An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher*

AJ Elbirt[1], C Paar[2]
Electrical and Computer Engineering Department
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609, USA

[1] Email: aelbirt@ece.wpi.edu
[2] Email: christof@ece.wpi.edu

Preprint---To be presented at FPGA2000

Abstract

With the expiration of the Data Encryption Standard (DES) in 1998, the Advanced Encryption Standard (AES) development process is well underway. It is hoped that the result of the AES process will be the specification of a new non-classified encryption algorithm that will have the global acceptance achieved by DES as well as the capability of long-term protection of sensitive information. The technical analysis used in determining which of the potential AES candidates will be selected as the Advanced Encryption Algorithm includes efficiency testing of both hardware and software implementations of candidate algorithms. Reprogrammable devices such as Field Programmable Gate Arrays (FPGAs) are highly attractive options for hardware implementations of encryption algorithms as they provide cryptographic algorithm agility, physical security, and potentially much higher performance than software solutions. This contribution investigates the significance of an FPGA implementation of Serpent, one of the Advanced Encryption Standard candidate algorithms. Multiple architecture options of the Serpent algorithm will be explored with a strong focus being placed on a high speed implementation within an FPGA in order to support security for current and future high bandwidth applications. One of the main findings is that Serpent can be implemented with encryption rates beyond 4 Gbit/s on current FPGAs.

Keywords: cryptography, algorithm-agility, FPGA, block cipher, VHDL

*This research was supported in part through NSF CAREER award #CCR-9733246.

1 Introduction

The Data Encryption Standard has become the most widely used cryptosystem in the world. Developed by IBM, the DES algorithm was first published in the Federal Register of March 17, 1975 and was adopted as an interoperability standard for non-classified applications on January 15, 1977. DES has been reviewed every five years since its adoption and expired in 1998 [1]. The National Institute of Standards and Technology (NIST) has initiated a process to develop a Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard specifying an Advanced Encryption Algorithm. It is intended that the AES will specify a non-classified, publicly disclosed encryption algorithm that will be as widely accepted as DES in the private and public sectors [2].

NIST has solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms. This field was recently narrowed down to five. Unlike DES, which was designed specifically for hardware implementations, one of the design criteria for AES candidates is that they can be efficiently implemented in both hardware and software. Thus, NIST has announced that both hardware and software performance measurements will be included in their efficiency testing. So far, however, essentially all performance comparisons have been restricted to software implementations on various platforms [3].

The advantages of a software implementation include ease of use, ease of upgrade, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [4]. Conversely, cryptographic algorithms and their associated keys that are implemented in hardware are, by nature, more physically secure as they cannot easily be modified by an outside attacker [5]. However, algorithm agility is required to support algorithm independent protocols — that is, switching of encryption algorithms as frequent as on a per-session basis. The majority of modern security protocols, such as SSL or IPsec, allow multiple encryption algorithms. Reconfigurable devices such as FPGAs are a highly attractive option for a hardware implementation as they provide the flexibility of dynamic system evolution as well as the ability to easily implement a wide range of functions/algorithms. It appears to be especially relevant to focus on high-throughput implementations for FPGA-based encryption. We demonstrate that FPGA solutions can be at least one order of magnitude

faster than the fastest AES algorithm implementation on a high-end processor which encrypts at approximately 100 Mbit/s [6].

For this study, the AES candidate chosen was the Serpent encryption algorithm. As will be shown in Section 5, the Serpent algorithm was chosen due to its compact round structure. This structure readily lends itself to a pipelined implementation within an FPGA, leading to high-speed implementations. What follows is an investigation of the Serpent algorithm to determine the nature of its underlying components. The characterization of the algorithm's components will lead to the choice of a target FPGA so as to yield an implementation that is optimized for high-speed operation within a commercially available device. Finally, multiple architecture options of the Serpent algorithm within the targeted FPGA will be discussed and the overall performance of the implementations will be evaluated versus a typical software implementation.

