implementations, since operations on 4-bit values cannot be efficiently performed using simple byte operations. Finally, the use of a 4-bit S-Box could facilitate future research involving side-channel resistant countermeasures, based on [14], [15].

2.4 AddRoundConstant Specification

The AddRoundConstant operation adds a 7-bit round constant c_i to bits 20:14 of a 64-bit argument, where 'addition' is finite field addition (XOR), and $AddRoundConstant(X, i)$ is defined as:

$$Y = AddRoundConstant(X, i) = (X_{63:21}) \,||\, (X_{20:14} \oplus c_i) \,||\, (X_{13:0}), \tag{6}$$

where $c_i = 0x5A \cdot 2^i$, multiplication is performed modulo $P(x) = x^8 + x^4 + x^3 + x + 1$ (i.e., the AES polynomial [12]), and the computed round constants for rounds 0 to 17 are shown in Table 3.

Table 3. Round Constants (in 7-bit hexadecimal)

i	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
c_i	5A	34	73	66	57	35	71	62	5F	25	51	22	5F	3E	67	4E	07	15

Round constants are generated using a similar strategy to that described in [11]. The addition of a 7-bit round constant, which does not fall on byte boundaries, is expected to be more complex in software (e.g., requiring multiple shift and concatenation operations per round iteration), but easier to perform in hardware, and is designed to influence the student-designer's decision calculus for allocation of primitives in hardware or software components.

2.5 Permutation Specification

In the BIG cipher, there are two permutation operations: $Perm_1(X)$ and $Perm_2(X)$, as well as their inverse permutations. Specifically, $Perm_1$ swaps the positions of neighboring 16-bit words, where:

$$Y = Perm_1(X) = Perm_1(X_0 \,||\, X_1 \,||\, X_2 \,||\, X_3) = X_1 || \, X_0 || \, X_3 || \, X_2, \tag{7}$$

and each X_i is 16-bits. Note that ${Perm_1}^{-1}(X) = Perm_1(X)$.

$Perm_2$ is a 43-bit right circular shift on a 64-bit operand, where:

$$Y = Perm_2(X) = Perm_2(X_{63:0}) = X_{42:0} \,||\, X_{63:43}, \tag{8}$$

and the inverse of $Perm_2$ is a 43-bit left circular shift on a 64-bit operand, where:

$$v = Perm_2^{-1}(\xi) = Perm_2^{-1}(\xi_{63:0}) = \xi_{20:0} \,||\, \xi_{63:21} \tag{9}$$

In particular, $Perm_2$ is inspired by the 61-bit rotation used in the key scheduling algorithm of PRESENT [13]. Previous research has shown that rotations by values that are not divisible by 2^n are difficult to perform in software, but trivially performed in hardware [16].

2.6 RoundKey Encryption Specification

The RoundKey scheduling algorithms are designed for flexible implementation, at the cost of security. Specifically, round keys are generated using only a combination of Feistel swaps, and half word-wise permutations (i.e., $Perm_1$). This causes round keys to be periodic; they repeat every six rounds. This informs the choice of number of rounds as 12 (6·2) or 18 (6·3); other choices are possible, however, a BIG cipher constructed with only 6 rounds is unable to withstand analytic attacks, and a BIG cipher with more than 18 rounds rapidly sacrifices performance.

Since the round key generation algorithms do not introduce confusion (e.g., S-Boxes) or whitening (e.g., key and data combination), there are likely a large number of weak keys. One example would be 0^{128}; others could be determined analytically.

However, one benefit of this simple round key generation scheme is the ease of "on the fly" round key scheduling for both encryption and decryption algorithms. By contrast, decryption round keys cannot be generated on-the-fly for algorithms such as AES; the designer must pre-compute round keys, which levies a significant tax on performance (especially for short messages with non-repeating secret keys), and memory to store round keys, e.g., 128 bits · 10 rounds = 160 bytes.

For the *i*th round of encryption, a round key is generated according to Algorithm 3, and as shown in Figure 3.

Algorithm 3. Round Key Generation (Encryption) Algorithm

$RK_i = generateRoundKey_{enc}(RK_{i-1})$
  $if\ i = 0$
    $RK_{i-1} = K\ //\ K\ is\ the\ original\ key$
  $RK_{H_{i-1}} = RK_{i-1}[127:64]$
  $RK_{L_{i-1}} = RK_{i-1}[63:0]$
  $tmp = Perm_1(RK_{L_{i-1}})$
  $RK_{L_i} = tmp \oplus RK_{H_{i-1}}$
  $RK_{H_i} = tmp$
  $return\ RK_{H_i} \,||\, RK_{L_i}$

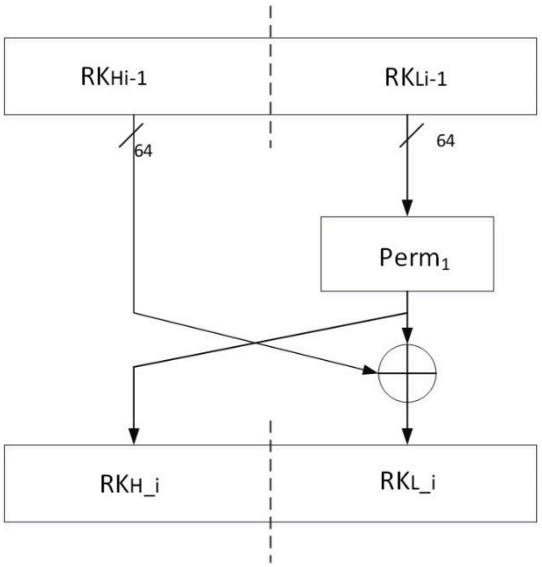

Figure 3. The Round Key Generation (encryption) operation '$generateRoundKey_{enc}$', where the flow of data is logically represented from top to bottom and encompasses the *i*th round key generation (encryption).

2.7 RoundKey Decryption Specification

For the *i*th round of decryption, a round key is generated according to Algorithm 4, and as shown in Figure 4.

