An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists *

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

[1] Email: {aelbirt, waihyip, christof}@ece.wpi.edu
[2] Email: spunge@alum.wpi.edu

The Third Advance Encryption Standard (AES3) Candidate Conference,
April 13-14, 2000, New York, USA.

Abstract

The technical analysis used in determining which of the Advanced Encryption Standard 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 FPGA implementations of four of the Advanced Encryption Standard candidate algorithm finalists. Multiple architectural implementation options are explored for each algorithm. A strong focus is placed on high throughput implementations, which are required to support security for current and future high bandwidth applications. The implementations of each algorithm will be compared in an effort to determine the most suitable candidate for hardware implementation within commercially available FPGAs.

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

1 Introduction

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 (AES), specifying an Advanced Encryption Algorithm to replace the Data Encryption Standard (DES) which expired in 1998 [1]. NIST has solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five have been selected as finalists. Unlike DES, which was designed specifically for hardware implementations, one of the design criteria for AES candidate algorithms 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, virtually all performance comparisons have been restricted to software implementations on various platforms [2].

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

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 [3] [4]. Conversely, cryptographic algorithms (and their associated keys) that are implemented in hardware are, by nature, more physically secure as they cannot easily be read or modified by an outside attacker [4]. The downside of traditional (ASIC) hardware implementation are the lack of flexibility with respect to algorithm and parameter switch. A promising alternative for implementation block cipher are reconfigurable hardware devices such as Field Programmable Gate Arrays (FPGAs). FPGAs are hardware devices whose function is not fixed and which can be programmed in-system. The potential advantages of encryption algorithms implemented in FPGAs include:

Algorithm Agility This term refers to the switching of cryptographic algorithms during operation. The majority of modern security protocols, such as SSL or IPsec, allow for multiple encryption algorithms. The encryption algorithm is negotiated on a per-session basis; e.g., IPsec allows among others DES, 3DES, Blowfish, CAST, IDEA, RC4 and RC6 as algorithms, and future extensions are possible. Whereas algorithm agility is costly with traditional hardware, FPGAs can be reprogrammed on-the-fly.

Algorithm Upload It is perceivable that fielded devices are upgraded with a new encryption algorithm which did not exist (or was not standardized!) at design time. In particular, it is very attractive for numerous security products to be upgraded for use of AES once the selection process is over. Assuming there is some kind of (temporary) connection to a network such as the Internet, FPGA-equipped encryption devices can upload the new configuration code.

Algorithm Modification There are applications which require modification of a standardized algorithm, e.g., by using proprietary S-boxes or permutations. Such modifications are easily made with reconfigurable hardware. Similarly, a standardized algorithm can be swapped with a proprietary one. Also, modes of operation can be easily changed.

Architecture Efficiency In certain cases, a hardware architecture can be much more more efficient if it is designed for a specific set of parameters; e.g., constant multiplication (of integers or in Galois fields) is far more efficient than general multiplication. With FPGAs it is possible to design and optimize an architecture for a specific parameter set.

Throughput Although typically slower than an ASIC implementations, FPGA implementations have the potential of running substantially faster then software implementations.

Cost Efficiency The time and costs for developing an FPGA implementation of a given algorithm are much lower than for an ASIC implementation. (However, for high-volume applications, ASIC solutions usually become the more cost-efficient choice.)

Note that algorithm agility remains an open research issue in regards to speed, physical security, and the cost associated with current high-end FPGA devices. However, we believe that cost is not a long-term limiting factor, as will be discussed in Section 3.3. For these reasons, this paper describes a thorough comparison the AES finalist algorithms RC6, Rijndael, Serpent, and Twofish with respect to implementation on state-of-the-art FPGAs. One aspect that seems to be especially relevant is the investigation of achievable encryption rates for FPGA-based implementations. We demonstrate that FPGA solutions encrypt at rates in the Gigabit range for all four algorithms investigated, which is at least one order of magnitude faster than most reported software implementations [5].

What follows is an investigation of the AES finalists to determine the nature of their underlying components. The characterization of the algorithms' components will lead to a discussion of the hardware architectures best suited for implementation of the AES finalists. A performance metric to measure the hardware cost for the throughput achieved by each algorithm's implementations will be developed and a

target FPGA will be chosen so as to yield implementations that are optimized for high-throughput operation within the commercially available device. Finally, multiple architecture options of the algorithms within the targeted FPGA will be discussed and the overall performance of the implementations will be evaluated versus typical software implementations.