2 Related Work

As opposed to custom hardware or software implementations, little work exists in the area of block cipher implementations within existing FPGAs. DES, the most common block cipher implementation targeted to FPGAs, has been shown to operate at speeds of up to 400 Mbit/s [7]. We believe that this performance can be greatly enhanced using today's technologies. As an example, the performance in [7] may be greatly improved by increasing the number of pipeline stages and retargetting the design to a more modern device. However, these speeds are still significantly faster than the best software implementations of DES [8] [9] [10], which typically have throughputs below 100 Mbit/s. This performance differential is an expected result of DES having been designed in the 1970s with hardware implementations in mind.

Other block ciphers have been implemented in FPGAs with varying degrees of success. A typical example is the IDEA block cipher which has been implemented at speeds ranging from 2.8 Mbit/s [11] to 528 Mbit/s [12]. Note that while the 528 Mbit/s throughput was achieved in a fully pipelined architecture, the implementation required four Xilinx XC4000 FPGAs. Additionally, IDEA is a 64-bit block cipher as opposed to the AES candidates which are 128-bit block ciphers. By employing bus widths and pipeline stages that are half the size of the AES candidates, FPGA implementations of IDEA are able to achieve higher clock frequencies and therefore higher throughputs as there is less routing congestion.

Some FPGA implementation throughputs for the AES candidates have been shown to be far slower than their software counterparts. Hardware throughputs of about 12 Mbit/s [14] [13] have been achieved for CAST-256. However, software implementations have resulted in throughputs of 37.8 Mbit/s for CAST-256 on a 200 MHz PentiumPro PC [6], a factor of three faster on average than FPGA implementations. These results were an important indication for us to investigate other AES candidate algorithms which are better suited for FPGA realizations.

When examining the AES candidate algorithms, it is important to note that they do not necessarily exhibit similar behavior to DES when comparing hardware and software implementations. While software implementations may be executed on processors operating at frequencies as high as 800 MHz, typical block cipher implementations that target FPGAs reach a maximum clock frequency of 50 MHz. Moreover, hardware pipelining becomes non-viable for some block ciphers. In particular, this is true when the pipeline's atomic operation is the algorithm's round function. This limitation is primarily caused by the requirement of some algorithms such as CAST-256 for large look-up-tables, typically on the order of 16–32K bits of memory. While many FPGAs are capable of providing up to 32K bits of memory, these look-up-tables would have to be replicated based on the number of rounds in the cipher in order to implement a fully pipelined design. This cannot be done given the resources available in current FPGAs. However, it will be shown that the Serpent algorithm is highly suitable for pipelined FPGA implementations, resulting in superior performance when compared to a software implementation.

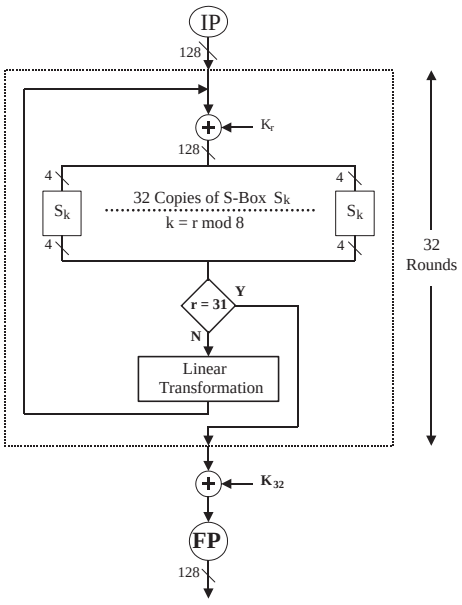

Figure 1: Serpent block diagram

3 Preliminaries: The Serpent Algorithm

The Serpent algorithm is a 32-round Substitution-Permutation (SP) network operating on four 32-bit words. The algorithm encrypts and decrypts 128-bit input data via a key of 128, 192, or 256 bits in length. The Serpent algorithm consists of three main components [15]:

- *Initial Permutation IP*
- *Thirty-two rounds consisting of a Round Function that performs Key Masking, S-Box Substitution, and (in all but the last round) data mixing via a Linear Transformation*
- *Final Permutation FP*

A block diagram for the Serpent algorithm is shown in Figure 1.