Algorithm 4. Round Key Generation (Decryption) Algorithm

$RK_i - 1 = generateRoundKeydec(RK_i)$
$if\ i = NUM\_ROUNDS - 1$
$RK_i = K\ //\ K\ is\ the\ original\ key$
$RK_{H_i} = RK_i[127:64]$
$RK_{L_i} = RK_i[63:0]$
$tmp = Perm_1(RK_{H_i})$
$RK_{H_{i-1}} = RK_{H_i} \oplus RK_{L_i}$
$RK_{L_{i-1}} = tmp$
$return\ RK_{H_{i-1}} || RK_{L_{i-1}}$

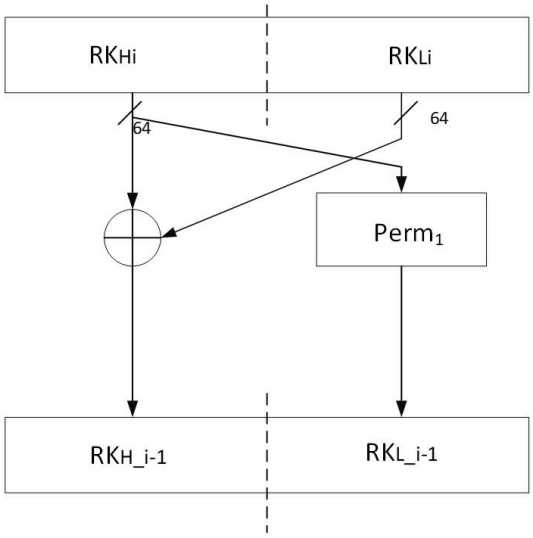

Figure 4. The Round Key Generation (decryption) operation '$generateRoundKey_{dec}$', where the flow of data is logically represented from top to bottom and encompasses the *i*th round key generation (decryption).

3 Security Analysis

In this section, we discuss diffusion and confusion, as defined by Shannon [18] and perform linear cryptanalysis on the six-round version of the BIG cipher. The linear analysis performed in this section is based on the work of Matsui on linear cryptanalysis of DES [17]. A linear approximation using six active S-Boxes was created manually,

leading to recovery of a relationship among 12 key bits using ~16,000c plaintext/ciphertext pairs, where c is a small positive number. This method, if used to create other approximations, would allow more of the key to be recovered.

3.1 Diffusion and Confusion

Diffusion in the BIG cipher is provided through the repeated permutations, which serve to mix up the state of the cipher and Feistel swaps. Confusion in the cipher is provided through the introduction of round key bits from the key scheduler on both sides of the Feistel structure and nonlinear S-Boxes.

Experimental results showing the number of bits in the ciphertext that flip due to a single bit change in the plaintext (for diffusion) or the key (for confusion) are shown in Table 4. Ideal (strong) diffusion and confusion are an average of 64-bit flips.

Table 4. Number of bit flips on average over 1.28 million tests for each property in 6, 12, and 18 rounds of the BIG cipher.

# of Rounds	Diffusion Bit Flips	Confusion Bit Flips
6	22.09	44.39
12	55.43	63.69
18	64.28	64.13

From the table it is obvious that the 6 round version exhibits weak diffusion and confusion, while the versions with more rounds offer stronger diffusion and confusion. Although the 12 round version offers improved diffusion and generally strong confusion, the fact that only 43% of bits flip due to diffusion on average suggests vulnerability to linear cryptanalysis. In contrast, the 18 round version offers near optimal diffusion and confusion in this test.

3.2 Linearized S-Box Approximations

To implement a linear attack against the algorithm, a linear approximation of the algorithm starts with approximating S-Boxes. This is accomplished by relating certain "active" input and output bits of an S-Box using XOR like in (10):

$$x_3 \oplus x_1 = y_2 \oplus y_1 \tag{10}$$

where $x = x_3||\, x_2||\, x_1||\, x_0$ is the 4-bit input to the S-Box and $y = \ y_3||\, y_2||\, y_1||\, y_0 = Sbox(x)$. In (10), input bits 3 and 1 are active, and output bits 2 and 1 are active. Approximations like these will be correct only for some S-Box input combinations. Those that are correct for many (or wrong for many) have *high bias.*

The S-Box definition presented in Section 2.2 has the following Normalized Linear Approximation Table shown in Table 5.

Table 5. S-Box Normalized Linear Approximation Table. The hexadecimal number identifying the rows represents which S-Box input bits are active (6 would be middle two bits active in the approximation). Similarly, the hexadecimal number identifying the columns represent which output bits are active. With 16 possible inputs to an S-Box in BIG, each approximation would ideally (from an algorithm designer's point of view) be correct for 8 inputs. The values in the table represent the deviation from 8 that occur for the 16 input combinations.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0:	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1:	0	-2	0	2	0	2	-4	2	0	-2	0	2	0	2	4	2
2:	0	0	0	0	-4	0	0	-4	-2	-2	2	2	-2	2	-2	2
3:	0	2	0	-2	0	2	0	-2	2	-4	-2	-4	-2	0	2	0
4:	0	0	2	-2	0	0	2	-2	0	0	-2	2	4	4	2	-2
5:	0	2	2	4	-4	2	2	0	0	2	-2	0	0	-2	2	0
6:	0	0	-2	2	0	-4	2	2	-2	-2	-4	0	-2	2	0	0
7:	0	-2	-2	4	0	-2	-2	-4	2	0	0	-2	2	0	0	-2
8:	0	-2	0	-2	-2	0	-2	0	0	2	-4	-2	2	0	-2	4
9:	0	0	4	0	-2	-2	-2	2	0	-4	0	0	2	-2	-2	-2
a:	0	-2	4	2	2	0	2	0	-2	0	2	-4	0	2	0	2
b:	0	4	0	0	-2	-2	-2	2	2	2	2	-2	0	4	0	0
c:	0	2	2	0	2	0	-4	-2	-4	2	-2	0	-2	0	0	-2
d:	0	0	-2	-2	-2	-2	0	0	-4	0	2	-2	2	-2	4	0
e:	0	2	2	0	2	-4	0	-2	2	0	0	2	0	-2	2	4
f:	0	4	-2	2	2	2	0	0	-2	-2	0	0	4	0	-2	2