2 Previous 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 [6]. We believe that this performance can be greatly enhanced using today's technology. These speeds are significantly faster than the best software implementations of DES [7] [8] [9], which typically have throughputs below 100 Mbit/s, although a 137 Mbit/s implementation has been reported as well [7]. 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 [10] to 528 Mbit/s [11]. Note that while the 528 Mbit/s throughput was achieved in a fully pipelined architecture, the implementation required four Xilinx XC4000 FPGAs.

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 [12] [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 [5], a factor of three faster than FPGA implementations. When scaled to a more current 600 MHz PentiumPro PC, it is expected that the same software implementation would outperform FPGA implementations by an even larger factor. While an FPGA implementation of RC6 achieved data rates of 37.8 Mbit/s [13], our findings indicate that considerably higher data rates are achievable.

When examining the AES finalists, it is important to note that they do not necessarily exhibit similar behavior to DES when comparing hardware and software implementations. One reason for this is that the AES finalists have been designed with efficient software implementations in mind. Additionally, software implementations may be executed on processors operating at frequencies as high as 800 MHz while typical implementations that target FPGAs reach a maximum clock frequency of 50 MHz.

3 Methodology

3.1 Design Methodology

There are two basic hardware design methodologies currently available: language based (high level) design and schematic based (low level) design. Language based design relies upon synthesis tools to implement the desired hardware. While synthesis tools continue to improve, they rarely achieve the most optimized implementation in terms of both area and speed when compared to a schematic implementation. As a result, synthesized designs tend to be (slightly) larger and slower than their schematic based counterparts. Additionally, implementation results can greatly vary depending on the synthesis tool as well as the design being synthesized, leading to potentially increased variances in the synthesized results when comparing synthesis tool outputs. This situation is not entirely different from a software implementation of an algorithm in a high-level language such as C, which is also dependent on coding style and compiler quality. As shown in [14], schematic based design methodologies are no longer feasible for supporting the increase in architectural complexity evidenced by modern FPGAs. As a result, a language based design methodology was chosen as the implementation form for the AES finalists with VHDL being the specific language chosen.

3.2 Implementations — General Considerations

Each AES finalist was implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. In an effort to achieve the maximum efficiency possible, note that key scheduling and decryption were not implemented for each of the AES finalists. Because FPGAs may be reconfigured in-system, the FPGA may be configured for key scheduling and then later reconfigured for either encryption or decryption. This option is a major advantage of FPGAs implementations over classical ASIC implementations. Round keys for encryption are loaded from the external key bus and are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. Each implementation was simulated for functional correctness using the test vectors provided in the AES submission package [15] [16] [17] [18]. After verifying the functionality of the implementations, the VHDL code was synthesized, placed and routed, and re-simulated with annotated timing using the same test vectors, verifying that the implementations were successful.

3.3 Selection of a Target FPGA

When examining the AES finalists 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 at high speeds where bus multiplexing is not an option. It is desirable 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 process, they may be loaded via a separate key input bus prior to the start of encryption. 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 desirable 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, it is critical that fast carry-chaining be provided between the FPGA's configurable units to maximize the performance of AES finalists that utilize arithmetic operations [13] [12].

In addition to architectural requirements, scalability and cost must be considered. We believe that the chosen FPGA should be the best chip available, capable of providing the largest amount of hardware resources as well as being highly flexible so as to yield optimal performance. Unfortunately, the cost associated with current high-end FPGAs is relatively high (several hundred US dollars per device). However, it is important to note that the FPGA market has historically evolved at an extremely rapid pace, with larger and faster devices being released to industry at a constant rate. This evolution has resulted in FPGA cost-curves that decrease sharply over relatively short periods of time. Hence, selecting a high-end device provides the closest model for the typical FPGA that will be available over the expected lifespan of AES.

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. 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 [19]. This type of configuration results in a highly flexible architecture that will accommodate the round functions' use of wide operand functions. Note that the XCV1000 also appears to be a good representative for a modern FPGA and that devices from other vendors are not fundamentally different. It is thus hoped that our results carry over, within limits, to other devices.

3.4 Design Tools

FPGA Express by Synopsys, Inc. and Synplify by Synplicity, Inc. were used to synthesize the VHDL implementations of the AES finalists. As this study places a strong focus on high throughput implementations,

the synthesis tools were set to optimize for speed. As will be discussed in Section 6, the resultant implementations exhibit the best possible throughputs with the associated cost being an increase in the area required in the FPGA for each of the implementations. Similarly, if the synthesis tools were set to optimize for area, the resultant implementations would exhibit reduced area requirements at the cost of decreased throughput.