Most block ciphers require an inverted key schedule as the only modification needed to perform decryption. Typically, these ciphers are based on Feistel networks as opposed to

SP-networks [16] [17]. This results in the Serpent decryption process requiring inverse operations for the S-Boxes (implemented in reverse order) and the Linear Transformation in addition to a reverse ordering of the key schedule [15]. Note that the implemented version of the Serpent algorithm performs encryption but not decryption — this will be discussed in Section 6. Additionally, key scheduling was not considered as part of the design and only 256-bit keys were considered. The assumption was made that key scheduling is to be performed external to the implementation as it is only required before the encryption or decryption process begins. Note also that it is perceivable that the FPGA is initially configured to generate the subkeys which would be stored externally. The FPGA would then be reconfigured with the actual Serpent architectures described in this article.

The Serpent algorithm employs one round function comprised of three operations occurring in sequence:

- *Bit-wise XOR with the 128-bit Round Key* K_r
- *Substitution via thirty-two copies of one of eight S-Boxes*
- *Data mixing via a Linear Transformation*

These operations are performed in each of the thirty-two rounds with the exception of the last round. In the last round, the Linear Transformation is replaced with a bit-wise XOR with a final 128-bit key.

One of a total of eight different S-Boxes is used per round, where each S-Box performs a 4-bit to 4-bit substitution operation. The S-Box used is the round number modulo eight: round 9 uses S-Box 1, round 18 uses S-Box 2, etc. Each round requires thirty-two copies of the appropriate S-Box to operate on the 128-bit input data. The thirty-two 4-bit S-Box outputs form the 128-bit data that is input to the Linear Transformation.

The Linear Transformation mixes the four 32-bit blocks of data, denoted by X_0, X_1, X_2, and X_3, based on the equations below. Note that $<<<$ denotes a left rotation and $<<$ denotes a left shift [15].

$$
\begin{aligned}
Input &:= X_0, X_1, X_2, X_3 \\
X_0 &:= X_0 <<< 13 \\
X_2 &:= X_2 <<< 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &:= X_1 <<< 1 \\
X_3 &:= X_1 <<< 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &:= X_0 <<< 5 \\
X_2 &:= X_2 <<< 22 \\
Output &:= X_0, X_1, X_2, X_3
\end{aligned}
$$

4 FPGA Architectural Requirements

When examining the Serpent algorithm for hardware implementation within an FPGA, a number of key aspects emerge. First, it is obvious that the implementation will require a large amount of I/O pins to fully support the 128-bit data stream and the thirty-three 128-bit round keys. It is important to decouple the 128-bit input and output data streams to allow for a fully pipelined architecture. Since the round keys cannot change during the encryption or decryption process, they may be loaded via a separate key input bus prior to the start of encryption or decryption, requiring thirty-three clock cycles to load they entire key schedule.

The round function of the Serpent algorithm is comprised of three basic operations — bit-wise XOR, substitution via S-Boxes, and the linear transformation. Each output bit of the linear transformation is computed as the parity of a subset of the input bits [15]. The parity function and the bit-wise XOR function are of little hardware complexity to implement and impose no special architectural requirements. However, note that the S-Boxes required for the Serpent algorithm are extremely small. This allows for an implementation via asynchronous logic as opposed to using an internal clocked memory array containing 4-bit to 4-bit S-Box RAM tables. Asynchronous logic is typically implemented within FPGAs in the form of look-up-tables. Therefore, it is desireable that the targeted FPGA have 4-bit look-up-tables to easily configure the S-Boxes as asynchronous logic.

Additionally, to implement a fully pipelined architecture requires 128-bit wide pipeline stages, resulting in the need for a register-rich architecture to achieve a fast, synchronous implementation. Moreover, it is desireable to have as many register bits as possible per each of the FPGA's configurable units to allow for a regular layout of design elements as well as to minimize the routing required between configurable units. Finally, note that the Serpent algorithm employs no arithmetic operations in any part of the cipher. This feature minimizes the need for fast carry-chaining that is needed to maximize the performance of other AES candidate algorithms [13] [14].