3.3 Approximating 6 Rounds of BIG

For this section, let the plaintext PT be expressed in bits as:

$$PT = PT_0 \,||\, PT_1 \,||\, \ldots \,||\, PT_{127} \tag{11}$$

Similarly let the key K be expressed in bits as:

$$K = K_0 \,||\, K_1 \,||\, \ldots \,||\, K_{127} \tag{12}$$

Using Table 5 to select approximations with high bias (those that have a value with a large absolute magnitude), the approximation:

$$PT_2 \oplus PT_4 \oplus PT_6 \oplus PT_7 \oplus PT_{70} \oplus PT_{75} \oplus W_0^2 = 0, \tag{13}$$

where W_0^2 is the first bit of W after the round $i = 2$. This approximation was created by approximating six "active" S-Boxes, each chosen for its ability to use a high bias of absolute magnitude 0.25. This is calculated by dividing the number in Table 5 for that

approximation by 16 [19]. Using the "Piling-up Lemma" from Matsui, the bias of the BIG approximation is calculated as:

$$\varepsilon_{total} = 2^{n-1} \prod_{i=1}^{n} \varepsilon_i = 2^5 2^{-12} = 2^{-7} \tag{14}$$

In this case with six S-Boxes, $n = 6$ and each $\varepsilon_i = 0.25 = 2^{-2}$. Also, by Matsui, the number of plaintext/ciphertext pairs to use this approximation with confidence of accuracy is

$$\varepsilon_{total}{}^{-2} = (2^{-7})^{-2} c \approx 16{,}000c \tag{15}$$

where c is a small positive number [17].

3.4 Uncovering Key Bits

The approximation presented in the previous section can be used to uncover a relationship between the key bits. Specifically, the approximation can be used to discover the value of this expression:

$$K_{16\ldots19} \oplus K_{59\ldots62} \oplus K_{123\ldots126} \tag{16}$$

This expression is determined by which key bits are used in the algorithm to decrypt from the ciphertext to the S-Box directly after W_0^2.

By setting $K_{16\ldots19} \oplus K_{59\ldots62} = 0$ in a candidate key and going through the sixteen possible values of $K_{123\ldots126}$, the linear approximation will have the largest (in magnitude) bias when the value for the expression in (16) is correct [17].

This method was tested with varying number of plaintext/ciphertext pairs, and the results are shown in Table 6.

Table 6. Linear attack results. Each # of plaintext/ciphertext pairs was tested on 10,000 keys. The third column is the number of keys where the attack correctly recovered all 4 bits in (16)

c	# of Plaintext/Ciphertext pairs	# of Correct Recoveries (out of 10,000 keys)	% Correctly Recovered
½	8,000	2,829	28.29%
1	16,000	4,293	42.93%
2	32,000	5,717	57.17%
4	64,000	6,824	68.24%
8	128,000	7,871	78.71%
16	256,000	8,347	83.47%
32	512,000	8,614	86.14%

The accuracy of the attack increases with an increased number of plaintext/ciphertext pairs. Similar approximations to (13) could be found for 6 rounds, leading to more relations like (16) being discovered. However, extending this attack to 12 rounds or 18 rounds would require more active S-Boxes, which would mean the required number of plaintext/ciphertext pairs would increase, making this attack infeasible for real world use.

For example, assuming the analysis was extended to 12 rounds, this would require at least 6 more S-Box approximations (as an unrealistically *bare* minimum). Assuming these approximations have a bias of magnitude 0.25 (the maximum available in Table 5), that would mean the bias of the full cipher approximation would decrease (using the Piling-up Lemma) by a factor of 2^{-6}. The total number of required plaintext/ciphertext pairs would then increase by a factor of 2^{12} to a minimum of 67.1 million.

4 Design Implementation and Optimization

When implementing a proposed algorithm, there are design variations which can be exploited in order to optimize the implementation toward a specific metric. We present three design optimization techniques: a pure software implementation, optimized for minimum clock cycles; a pure hardware implementation, optimized for minimum ALMs used; and hardware-software codesign implementation, optimized for maximum throughput/area (TP/A). When implementing these designs, the Terasic DE1-SoC hardware is used to implement the base processor (the Nios II (e) Classic) for all implementations, while the Eclipse SBT is used to implement the software interface (written in C code). Additional constraints within the design include: 8KB of Intel On-Chip RAM, the JTAG UART, clock and reset modules, a timer module, and the respective interconnect for all components, which mandates a lower-limit of 676 ALMs (the minimum ALM usage of the software design implementation, and base ALM usage for the hardware design implementation and hardware-software codesign implementation).

The base interface to the Nios II processor is shown in Figure 5. Metrics for all implementations include the area, in ALMs, required to instantiate all of the below hardware, plus any additional hardware accelerators for the hardware and hardware-software codesigned versions. Additionally, all implementations are for the 12 round BIG cipher version, and use a single clock domain, operating on the 50MHz clock source generated by the Terasic DE1-SoC.

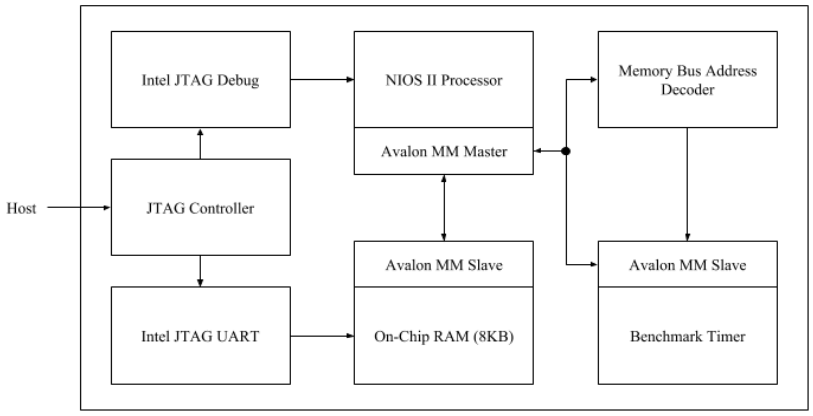

Figure 5. The base Qsys (Platform Designer) interface for the BIG Cipher implementation, which bridges the Nios II Processor (HW) to the respective implementation (SW, HW, or HW/SW).