XACTstep 2.1i by Xilinx, Inc. was used to place and route the synthesized implementations. For the sub-pipelined architectures, a 40 MHz timing constraint was used in both the synthesis and place-and-route processes as it resulted in significantly higher system clock frequencies. However, the 40 MHz timing constraint was found to have little affect on the other architecture types, resulting in nearly identical system clock frequencies to those achieved without the timing constraint.

Finally, Speedwave by Viewlogic Systems, Inc. and Active-HDL^{TM} by ALDEC, Inc. were used to perform behavioral and timing simulations for the implementations of the AES finalists. The simulations verified both the functionality and the ability to operate at the designated clock frequencies for the implementations.

4 Architecture Options and the AES Finalists

Before attempting to implement the AES finalists in hardware, it is important to understand the nature of each algorithm as well as the hardware architectures most suited for their implementation. What follows is an investigation into the key components of the AES finalists. Based on this breakdown, a discussion is presented on the hardware architectures most suited for implementation of the AES finalists.

4.1 Core Operations of the AES Finalist Algorithms

Algorithm	**XOR**	**Mod 2^{32} Add**	**Mod 2^{32} Subtract**	**Fixed Shift**	**Variable Rotate**	**Mod 2^{32} Multiply**	**GF(2^8) Multiply**	**LUT**
MARS	•	•	•	•	•	•		•
RC6	•	•		•	•	•		
Rijndael	•			•			•	•
Serpent	•			•				•
Twofish	•	•		•			•	•

Table 1: AES finalists core operations [20]

Modern FPGAs have a structure comprised of a two-dimensional array of configurable function units interconnected via horizontal and vertical routing channels. Configurable function units are typically comprised of look-up-tables and flip-flops. Look-up-tables may be configured as either combinational logic or memory elements. Additionally, many modern FPGAs provide variable-size SRAM blocks that may be used as either memory elements or look-up-tables [21].

In terms of complexity, the operations detailed in Table 1 that require the most hardware resources as well as computation time are the modulo 2^{32} multiplication and the variable rotation operations [20]. Implementing wide multipliers in hardware is an inherently difficult task that requires significant hardware resources. Additionally, algorithms that employ large variable rotations require a moderate amount of multiplexing hardware if carefully designed (see Section 5.1 for further discussion). S-Boxes may be implemented in either combinatorial logic or embedded RAM — the advantages of each of these options are discussed in Section 4.2. Fast operations such as bit-wise XOR, modulo 2^{32} addition and subtraction, and fixed value shifting are constructed from simple hardware elements. Additionally, the Galois field multiplications required in Rijndael and Twofish can also be implemented very efficiently in hardware as they are multiplications by a constant. Galois field constant multiplication requires far less resources than general multiplications [22].

Based on our evaluation of the AES finalists, the MARS algorithm appeared to be the most resource intensive based on its use of large S-Boxes, and modulo 2^{32} multiplication. As a result, it was conjectured

that the MARS algorithm would exhibit lesser performance when compared to the other AES finalists. Due to this evaluation and a lack of development resources, the MARS algorithm was omitted from this study.

4.2 Hardware Architectures

The AES finalists are all comprised of a basic looping structure (some form of either Feistel or substitution-permutation network) whereby data is iteratively passed through a round function. Based on this looping structure, the following architecture options were investigated so as to yield optimized implementations:

- Iterative Looping
- Loop Unrolling
- Partial Pipelining
- Partial Pipelining with Sub-Pipelining

Iterative looping over a cipher's round structure is an effective method for minimizing the hardware required when implementing an iterative architecture. When only one round is implemented, an n-round cipher must iterate n times to perform an encryption. This approach has a low register-to-register delay but a requires a large number of clock cycles to perform an encryption. This approach also minimizes in general the hardware required for round function implementation but can be costly with respect to the hardware required for round key and S-Box multiplexing. Iterative looping is a subset of loop unrolling in that only one round is unrolled whereas a loop unrolling architecture allows for the unrolling of multiple rounds, up to the total number of rounds required by the cipher. As opposed to an iterative looping architecture, a loop unrolling architecture where all n rounds are unrolled and implemented as a single combinatorial logic block maximizes the hardware required for round function implementation while the hardware required for round key and S-Box multiplexing is completely eliminated. However, while this approach minimizes the number of clock cycles required to perform an encryption, it maximizes the worst case register-to-register delay for the system, resulting in an extremely slow system clock.