Based on the aforementioned considerations, the Xilinx Virtex XCV1000BG560-4 FPGA was chosen as the target device. The XCV1000 has 128K bits of embedded RAM divided among thirty-two RAM blocks that are separate from the main body of the FPGA. The 560-pin ball grid array package provides 512 usable I/O pins and over one million usable gates. The XCV1000 is comprised of a 64×96 array of look-up-table based Configurable Logic Blocks (CLBs), each of which acts as a 4-bit element comprised of two 2-bit slices for a total of 12288 CLB slices. This type of configuration results in a highly flexible architecture that will accommodate the round functions' use of wide operand functions [18].

5 Implementations of the Serpent Algorithm

When examining the Serpent algorithm, a number of architecture options were investigated and implemented:

- Iterative Looping
- Iterative Looping With Partial Loop Unrolling
- Full Loop Unrolling
- Full (32-Stage) Pipelining

All versions of the Serpent algorithm were implemented entirely in VHDL using a bottom-up design and test methodology. The S-Boxes, Linear Transformation, Initial Permutation, and Final Permutation were implemented and tested

as stand-alone units. These core functions were integrated to form the nine possible versions of the round function. Eight versions of the round function, one for each possible S-Box grouping, are required to implement the first thirty-one rounds which follow the equation:

$$B_{i+1} \quad := \quad LT(S_{(i \bmod 8)}(B_i \oplus K_i))$$

The final version of the round function is required to implement the final round which follows the equation:

$$B_{32} \quad := \quad S_7(B_{31} \oplus K_{31}) \oplus K_{32}$$

The same hardware interface was used for each of the implementations of the Serpent algorithm. Round keys are loaded into registers and a *go* signal is used to initiate the encryption process. Note that all thirty-three keys must be loaded for the *go* signal to be recognized. The *go* signal is asserted on the falling edge of the system clock. Simultaneously, the first 128-bit block of plaintext is placed on the input data bus, and future data transitions on the falling edge of the system clock. When the *go* signal is asserted, the system is enabled and the key loading process is disabled. The assertion of the *valid* signal indicates that a valid 128-bit block of ciphertext has been placed on the output data bus. If the *go* signal is de-asserted, the *valid* signal is de-asserted after the last valid 128-bit block of ciphertext has been placed on the output data bus and the key loading process is re-enabled. Additionally, in the case of the fully pipelined implementation, the pipeline is flushed.

The VHDL implementations of the Serpent algorithm were simulated for functional correctness using the test vectors provided in the AES submission package [15]. After verifying the functionality of the implementations, the VHDL was synthesized and then placed and routed. The functionality of the placed and routed implementations was then re-simulated with annotated timing using the same test vectors, verifying that the implementations of the Serpent algorithm were successful.

5.1 Architecture 1: Iterative Looping

By only implementing a single round of the Serpent algorithm, a looping architecture with thirty-two iterations would seem to provide the greatest area-optimized solution. However, a significant drawback to this architecture is that the Serpent algorithm requires a different set of S-Box groupings for each round. This results in the need for additional 8-to-1 multiplexing hardware to switch between S-Box groupings based on the current round. Additionally, 32-to-1 multiplexing hardware is also required to switch between keys based on the current round. These multiplexers result in an increase in both the hardware resources required and the delay for computing the result of the current round. Embedded RAM may be used to replace the multiplexing hardware. By storing the keys within the RAM blocks, the appropriate key may be addressed based on the current round. However, due to the limited number of RAM blocks, as well as their bit width, this methodology is not feasible for a fully pipelined implementation. A pipelined approach requires more RAM blocks than are typically available. Therefore, the use of embedded RAM is not considered for this study to maintain consistency between architectural implementations.

In the iterative looping architecture, a single round is implemented via multiplexing of unique components. A single copy of each S-Box grouping as well as the round keys are multiplexed based on the round in progress and are followed by a single instantiation of the Linear Transformation. The output of each round is stored in a shared register that is used also used as the input to the following round, requiring a total of thirty-two loop iterations. The system is controlled via a state machine that allows for feedback of the ciphertext as required for Cipertext Feedback (CFB) mode at no throughput cost and follows the format described in Section 5. A block diagram for the Iterative Looping Architecture is shown in Figure 2. Note that PT represents the plain-text and CT represents the cipher-text.