4.1 Software Design Implementation

In our pure software implementation, minimizing the required encryption and decryption clock cycles mandates a tactful approach toward C programming, including such techniques as: loop unrolling, function in-lining, compiler optimizing, intelligent memory accessing, intelligent round key generation, and smarter S-Box utilization.

Given a minimum clock cycle optimization target, and assuming that there is sufficient instruction memory available to allow reduced code density to provide a clock cycle reduction, we can increase the amount of instruction memory consumed to decrease the total clock cycles required. Specifically, we explicitly unroll all loops and in-line all function calls; thus we remove the branch and jump overheads, which reduces the overall required clock cycles for the implementation.

In order to further reduce the required encryption and decryption clock cycles, memory accesses are performed in an intelligent manner, based on the sequential nature of the stored block data (e.g., plaintext/ciphertext, key, round keys, and S-Boxes). Specifically, when multiple bytes of data are known to be stored sequentially, rather than reading or writing individual bytes (which requires a single clock cycle), the data is read as either a 16-bit or 32-bit integer (which requires a single clock cycle) and interpreted accordingly, effectively 'batching' multiple memory accesses into a single access and operating on the bytes in parallel.

By examining the BIG cipher's specification, we note that a round-key generation has a period of 6 rounds; thus, these six round keys can be computed a single time, stored in memory, and utilized for the remainder of an encryption/decryption process, thus amalgamating multiple operations into a single memory access. Using a similar principle, the 4-bit S-Box 'nibbles' are also recomputed to work on 'bytes', which allows for a further compaction of operations.

Finally, we employ the use of high compiler optimizations, such as the "#pragma GCC optimize" approach, which optimizes the compilation for minimum clock cycles.

We present the results of the optimization techniques in Table 7:

Table 7. Software Design Implementation Results

	Encrypt Cycles	Decrypt Cycles	Total Cycles	Throughput (Mbps)	.text (bytes)	ALMs	Mem. Bits	TP/A Ratio (Kbps/ALM)
Optimized Design	26,667	26,734	53,401	0.24	6,084	676	76,800	0.355

4.2 Hardware Design Implementation

In our pure hardware implementation, which is optimized for minimum area (measured as ALMs), a multi-slice partition is used to split each encryption/decryption round into 16 slices of 4-bit nibbles, which reduces the number of required S-Boxes, and thus overall ALMs. This design is shown in Figure 6.

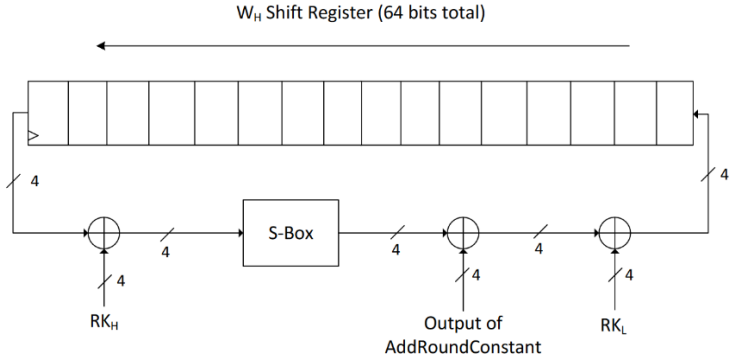

Figure 6. Multi-slice Partition design (for encryption). Inputs from RK_H, the output of $AddRoundConstant$, and RK_L are specifically chosen 4-bit slices of those values for each iteration. After 16 iterations of this, the full W_H word can be calculated using a single S-Box.

By observing the parallel structure of the BIG cipher, and by utilizing a shift register for the upper half of the plaintext/ciphertext word (where no permutation of data exists), we implement the multi-slice design. For clarity, we display this design as a Platform Designer block in Figure 7, an algorithmic-state-machine (ASM) in Figure 8, and as a block diagram in Figure 9; we display results in Table 8.

Table 8. Hardware Design Implementation Results

	Encrypt Cycles	Decrypt Cycles	Total Cycles	Throughput (Mbps)	.text (bytes)	ALMs	Mem. Bits	TP/A Ratio (Kbps/ALM)
Optimized Design	2,040	2,040	4,080	3.138	2,588	1,170	76,800	2.68

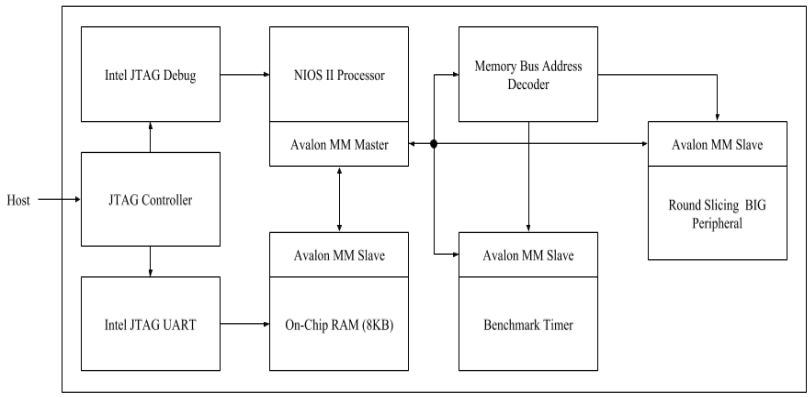

Figure 7. The pure HW Qsys (Platform Designer) interface for the BIG Block Cipher's implementation, which bridges the Nios II Processor (HW) to the HW Implementation (Round Slicing BIG Peripheral)