A partially pipelined architecture offers the advantage of high throughput rates by increasing the number of blocks of data that are being simultaneously operated upon. This is achieved by replicating the round function hardware and registering the intermediate data between rounds. Moreover, in the case of a full-length pipeline (a specific form of a partial pipeline), the system will output a 128-bit block of ciphertext at each clock cycle once the latency of the pipeline has been met. However, an architecture of this form requires significantly more hardware resources as compared to a loop unrolling architecture. In a partially pipelined architecture, each round is implemented as the pipeline's atomic unit and are separated by the registers that form the actual pipeline. However, many of the AES finalists cannot be implemented using a full-length pipeline due to the large size of their associated round function and S-Boxes, both of which must be replicated n times for an n-round cipher. As such, these algorithms must be implemented as partial pipelines. Additionally, a 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 or Counter Mode [3, Section 9.9]. When operating in feedback modes such as Ciphertext Feedback Mode, the ciphertext of one block must be available before the next block can be encrypted. As a result, multiple blocks of plaintext cannot be encrypted in a pipelined fashion when operating in feedback modes. For the remainder of our discussion, feedback mode will be abbreviated as FB and non-feedback mode will be abbreviated as NFB.

Sub-pipelining a (partially) pipelined architecture is advantageous when the round function of the pipelined architecture is complex, resulting in a large delay between pipeline stages. By adding sub-pipeline stages, the atomic function of each pipeline stage is sub-divided into smaller functional blocks. This results in a decrease in the pipeline's delay between stages. However, each sub-division of the atomic function increases the number of clock cycles required to perform an encryption by a factor equal to the number of

sub-divisions. At the same time, the number of blocks of data that may be operated upon in NFB mode is increased by a factor equal to the number of sub-divisions. Therefore, for this technique to be effective, the worst case delay between stages will be decreased by a factor of m where m is the number of added sub-divisions. However, if the atomic function of the partially pipelined architecture has a small stage delay, sub-dividing the stage will achieve no significant decrease in the worst case stage delay. In this case, sub-pipelining would result in no significant increase in the system's clock frequency but would increase the logic resources and clock cycles required to perform an encryption, resulting in reduced throughput.

Many FPGAs provide embedded RAM which may be used to replace the round key and S-Box 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 restricted bit width, this methodology is not feasible for architectures with many pipeline stages or unrolled loops. Those architectures require more RAM blocks than are typically available. Additionally, the switching time for the RAM is more than a factor of three longer than that of a standard CLB slice element, resulting in the RAM element having a lesser speed-up effect on the overall implementation. Therefore, the use of embedded RAM is not considered for this study to maintain consistency between architectural implementations.

5 Architectural Implementation Analysis

For each of the AES finalists, the four architecture options described in Section 4.2 were implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. Round keys are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. These implementations yielded a great deal of knowledge in regards to the FPGA suitability of each AES finalist. What follows is a discussion of the knowledge gained regarding each algorithm when implemented using the four architecture types.

5.1 Architectural Implementation Analysis — RC6

When implementing the RC6 algorithm, it was first determined that the RC6 modulo 2^{32} multiplication was the dominant element of the round function in terms of required logic resources. Each RC6 round requires two copies of the modulo 2^{32} multiplier. However, it was found that the RC6 round function does not require a general modulo 2^{32} multiplier. The RC6 multipliers implement the function $A(2A + 1)$ which may be implemented as $2A^2 + A$. Therefore, the multiplication operation was replaced with an array squarer with summed partial products, requiring fewer hardware resources and resulting in a faster implementation. The remaining components of the RC6 round function — fixed and variable shifting, bit-wise XOR, and modulo 2^{32} addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. While variable shifting operations have the potential to require considerable hardware resources, the 5-bit variable shifting required by the RC6 round function required few hardware resources. Instead of implementing a 32-to-1 multiplexor for each of the thirty-two rotation output bits (controlled by the five shifting bits), a five-level multiplexing approach was used. The variable rotation is broken into five stages, each of which is controlled by one of the five shifting bits. For each rotation output bit of a given stage, a 2-to-1 multiplexor controlled by the stage's shifting bit is used. This implementation requires a total of 160 2-to-1 multiplexors as opposed to the thirty-two 32-to-1 multiplexors required for a one-stage implementation. However, using 2-to-1 multiplexors to form the five-stage barrel-shifter results in an overall implementation that is smaller and faster when compared to the one-stage barrel-shifter implementation as described in [18, Section 3.4]. Finally, it was found that the synthesis tools could not minimize the overall size of a RC6 round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire twenty rounds of the algorithm within the target FPGA.