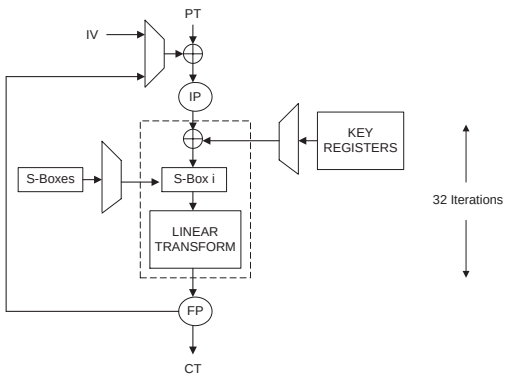

Figure 2: Iterative looping architecture block diagram

5.2 Architecture 2: Iterative Looping With Partial Loop Unrolling

Partial loop unrolling is an effective method for minimizing the multiplexing hardware required when implementing an iterative architecture. In the iterative looping with partial loop unrolling architecture, eight individual rounds are implemented and chained together with no registers between the rounds. This unrolling removes the need for the multiplexing of the S-Box groupings as each round contains one of the eight possible groupings. The data is passed through this structure in four iterations. This results in a decrease in both hardware resources and a the overall delay in computing the eight round grouping as compared to the iterative looping architecture. While the round keys must still be multiplexed, the multiplexing is done in groups of four to form the thirty-two round structure and resulting in a total of four loop iterations. The output of each iteration is stored in a shared register that is used also used as the input to the following iteration. The system is controlled via a state machine that allows for feedback of the ciphertext as required for CFB mode at no throughput cost and follows the format described in Section 5. A block diagram for the Iterative Looping with Partial Loop Unrolling Architecture is shown in Figure 3.

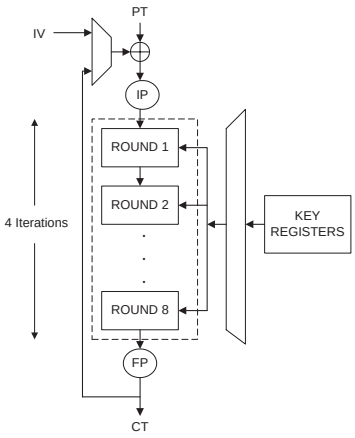

Figure 3: Iterative looping with partial loop unrolling architecture block diagram

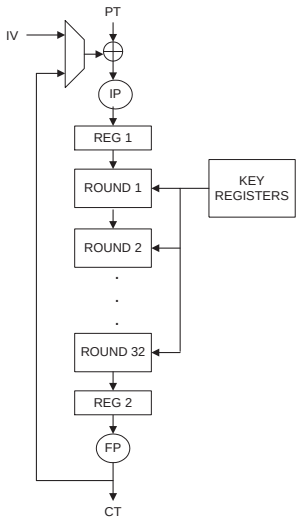

Figure 4: Full loop unrolling architecture block diagram

5.3 Architecture 3: Full Loop Unrolling

A fully unrolled architecture presents a more area-optimized solution for the implementation of the Serpent algorithm as compared to a fully pipelined architecture (to be discussed in Section 5.4). Surrounding the thirty-two rounds of the algorithm with two 128-bit registers, a single-stage pipeline is formed. The advantage of this architecture is the removal of thirty-one 128-bit registers that would be required for a fully pipelined architecture, resulting in a reduction of the area required for the implementation. However, implementing the thirty-two rounds of the algorithm in asynchronous logic severly hampers the overall clock frequency of the system.

In the full loop unrolling architecture, all of the elements of all of the round functions are implemented as asynchronous logic. The input plaintext and output ciphertext are registered separately, allowing for feedback of the ciphertext as required for CFB mode with no associated throughput pentalty. Each of the thirty-two rounds is instantiated and chained together to form the implementation with the appropriate key being assigned to a given round. The pipeline itself is controlled via a state machine which follows the format described in Section 5. A block diagram for the Full Loop Unrolling Architecture is shown in Figure 4.

5.4 Architecture 4: Full Pipelining

A fully pipelined architecture offers the advantage of extremely high throughput rates — once the latency of the pipeline has been met, the system will output a 128-bit block of ciphertext at each clock cycle. However, an architecture of this form requires significantly more hardware resources as compared to the other potential architectures. In a fully pipelined architecture, each round is implemented an individual element separated by 128-bit registers that form the actual pipeline.