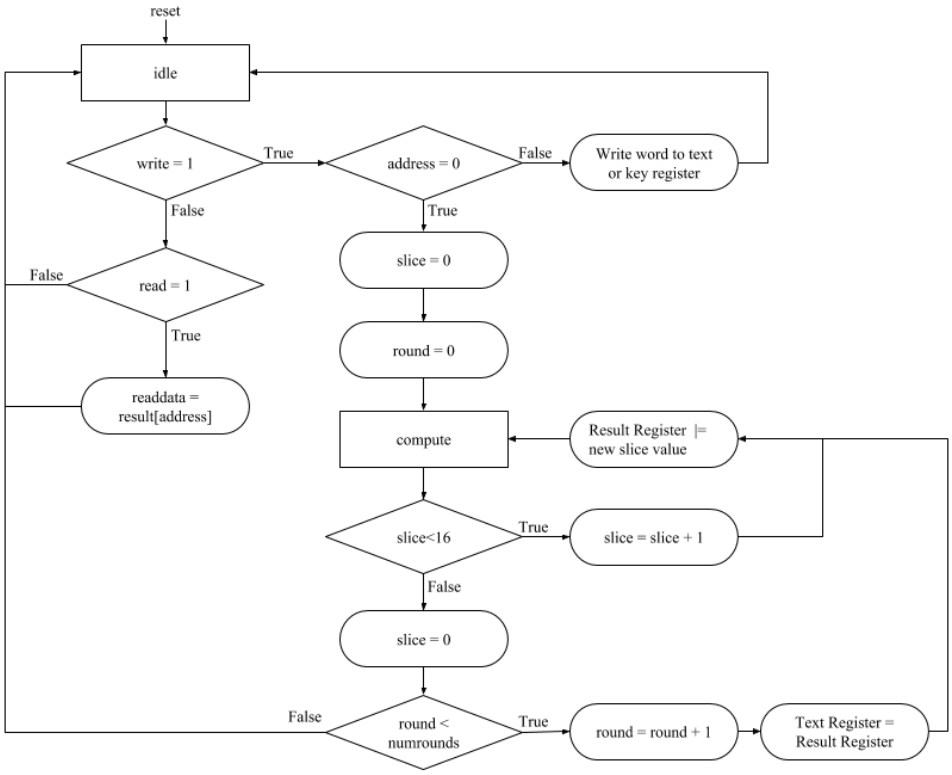

Figure 8. The algorithmic-state-machine (ASM) of the pure HW implementation, which defines the behavior of the multi-slice approach.

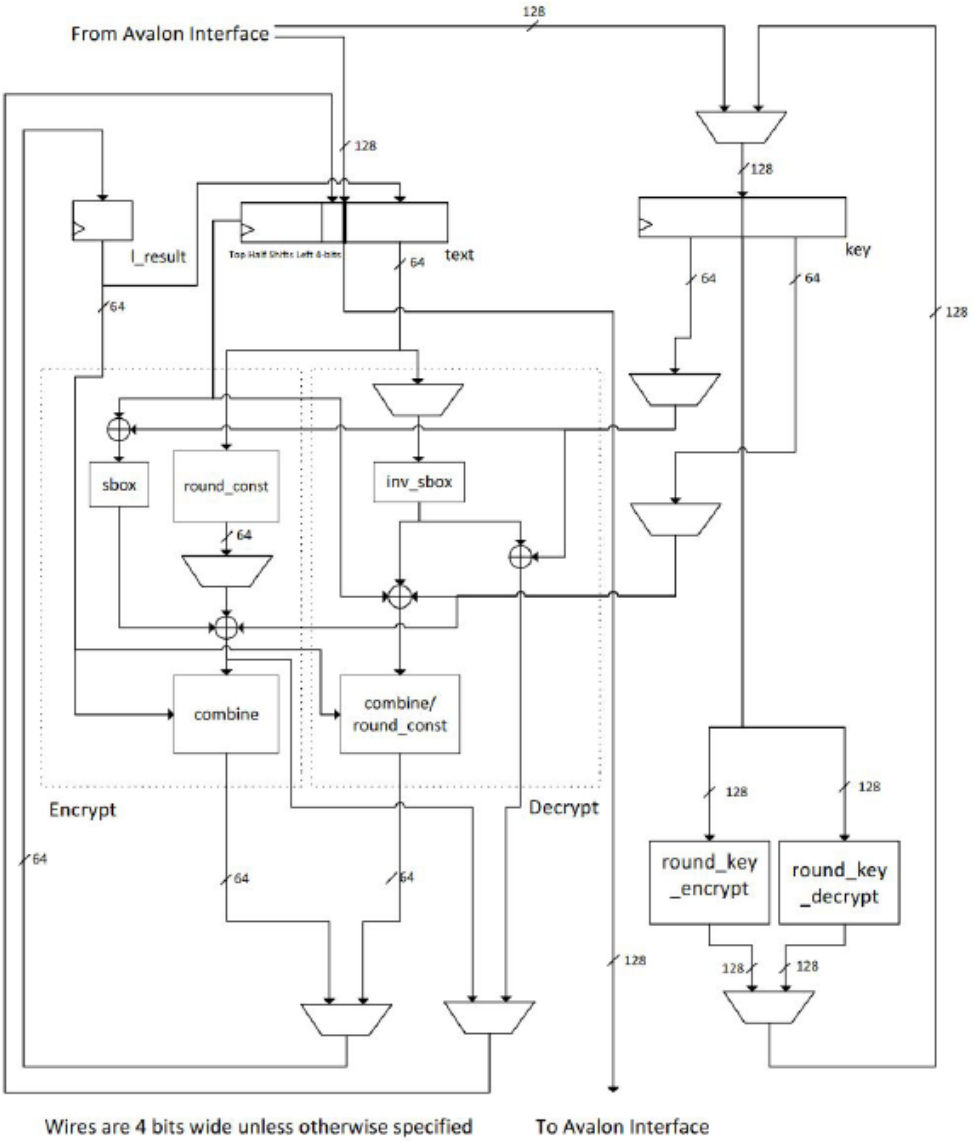

Figure 9. The block diagram representation of the pure hardware design implementation's minimum area optimization.

4.3 Hardware/Software Codesign Implementation

Having demonstrated optimization techniques for both a pure software and a pure hardware implementation, we present a hardware-software codesign implementation, which is optimized around maximizing the TP/A ratio by making effective hardware-software

codesign decisions. Initially, we considered three potential design options: Nios II custom encryption and decryption instructions; a parallelized, pure-hardware implementation with an appropriate finite state machine (FSM); and a serialized, pure hardware implementation, using software for control logic.

Nios II Custom Instruction. We observed the potential capability of the custom instruction to provide a single-cycle, tightly-coupled hardware operation upon some operands; as such, we developed an appropriate series of encryption and decryption custom instructions. After development, the resulting TP/A ratio of the system was measured as 1.45 Kbps/ALM.