As discussed in Section 4.2, implementing a single round of the RC6 algorithm provides the greatest area-optimized solution. Further loop unrolling provided only minor throughput increases as the decrease in

the number of cycles per encrypted block was offset by the rapidly decreasing system clock frequency. 2-stage partial pipelining was found to yield the highest throughput when operating in FB mode, outperforming the single round iterative looping implementation by achieving a significantly higher system clock frequency.

When operating in NFB mode, a partially pipelined architecture with two additional sub-pipeline stages was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, with the 10-stage partial pipeline implementation displaying the best throughput and results. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly two thirds of the round function's associated delay was attributed to the modulo 2^{32} multiplier. Therefore, two additional pipeline sub-stages were implemented so as to subdivide the multiplier into smaller blocks, resulting in a total of three pipeline stages per round function. As a result, an increase by a factor of more than 2.5 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the adders used to sum the partial products (a non-trivial task) to balance the delay between sub-pipeline stages.

5.2 Architectural Implementation Analysis — Rijndael

When implementing the Rijndael algorithm, it was first determined that the Rijndael S-Boxes were the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is an 8-bit to 8-bit look-up-table, requiring significant hardware resources. However, the remaining components of the Rijndael round function — byte swapping, constant Galois field multiplication, and key addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA.

Surprisingly, a one round partially pipelined implementation with one sub-pipeline stage provided the most area-optimized solution. As compared to a one-stage implementation with no sub-pipelining, the addition of a sub-pipeline stage afforded the synthesis tool greater flexibility in its optimizations, resulting in a more area efficient implementation. While 2-stage loop unrolling was found to yield the highest throughput when operating in FB mode, the measured throughput was within 10% of the single stage implementation. Due to the probabilistic nature of the place-and-route algorithms, one can expect a variance in performance based on differences in the starting point of the process. When performing this process multiple times, known as multi-pass place-and-route, it is likely that the single round implementation would achieve a throughput similar to that of the 2-stage loop unrolled implementation.

When operating in NFB mode, partial pipelining was found to offer the advantage of extremely high throughput rates once the pipeline latency was met, with the 5-stage partial pipeline implementation displaying the best throughput results. While Rijndael cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architecture.

Sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Rijndael round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

5.3 Architectural Implementation Analysis — Serpent

When implementing the Serpent algorithm, it was first determined that since the Serpent S-Boxes are relatively small (4-bit to 4-bit), it is possible to implement them using combinational logic as opposed to memory elements. Additionally, the S-Boxes map extremely well to the Xilinx CLB slice, which is comprised of 4-bit look-up-tables, allowing one S-Box to be implemented in a total of two CLB slices, yielding a compact implementation which minimizes routing between CLB slices. Finally, the components of the Serpent round function — key masking, S-Box substitution, and linear transformation — were found to be simple in structure, resulting in the round function requiring few hardware resources.

Implementing a single round of the Serpent algorithm provides the greatest area-optimized solution. However, a significant performance improvement was achieved by performing 8-round loop unrolling, removing the need for S-Box multiplexing hardware as one copy of each possible S-Box grouping is now included within one of the eight rounds. This amount of loop unrolling achieved a significant performance increase with little increase in hardware resources due to the compact nature of the Serpent round function. As expected, unrolling thirty-two rounds of the Serpent algorithm resulted in a lesser performance when compared to the eight round implementation. Implementing the thirty-two rounds of the algorithm in combinatorial logic severely hampered the overall clock frequency of the system, overriding the performance increase caused by the removal of the multiplexing hardware required to switch between keys.

When operating in NFB mode, a full-length pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, outperforming smaller partially pipelined implementations. In the fully pipelined architecture, all of the elements of a given round function are implemented as combinatorial logic. Other AES finalists cannot be implemented using a fully pipelined architecture due to the larger round functions. 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.

Finally, sub-pipelining of the partially pipelined architectures was determined to yield no throughput increase. Because the round function components are all simple in structure, there is little performance to be gained by subdividing them with registers in an attempt to reduce the delay between stages. As a result, the increase in the system's clock frequency would not outweigh the increase in the number of clock cycles required to perform an encryption, resulting in a performance degradation.

5.4 Architectural Implementation Analysis — Twofish

When implementing the Twofish algorithm, it was first determined that the synthesis tools were unable to minimize the Twofish S-Boxes to the extent of other AES finalist algorithms due to the S-Boxes being key-dependent. Therefore, the overall size of a Twofish round was too large to allow for a fully unrolled or fully pipelined implementation of the algorithm within the target FPGA. Moreover, the key-dependent S-Boxes were found to require nearly half of the delay associated with the Twofish round function.