In the fully pipelined architecture, all of the elements of a given round function are implemented as asynchronous logic. Some of the other AES candidates cannot be implemented using a pipelined architecture due to the large size of the S-Boxes. However, due to the small size of the Serpent S-Boxes (4-bit look-up-tables), the cost of S-Box replication is minimal in terms of the required hardware. It is important to note that due to the small amount of resources required to implement the round structure, the Serpent algorithm readily lends itself to a pipelined architecture within the targeted FPGA. However, the pipelined architecture can be fully exploited only in modes of operations which do not require feedback of the encrypted data, such as Electronic Code-Book (ECB) or counter mode. When operating in CFB mode, the resultant ciphertext must be XORed with the next block of incoming plaintext before encryption occurs, greatly reducing the speed-up gained via the use of a pipelined architecture (see Section 5 for further discussion of ECB vs. CFB mode).

The Serpent algorithm is implemented by instantiating components for each round based on the associated S-Box grouping. The output of each round is registered, becoming the input to the following round. The pipeline itself is controlled via a state machine which follows the hardware interface format described in Section 5. A block diagram for the Full Pipeline Architecture is shown in Figure 5.

6 Performance Evaluation

A number of parameters must be considered when evaluating the performance of the Serpent implementations. These parameters may be divided into two main classes — resource utilization characteristics and timing characteristics. Table 1 details the resource utilization characteristics for each of the Serpent implementations.

From Table 1 it is evident that the Serpent implementation is quite large, ranging from 45% to 73% utilization of the XCV1000 FPGA, allowing for a moderate amount of future expansion. Additionally, we see that the Serpent implementations may be targeted to a smaller FPGA such as the XCV800 [19]. However, a decrease in the size of the FPGA may result in increased routing congestion which in turn may lead to a decrease in performance via a decrease in the maximum operating frequency of the implementations and is therefore not recommended. Also note that it would

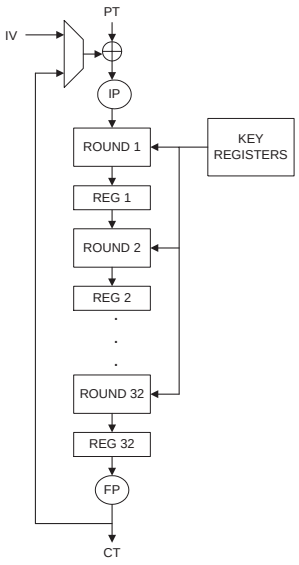

Figure 5: Full pipeline architecture block diagram

Architecture	CLB Slices	Utilization
1	5511	44.85 %
2	7964	64.81 %
3	8103	65.94 %
4	9004	73.27 %

Table 1: Resource utilization characteristics on the XCV1000

be impossible to implement a second copy of the Serpent algorithm to perform decryption within the same chip for Architectures 2, 3, and 4. Decryption requires inverse S-Boxes and an inverse Linear Transformation. Because the S-Boxes and the Linear Transformation form the bulk of the round function, it is fair to estimate the decryption implementation as requiring more space within the FPGA than the amount available for Architectures 2, 3, and 4. However, an interesting alternative which explores the specific capabilities of FPGAs is to reconfigure the target device with a decryption architecture on-the-fly.

Table 2 details the timing characteristics for the Serpent algorithm for both the hardware implementations and a typical software implementation. The software implementation of the Serpent algorithm was coded in C and executed on a 200 MHz PentiumPro PC [6].

Architecture	Cycles/ Enc Block	Frequency	Throughput
1	32	15.48 MHz	61.92 Mbit/s
2	4	13.88 MHz	444.16 Mbit/s
3	1	2.44 MHz	312.32 Mbit/s
4 - ECB Mode	1	37.97 MHz	**4.86 Gbit/s**
Software [6]	952	200.00 MHz	26.90 Mbit/s

Table 2: Timing characteristics on the XCV1000

From Table 2 it is evident that the hardware implementations outperform the software implementation in terms of efficiency by a factor ranging from 30 to 952 when comparing the number of clock cycles required to encrypt one 128-bit block of data. Additionally, the hardware implementations outperform the software implementation in terms of encryption throughput by a factor ranging from 2 to 180. Especially impressive is the throughput of Architecture 4. The throughput of 4.86 Gbit/s is within a factor 2 of the fastest reported DES implementation on an ASIC fabricated with static 0.6 micron CMOS technology [20]. The achieved throughputs for Architectures 2, 3, and 4 seem sufficient for current and next generation network applications.