Parallel Hardware Implementation. While developing the Nios II Custom Instruction, we simultaneously developed a memory-mapped, parallel hardware-accelerator, capable of performing encryption and decryption in accordance with the BIG specification, through the use of two memory-mapped interfaces: a control system and a datapath. Upon completion of the implementation, we achieved a resulting TP/A ratio of 13.16 Kbps/ALM, a 9× improvement over the Nios II Custom Instruction TP/A ratio; as such, the parallel hardware implementation was selected for final optimization.

Optimization Methodology. After choosing to optimize the parallel hardware implementation further, we concluded the most effective optimization techniques involved not only an appropriately modeled HDL design, but also the appropriate utilization of advanced synthesis options. Specifically, we optimized the synthesis options for minimally synthesized area, by: directly optimizing for area, eliminating register duplication (allowing shared registers), allowing resource sharing, and allowing RAM replacement. Upon optimization, we achieved an optimized TP/A ratio of 13.36 Kbps/ALM.

Computational Analysis. It is prudent to observe the limitations of a design, in order to identify and mitigate system bottlenecks. As such, we examine the efficiency of our hardware-accelerator, relative to the hardware-software interface. First, we identify the parameter of comparison as bits/cycle, where a higher ratio represents a more streamlined design. If the hardware-software interface has a higher ratio than the hardware-accelerator, the system is defined as computationally constrained, and suggests further optimization is needed for the hardware-accelerator, otherwise, the system is communication constrained, and the hardware-software interface to the hardware-accelerator becomes the bottleneck.

Using the methods and notations in [20], we identify the parameters of the hardware-software interface as v (bits per transfer) and B (cycles per transfer), where, based on the BIG specification,

$$v = 832\ bits/transfer, \tag{17}$$

$$B = 643\ cycles/transfer, \tag{18}$$

and

$$v/B \;=\; 1.311\; bits/cycle \tag{19}$$

Similarly, we identify the parameters of the hardware-accelerator as w (bits per execution) and H (cycles per execution), where, based on the BIG specification,

$$w \;=\; 256\; bits/execution, \tag{20}$$

$$H \;=\; 92\; cycles/execution, \tag{21}$$

$$w/H \;=\; 2.783\; bits/cycle, \tag{22}$$

and

$$v/B \;<\; w/H, \tag{23}$$

thus, the *hardware-software interface* is the bottleneck of the system, and the implementation is *communication constrained*, which suggests maximum throughput has been achieved for a memory-mapped hardware-accelerator.

Conclusively, we present the results of the optimized hardware-software codesign implementation in Table 9, as well as the interface of the implementation in Figure 10.

Table 9. Hardware-Software Codesign Implementation Results

	Encrypt Cycles	Decrypt Cycles	Total Cycles	Throughput (Mbps)	.text (bytes)	ALMs	Mem. Bits	TP/A Ratio (Kbps/ALM)
Optimized Design	402	395	797	16.0602	1,848	1,220	76,800	13.1641

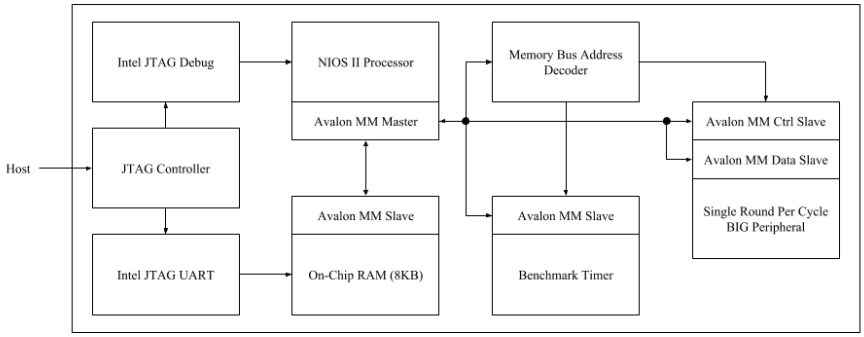

Figure 10. The HW/SW codesign Qsys (Platform Designer) interface for the BIG Cipher implementation, which bridges the Nios II Processor (HW) to the HW/SW Implementation (Round Slicing BIG Peripheral)

5 Conclusion

In this work, we presented a specification of a block cipher, the BIG (Block) (Instructional, Generic) cipher, intended to synergize the cryptographic educational competencies of mathematics, engineering, and computer science. This work demonstrates the ability of senior university undergraduate computer engineering students to analyze a cipher specification, conduct basic security analysis using linear cryptanalysis, and produce several implementations, optimized for distinct targets. All implementations leveraged an identical baseline of the Intel Nios II soft core microprocessor, 8 kilobytes of on-chip RAM, JTAG and interconnect peripherals, and a custom-designed benchmark timer.

Security analysis leveraged the linear cryptanalytic methods of [17] to conduct a reduced-round attack on the BIG cipher using 6 rounds. A recovery of a relationship of 12 key bits is described using as few as 16,000 plaintext/ciphertext pairs; however, a similar attack on a 12-round version would require more than 67 million plaintext/ciphertext pairs. Additionally, analysis of the 12-round version implemented in this research revealed a suboptimal diffusion rate of 43.3% bit-flips after 12 rounds, which could be exploited using bias vulnerabilities.

Implementation results are summarized in Table 10. The purely software implementation has the highest total latency (53,401 clock cycles), and the lowest throughput and throughput-to-area (TP/A) ratio, at 0.24 Mbps and 0.355 Kbps/ALM, respectively. The hardware implementation has improved throughput 3.138 Mbps and 2.68 Kbps/ALM, which represent 13× and 7.5× improvements, respectively, over the software implementation. This requires 1,170 ALMs, which is 494 additional Cyclone V FPGA ALMs above the baseline Nios II and peripherals. Finally, the hardware-software codesigned implementation achieves a 67× speed-up and 37× increase in TP/A ratio over the software implementation, and 5× speed-up and 5× increase in TP/A ratio compared to the hardware implementation, while requiring a total of 1,220 ALMs, which is 544 additional ALMs over the baseline Nios II and peripherals. Our analysis shows that the hardware-software designed is communication constrained, which illustrates upper-bounds on performance improvements realized by memory-mapped hardware accelerators.