As expected, implementing a single round of the Twofish algorithm provides the greatest area-optimized solution in terms of total CLB slices required for the implementation. Additional loop unrolling provided minor throughput increases as the decrease in the number of cycles per encrypted block was offset by the rapidly decreasing system clock frequency. However, single stage partial pipelining with one sub-pipeline stage was found to yield the best throughput and when operating in feedback mode. With a small increases in the required hardware resources, the sub-pipelined architecture was able to reach a significantly faster system clock frequency as compared to the loop unrolling and partial pipeline implementations.

When operating in NFB mode, a partially pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, with the 8-stage partial pipeline implementation displaying the best throughput results. While Twofish cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architecture.

Finally, sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Twofish round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

6 Performance Evaluation

Tables 2 and 3 detail the throughput measurements for the implementations of the three architecture types for each of the AES finalists for both NFB and FB mode. The architecture types — loop unrolling (LU), full or partial pipelining (PP), and partial pipelining with sub-pipelining (SP) — are listed along with the number of stages and (if necessary) sub-pipeline stages in the associated implementation; e.g., LU-4 implies a loop unrolling architecture with four rounds, while SP-2-1 implies a partially pipelined architecture with two stages and one sub-pipeline stage per pipeline stage. As a result, the SP-2-1 architecture implements two rounds of the given cipher with a total of two stages per round. Throughput is calculated as:

$$Throughput \;\; := \;\; (128 \;\; \text{Bits} \;\; * \;\; \text{Clock} \;\; \text{Frequency})/(\text{Cycles} \;\; \text{Per} \;\; \text{Encrypted} \;\; \text{Block})$$

Note that the implementation of a one stage partial pipeline architecture, an iterative looping architecture, and a one round loop unrolled architecture are all equivalent and are therefore not listed separately. Also note that the computed throughput for implementations that employ any form of hardware pipelining (as discussed in Section 4) are made assuming that the pipeline latency has been met.

The number of CLBs required as well as the maximum operating frequency for each implementation was obtained from the Xilinx report files. Note that the Xilinx tools assume the absolute worst possible operating conditions — highest possible operating temperature, lowest possible supply voltage, and worst-case fabrication tolerance for the speed grade of the FPGA [23]. As a result, it is common for actual implementations to achieve slightly better performance results than those specified in the Xilinx report files.

While this study focuses on high throughput implementations, the hardware resources required to achieve this throughput is also a critical parameter. No established metric exists to measure the hardware resource costs associated with the measured throughput of an FPGA implementation. Two area measurements of FPGA utilization are readily apparent — logic gates and CLB slices. It is important to note that the logic gate count does not yield a true measure of actual FPGA utilization. Hardware resources within CLB slices may not be fully utilized by the place-and-route software so as to relieve routing congestion. This results in an increase in the number of CLB slices without a corresponding increase in logic gates. To achieve a more accurate measure of chip utilization, CLB slice count was chosen as the most reliable area measurement. Therefore, to measure the hardware resource cost associated with an implementation's resultant throughput, the Throughput Per Slice (TPS) metric is used. We defined TPS as:

$$TPS \;\; := \;\; (\text{Encryption} \;\; \text{Rate})/(\#\;\; \text{CLB} \;\; \text{Slices} \;\; \text{Used})$$

Therefore, the optimal implementation will display the highest throughput and have the largest TPS. Note that the TPS metric behaves inversely to the classical time-area (TA) product.

When comparing implementations using the TPS and throughput metrics, it is required that the architectures are implemented on the same FPGA. Different FPGAs within the same family yield different timing results as a function of available logic and routing resources, both of which change based on the die size of the FPGA. Additionally, it is impossible to legitimately compare FPGAs from separate families as each family of FPGAs has a unique architecture which greatly affects the measured throughput and TPS. Finally, it is critical to note that throughput (and therefore TPS) may not scale linearly based on the number of rounds implemented for the three architecture types detailed in Section 4.1. As a result, it is imperative that multiple implementations be examined for each architecture type, varying the round count to determine the optimal number of rounds per implementation.