Overall, the hardware implementations outperform the software implementation in every aspect of the timing characteristic measurements. This superior performance is mainly attributable to the parallelism exploited by the hardware implementations. Examples of exploited parallelism include the simultaneous computation of thirty-two copies of a given round function's S-Box as well as the ability to compute an entire round function in one clock cycle. While the software implementation operates at a significantly higher clock frequency, this gain is greatly outweighed by the increased number of clock cycles required to perform a round function. By minimizing the number of clock cycles per encryption, the hardware implementations achieve greater efficiency than their software counterpart. Moreover, additional parallelism may be exploited through the use of hardware pipelining, allowing for the encryption of one 128-bit block of data per clock cycle once the pipeline latency has been met. Through the use of these methods, the hardware implementations achieve far superior throughput results when compared to a software implementation.

It is of interest to note that when comparing Architectures 2 and 3, we see that partial loop unrolling outperforms full loop unrolling in both area and speed, leading to the conclusion that Architecture 3 is not worthwhile. Additionally, it is important to note that we see an increase in area by a factor of only 1.6 when comparing the iterative looping and fully pipelined architectures (1 and 4). While the round structure is decreased by a factor of 32, this decrease in size is proportional to the overhead incurred when storing the thirty-three 128-bit keys for each implementation. Because of this large amount of overhead, the decrease in the number of rounds has a much more limited affect on the overall decrease in area when comparing Architectures 1 and 4.

Although our fully pipelined design can support modes of operation such as the ECB or counter mode, it cannot be used for a straightforward CFB mode application. The cipher feedback (CFB) mode requires an XOR operation of the previous ciphertext with the current plaintext before the latter can be encrypted. One solution in this situation is to apply an interleaved CFB mode, with blocks 1, 33, 65, ..., and 2, 34, 66, ..., etc. forming chained blocks.

Another less efficient alternative is to encrypt one block and wait thirty-two clock cycles until the next block of plaintext is fed into the cipher and XOR-ed with the previous output. Although the encryption rate achieved by this implementation is a factor of 32 smaller than the 4.86 Gbit/s achieved by the pipelined implementation, it is still 151.88 Mbit/s. This throughput is still sufficient for ATM OC-3 payload encryption, which requires a data throughput of 140.85 Mbit/s, as only 48 of the 53 bytes in each cell require encryption.

Finally, when comparing the hardware implementations of the Serpent algorithm, it is of interest to examine the

area vs. throughput trade-off. In an effort to create a viable measurement metric for comparison purposes, we examine the throughput per CLB slices achieved for each implementation, the results of which are shown in Table 3. Note that this metric behaves inversely to the classical time-area (TA) product.

Architecture	**Throughput/ CLB Slice x 10^3**	**Relative Throughput/ CLB Slice**
1	11.24	1.00
2	55.77	4.96
3	38.54	3.43
4 - ECB Mode	539.76	48.02

Table 3: Area vs. throughput on the XCV1000

From Table 3 it is evident that Architecture 4 operating in ECB mode is by far the most efficient one. When operating in CFB mode, Architecture 2 exhibits the most favorable performance results. This is a result of the minimization of the hardware multiplexing and the low number of iterations. While Architecture 3 requires no iterations and has no pipeline latency, its low operating frequency results in a lesser performance as compared to Architecture 2. Architecture 1 displays even worse performance due to its multiple loop iterations and large amount of hardware multiplexing for both the keys and the S-Box groupings.