Table 10. Summarized Implementation Results

	Encrypt Cycles	Decrypt Cycles	Total Cycles	Throughput (Mbps)	.text (bytes)	ALMs	Mem. Bits	TP/A Ratio (Kbps/ALM)
Software Design	26,667	26,734	53,401	0.24	6,084	676	76,800	0.355
Hardware Design	2,040	2,040	4,080	3.138	2,588	1,170	76,800	2.68
HW/SW Codesign	402	395	797	16.0602	1,848	1,220	76,800	13.1641

References

1. IBM Industry Insights, https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/, last accessed 2018/12/20.
2. CIA World Factbook, https://www.cia.gov/library/publications/the-world-factbook/rankorder/2153rank.html, last accessed 2018/12/20.
3. Handbook of Applied Cryptography, Menezes, A., van Oorschot, P. Vanstone, A., CRC Press, Florida (2001).
4. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M. The LED Block Cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011, LNCS, vol. 6917, pp. 326-341. Springer-Verlag Berlin, Heidelberg (2011).
5. Homeland Security Digital Library. FIPS 46-3 Data Encryption Standard. ANSI X3.92. National Institute of Standards and Technology (1995).
6. Cryptographic Competitions, AES: The Advanced Encryption Standard, https://competitions.cr.yp.to/aes.html, last accessed 2018/12/28.
7. NIST Hash Functions SHA-3 Project, https://csrc.nist.gov/projects/hash-functions/sha-3-project, last accessed 2018/12/28.
8. NIST Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process, https://csrc.nist.gov/projects/lightweight-cryptographyhttps://www.nist.gov/, last accessed 2018/10/14.
9. Cryptographic Competitions, CAESAR, https://competitions.cr.yp.to/caesar.html, last accessed 2018/12/28.
10. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L. The Simon and Speck Families of Lightweight Block Ciphers. DAC 2015. ACM. NSA Research Directorate (2013).

11. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E. TWINE: A Lightweight, Versatile Block Cipher. Proc of ECRYPT Workshop on Lightweight Cryptography, pp. 146-169. NEC Corporation, Kawasaki, Japan.
12. FIPS Publication 197," Advanced Encryption Standard (AES)", 2001.
13. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C. PRESENT: An Ultra-Lightweight Block Cipher, France, Germany, and Denmark (2007).
14. Nikova, S., Rijmen, V., Schlaffer, M. Secure hardware implementations of nonlinear functions in the presence of glitches. Journal of Cryptology, pp. 292-321. (2010).
15. Bilgin, B., Nikova, S., Nikov, V., Rijmen, V. and Stütz, G., "Threshold Implementations of all 3 x 3 and 4 x 4 S-boxes," Chapter *Cryptographic Hardware and Embedded Systems* – CHES 2012, Volume 7428 of the series Lecture Notes in Computer Science pp 76-91, 2012.
16. Diehl, W., Farahmand, F., Yalla, P., Kaps, J-P., Gaj, Kris. Comparison of hardware and software implementations of selected lightweight block ciphers. In: 27th International Conference on Field Programmable Logic and Applications (FPL) 2017. Ghent, Belgium (2017).
17. Matsui, M. Linear Cryptanalysis Method for DES Cipher. Workshop on the Theory and Application of Cryptographic Techniques, pp. 386-397. Springer-Verlag Berlin, Heidelberg (1998).
18. Shannon Communication Theory, math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf, last accessed 2018/12/29.
19. Howard M. Heys (2002) A Tutorial on Linear and Differential Cryptanalysis, Cryptologia, 26:3, pp. 189-221, https://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf, last accessed 2019/01/05.
20. Schaumont, P. A Practical Introduction to Hardware/Software Codesign, 2nd edn. Springer, New York City (2012).

6 Appendices

6.1 Appendix A

In order to facilitate replication of the BIG cipher, we provide three sample vectors to be used for testing (Table 11, Table 12, and Table 13).

Table 11. 12 Rounds Test Set 1 (PT= Plaintext, K = Key, CT= Ciphertext)

PT	= {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}
K	= {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}
CT	= {0xA8, 0x5E, 0x68, 0x2C, 0x0E, 0x14, 0x0E, 0x79, 0x67, 0x9E, 0xC7, 0x22, 0x13, 0x5B, 0x6C, 0x64}

Table 12. 12 Round Test Set 2 (PT= Plaintext, K = Key, CT= Ciphertext)

PT	= {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xFE, 0xBA, 0xBE, 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0}
K	= {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0xFF, 0xEE, 0xDD, 0xCC, 0xAA, 0x99, 0x88, 0x77}

CT	= {0xDA, 0xB1, 0xC4, 0xC0, 0xCA, 0x4D, 0xCF, 0x5B, 0x50, 0xEA, 0xF6, 0x17, 0xDB, 0x92, 0x55, 0x13}

Table 13. 18 Round Test Set 3 (PT= Plaintext, K = Key, CT= Ciphertext)

PT	= {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xFE, 0xBA, 0xBE, 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0}
K	= {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0xFF, 0xEE, 0xDD, 0xCC, 0xAA, 0x99, 0x88, 0x77}
CT	= {0x13, 0x4B, 0x75, 0x0C, 0xD4, 0xB8, 0x00, 0x65, 0x93, 0x50, 0x17, 0x7F, 0x76, 0x3B, 0x4A, 0x3D}