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)
RC6	LU-1	2638	13.8	20	88.5
RC6	LU-2	3069	7.3	10	94.0
RC6	LU-4	4070	3.7	5	94.8
RC6	LU-5	4476	2.9	4	92.2
RC6	LU-10	6406	1.5	2	97.4
RC6	PP-2	3189	19.8	10	253.0
RC6	PP-4	4411	12.3	5	315.5
RC6	PP-5	4848	12.1	4	386.7
RC6	PP-10	7412	13.3	2	848.1
RC6	SP-1-1	2967	26.2	20	167.6
RC6	SP-2-1	3709	26.4	10	337.8
RC6	SP-4-1	5229	24.6	5	629.8
RC6	SP-5-1	5842	25.8	4	825.2
RC6	SP-10-1	8999	26.6	2	1704.6
RC6	SP-1-2	3134	39.1	20	250.0
RC6	SP-2-2	4062	38.9	10	497.4
RC6	SP-4-2	5908	31.3	5	802.3
RC6	SP-5-2	6415	33.3	4	1067.0
RC6	**SP-10-2**	**10856**	**37.5**	**2**	**2397.9**
Rijndael	LU-1	3528	25.3	11	294.2
Rijndael	LU-2	5302	14.1	6	300.1
Rijndael	LU-5	10286	5.6	3	237.4
Rijndael	PP-2	5281	23.5	5.5	545.9
Rijndael	PP-5	10533	20.0	2.2	1165.8
Rijndael	SP-1-1	3061	40.4	10.5	491.9
Rijndael	SP-2-1	4871	38.9	5.25	949.1
Rijndael	**SP-5-1**	**10992**	**31.8**	**2.1**	**1937.9**
Serpent	LU-1	5511	15.5	32	61.9
Serpent	LU-8	7964	13.9	4	444.2
Serpent	LU-32	8103	2.4	1	312.3
Serpent	PP-8	6849	30.4	4	971.8
Serpent	**PP-32**	**9004**	**38.0**	**1**	**4860.2**
Twofish	LU-1	2666	13.0	16	104.2
Twofish	LU-2	3392	7.1	8	113.6
Twofish	LU-4	4665	3.3	4	106.8
Twofish	LU-8	6990	1.7	2	108.1
Twofish	PP-2	3519	11.9	8	190.4
Twofish	PP-4	5044	11.5	4	369.3
Twofish	PP-8	7817	10.8	2	689.5
Twofish	SP-1-1	3053	29.9	16	239.2
Twofish	SP-2-1	3869	28.6	8	457.1
Twofish	SP-4-1	5870	27.3	4	872.3
Twofish	**SP-8-1**	**9345**	**24.8**	**2**	**1585.3**

Table 2: AES finalist performance evaluation — non-feedback mode

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)
RC6	LU-1	2638	13.8	20	88.5
RC6	LU-2	3069	7.3	10	94.0
RC6	LU-4	4070	3.7	5	94.8
RC6	LU-5	4476	2.9	4	92.2
RC6	LU-10	6406	1.5	2	97.4
RC6	**PP-2**	**3189**	**19.8**	**20**	**126.5**
RC6	PP-4	4411	12.3	20	78.9
RC6	PP-5	4848	12.1	20	77.3
RC6	PP-10	7412	13.3	20	84.8
RC6	SP-1-1	2967	26.2	40	83.8
RC6	SP-2-1	3709	26.4	40	84.5
RC6	SP-4-1	5229	24.6	40	78.7
RC6	SP-5-1	5842	25.8	40	82.5
RC6	SP-10-1	8999	26.6	40	85.2
RC6	SP-1-2	3134	39.1	60	83.3
RC6	SP-2-2	4062	38.9	60	82.9
RC6	SP-4-2	5908	31.3	60	66.9
RC6	SP-5-2	6415	33.3	60	71.1
RC6	SP-10-2	10856	37.5	60	79.9
Rijndael	LU-1	3528	25.3	11	294.2
Rijndael	**LU-2**	**5302**	**14.1**	**6**	**300.1**
Rijndael	LU-5	10286	5.6	3	237.4
Rijndael	PP-2	5281	23.5	11	273.0
Rijndael	PP-5	10533	20.0	11	233.2
Rijndael	SP-1-1	3061	40.4	21	246.0
Rijndael	SP-2-1	4871	38.9	21	237.3
Rijndael	SP-5-1	10992	31.8	21	193.8
Serpent	LU-1	5511	15.5	32	61.9
Serpent	**LU-8**	**7964**	**13.9**	**4**	**444.2**
Serpent	LU-32	8103	2.4	1	312.3
Serpent	PP-8	6849	30.4	32	121.5
Serpent	PP-32	9004	38.0	32	151.9
Twofish	LU-1	2666	13.0	16	104.2
Twofish	LU-2	3392	7.1	8	113.6
Twofish	LU-4	4665	3.3	4	106.8
Twofish	LU-8	6990	1.7	2	108.1
Twofish	PP-2	3519	11.9	16	95.2
Twofish	PP-4	5044	11.5	16	92.3
Twofish	PP-8	7817	10.8	16	86.2
Twofish	**SP-1-1**	**3053**	**29.9**	**32**	**119.6**
Twofish	SP-2-1	3869	28.6	32	114.3
Twofish	SP-4-1	5870	27.3	32	109.0
Twofish	SP-8-1	9345	24.8	32	99.1