7 Conclusions

The importance of the Advanced Encryption Standard and the significance of a hardware implementation of the Serpent algorithm, an AES candidate, has been examined and the nature of the algorithm's underlying elements has been investigated. This investigation led to the architectural requirements for a target FPGA that optimized the performance of the Serpent implementation. Multiple architecture options of the Serpent algorithm were discussed, elaborating on key design choices that impacted the performance of the system. The performance of four Serpent hardware architectures were evaluated against the performance of a typical software implementation. From this evaluation the conclusion was reached that the nature of the Serpent algorithm coupled with the register-rich architecture of the chosen Xilinx XCV1000 FPGA result in a fast, synchronous, pipelined implementation which operates at encryption rates well beyond 4 Gbit/s. This data rate is sufficient for next-generation networks. When operating in CFB mode, it has been shown that an iterative looping implementation with partial loop unrolling results in the best performance when evaluating the area vs. speed trade-off. All of the hardware implementations of the Serpent algorithm resulted in a reduction in the number of clock cycles required to perform encryption of a 128-bit block of plaintext by a factor of at least 30 and an overall increase in throughput by a factor of over at least 2 and as much as 180 when compared to the software implementation. Should even greater performance be required, the hardware implementations may be further optimized through the use of hardware floorplanning.

References

[1] D. Stinson, *Cryptography, Theory and Practice*. Boca Raton, FL: CRC Press, 1995.

[2] National Institute of Standards and Technology (NIST), *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[3] National Institute of Standards and Technology (NIST), *Second Advanced Encryption Standard (AES) Conference*, (Rome, Italy), March 1999.

[4] B. Schneier, *Applied Cryptography*. John Wiley & Sons Inc., 2nd ed., 1995.

[5] R. Doud, "Hardware Crypto Solutions Boost VPN," *Electronic Engineering Times*, pp. 57–64, April 1999.

[6] B. Gladman, "Implementation Experience with AES Candidate Algorithms," in *Proceedings: Second AES Candidate Conference (AES2)*, (Rome, Italy), March 1999.

[7] J. Kaps and C. Paar, "Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine," in *5th Annual Workshop on Selected Areas in Cryptography (SAC '98)* (S. Tavares and H. Meijer, eds.), vol. LNCS 1556, (Queen's University, Kingston, Ontario, Canada), Springer-Verlag, August 1998.

[8] E. Biham, "A Fast New DES Implementation in Software," Technical Report, Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel, 1997.

[9] A. Pfitzmann and R. Assman, "More Efficient Software Implementations of (Generalized) DES," *Computers & Security*, vol. 12, no. 5, pp. 477–500, 1993.

[10] J. Hughes, "Implementation of NBS/DES Encryption Algorithm in Software," in *Colloquium on Techniques and Implications of Digital Privacy and Authentication Systems*, 1981.

[11] D. Runje and M. Kovac, "Universal Strong Encryption FPGA Core Implementation," in *Proceedings of Design, Automation, and Test in Europe*, (Paris, France), pp. 923–924, February 1998.

[12] O. Mencer, M. Morf, and M. Flynn, "Hardware Software Tri-Design of Encryption for Mobile Communication Units," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, (Seattle, WA), May 1998.

[13] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering CCECE'99*, (Edmonton, Alberta, Canada), May 1999.

[14] A. Elbirt, "An FPGA Implementation and Performance Evaluation of the CAST-256 Block Cipher," Technical Report, Cryptography and Information Security Group, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Worcester, MA, May 1999.

[15] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[16] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, no. 228, pp. 15–23, 1973.

[17] B. Schneier and J. Kelsey, "Unbalanced Feistel Networks and Block Cipher Design," in *International Workshop on Fast Software Encryption* (D. Gollmann, ed.), vol. LNCS 1039, (Cambridge, UK), Springer-Verlag, 1996.

[18] A. Elbirt and C. Paar, "Towards an FPGA Architecture Optimized for Public-Key Algorithms," in *The SPIE's Symposium on Voice, Video, and Data Communications*, (Boston, MA), September 19–22 1999.

[19] Xilinx Inc., *Virtex 2.5V Field Programmable Gate Arrays*, 1998.

[20] D. Wilcox, L. Pierson, P. Robertson, E. Witzke, and K. Gass, "A DES ASIC Suitable for Network Encryption at 10 Gbps and Beyond," in *Workshop on Cryptographic Hardware and Embedded Systems - CHES '99* (Ç. Koç and C. Paar, eds.), vol. LNCS 1717, (Worcester, MA), Springer-Verlag, 1999.