6.2 Appendix B

Algorithm 5. Generic C BIG Encryption and Decryption Implementation

```
#include <stdint.h>
#include <string.h>

// number of rounds for encryption/decryption (a multiple of 6)
#define NUM_ROUNDS 12

// array where SBox(Y) = sbox[Y]
const uint8_t sbox[16] = { 0xC, 0x9, 0xD,0x2, 0x5, 0xF, 0x3,
0x6, 0x7, 0xE, 0x0,
  0x1, 0xA, 0x4, 0xB, 0x8 };

// array where InvSBox(v) = inv_sbox[v]
const uint8_t inv_sbox[16] = { 0xA, 0xB, 0x3, 0x6, 0xD, 0x4,
0x7, 0x8, 0xF, 0x1,
  0xC, 0xE, 0x0, 0x2, 0x9, 0x5 };

// round constants through 18 rounds
const uint8_t round_constants[18] = { 0x5A, 0x34, 0x73, 0x66,
0x57, 0x35, 0x71,
  0x62, 0x5F, 0x25, 0x51, 0x22, 0x5F, 0x3E, 0x67, 0x4E, 0x07,
0x15 };

// Performs XOR on two arrays of 8 bytes producing X = X ^ Y
void xor_8_bytes(uint8_t * X, const uint8_t * Y)
{
  int i;
  for (i = 0; i < 8; i++)
    X[i] = X[i] ^ Y[i];
}

// Performs SubBytes on an array of 8 bytes
```

```
void SubBytes(uint8_t * Y)
{
  int i;
  uint8_t temp0, temp1;
  for (i = 0; i < 8; i++)
  {
    // lower nibble
    temp1 = Y[i] & 0xF;
    temp1 = sbox[temp1];

    // upper nibble
    temp0 = Y[i] >> 4;
    temp0 = sbox[temp0];
    Y[i] = (temp0 << 4) | temp1;
  }
}

// Performs InvSubBytes on an array of 8 bytes
void InvSubBytes(uint8_t * Y)
{
  int i;
  uint8_t temp0, temp1;
  for (i = 0; i < 8; i++)
  {
    // lower nibble
    temp1 = Y[i] & 0xF;
    temp1 = inv_sbox[temp1];

    // upper nibble
    temp0 = Y[i] >> 4;
    temp0 = inv_sbox[temp0];
    Y[i] = (temp0 << 4) | temp1;
  }
}

// Performs AddRoundConstant on an array of 8 bytes
// X is the array of 8 bytes
// i is the round
void AddRoundConstant(uint8_t * X, int i)
{
  X[6] ^= (round_constants[i] & 0x3) << 6;
  X[5] ^= round_constants[i] >> 2;
}

// Performs Perm_1 on two arrays of 8 bytes
```

```
// Y is the output array
// X is the input array
void Perm_1(uint8_t * Y, const uint8_t * X)
{
  Y[0] = X[2];
  Y[1] = X[3];
  Y[2] = X[0];
  Y[3] = X[1];
  Y[4] = X[6];
  Y[5] = X[7];
  Y[6] = X[4];
  Y[7] = X[5];
}

// Performs Perm_1 on an array of 8 bytes
void AssignPerm_1(uint8_t * X)
{
  uint8_t temp[8];
  memcpy(temp, X, 8);
  Perm_1(X, temp);
}

// Performs Perm_2 on an array of 8 bytes
void Perm_2(uint8_t * X)
{
  uint64_t op = 0, temp;

  // build a 64-bit unsigned integer out of the array
  int i;
  for (i = 0; i < 8; i++)
  {
    op <<= 8;
    op |= X[i];
  }
  temp = op;

  // perform circular shift
  temp <<= 21;
  op >>= 43;
  op |= temp;

  // deconstruct into bytes
  for (i = 7; i >= 0; i--)
  {
    X[i] = op & 0xFF;
```

```
    op >>= 8;
  }
}


// Performs (Perm_2)^-1 on an array of 8 bytes
void InvPerm_2(uint8_t * X)
{
  uint64_t op = 0, temp;

  // build a 64-bit unsigned integer out of the array
  int i;
  for (i = 0; i < 8; i++)
  {
    op <<= 8;
    op |= X[i];
  }
  temp = op;

  // perform the inverse circular shift
  temp >>= 21;
  op <<= 43;
  op |= temp;

  // deconstruct into bytes
  for (i = 7; i >= 0; i--)
  {
    X[i] = op & 0xFF;
    op >>= 8;
  }
}

// Calculates next round key (16-byte array)
void generateRoundKey_enc(uint8_t * round_key)
{
  uint8_t previous_round_key[16];
  memcpy(previous_round_key, round_key, 16);
  Perm_1(round_key, &previous_round_key[8]);
  int i;
  for (i = 0; i < 8; i++)
  {
    round_key[8 + i] = round_key[i] ^ previous_round_key[i];
  }
}
```

```
// Calculates previous round key (16-byte array)
void generateRoundKey_dec(uint8_t * round_key)
{
  uint8_t previous_round_key[16];
  memcpy(previous_round_key, round_key, 16);
  Perm_1(&round_key[8], previous_round_key);
  int i;
  for (i = 0; i < 8; i++)
  {
    round_key[i] = previous_round_key[i] ^ previous_round_key[8
+ i];
  }
}

// BIG Encryption Function
// plaintext, key, and ciphertext are arrays of 16 bytes
void encrypt(const uint8_t * plaintext, const uint8_t * key,
  uint8_t * ciphertext)
{
  uint8_t W[16], RK[16], temp[8];

  memcpy(W, plaintext, 16);
  memcpy(RK, key, 16);

  int i;
  for (i = 0; i < NUM_ROUNDS; i++)
  {
    generateRoundKey_enc(RK);

    xor_8_bytes(W, RK);
    SubBytes(W);

    AddRoundConstant(&W[8], i);
    xor_8_bytes(&W[8], &RK[8]);
    AssignPerm_1(&W[8]);
    Perm_2(&W[8]);

    // Feistel swap
    memcpy(temp, W, 8);
    xor_8_bytes(W, &W[8]);
    memcpy(&W[8], temp, 8);
  }
  memcpy(ciphertext, W, 16);
}
```

```
// BIG Decryption Function
// ciphertext, key, and plaintext are arrays of 16 bytes
void decrypt(const uint8_t * ciphertext, const uint8_t * key,
  uint8_t * plaintext)
{
  uint8_t W[16], RK[16], temp[8];

  memcpy(W, ciphertext, 16);
  memcpy(RK, key, 16);

  int i;
  for (i = NUM_ROUNDS - 1; i >= 0; i--)
  {
    memcpy(temp, &W[8], 8);
    xor_8_bytes(&W[8], W);

    InvPerm_2(&W[8]);
    AssignPerm_1(&W[8]);
    xor_8_bytes(&W[8], &RK[8]);
    AddRoundConstant(&W[8], i);

    InvSubBytes(temp);
    memcpy(W, temp, 8);
    xor_8_bytes(W, RK);

    generateRoundKey_dec(RK);
  }
  memcpy(plaintext, W, 16);

}
```