Table 3: AES finalist performance evaluation — feedback mode

Alg.	Arch.	Throughput (Gbit/s)		Slices	TPS
RC6	SP-10-2	**2.40**		10856	220881
Rijndael	SP-5-1	**1.94**		10992	176297
Serpent	PP-32	**4.86**		9004	539778
Twofish	SP-8-1	**1.59**		9345	169639

Table 4: AES finalist performance evaluation — non-feedback mode speed-optimized implementations

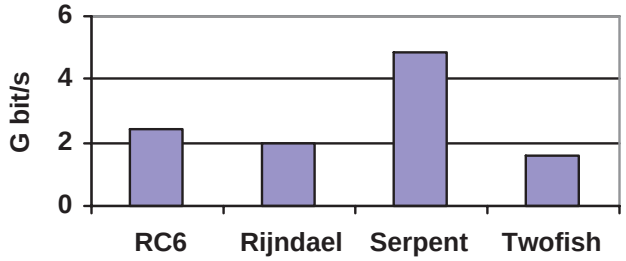

Figure 1: Best throughput — non-feedback mode

Alg.	Arch.	Throughput (Mbit/s)		Slices	TPS
RC6	PP-2	**126.5**		3189	39662
Rijndael	LU-2	**300.1**		5302	56605
Serpent	LU-8	**444.2**		7964	55771
Twofish	SP-1-1	**119.6**		3053	39169

Table 5: AES finalist performance evaluation — feedback mode speed-optimized implementations

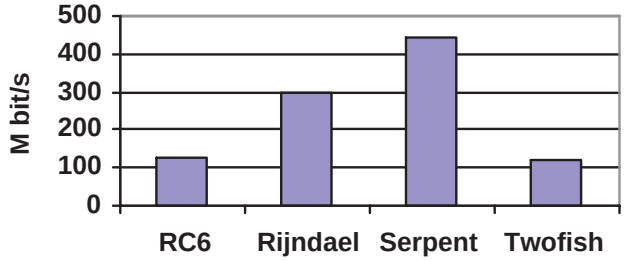

Figure 2: Best throughput — feedback mode

Tables 4 and 5 detail the optimal implementations of the AES finalists in both FB and NFB modes. Additionally, TPS is also shown for each of the implementations. It is critical to note that for the purposes of this study, the optimal implementation for an AES finalist is defined to yield the highest throughput. *As previously discussed, the synthesis tools were set to optimize for speed to guarantee that the highest throughputs would be achieved for each implementation. However, should an optimal implementation be defined based on either TPS or area, the implementation results shown in Tables 2 and 3 (and, as a result, those shown in tables 4 and 5 as well) are no longer representative of the best possible implementations for the architectures studied. To achieve a true representation that defines optimality based on either TPS or area, synthesis must be performed with the tools set to optimize for area.* While an area-efficiency analysis of the AES finalists warrants investigation, it is beyond the scope of this study.

Based on the data shown in Tables 4 and 5, the Serpent algorithm clearly outperforms the other AES finalists in both modes of operation. As compared to its nearest competitor, Serpent exhibits a throughput increase of a factor 2.2 in NFB mode and a factor 1.5 in FB mode. Interestingly, RC6, Rijndael, and Twofish

all exhibit similar performance results in NFB mode. However, Rijndael exhibits significantly improved performance in FB mode as compared to RC6 and Twofish, although it is still 50% slower than Serpent.

One of the main findings of our investigation, namely that Serpent appears to be especially well suited for an FPGA implementation from a performance perspective, seems especially interesting considering that Serpent is clearly not the fastest algorithm with respect to most software comparisons [5]. Another major result of our study is that all four algorithms considered easily achieve Gigabit encryption rates with standard commercially available FPGAs. The algorithms are at least one order of magnitude faster than the best reported software realizations. These speed-ups are essentially achieved by parallelization (pipelining and sub-pipelining) of the loop structure and by wide operand processing (e.g., processing of 128 bits in once clock cycle), both of which are not feasible on current processors. We would like to stress that the pipelined architectures cannot be used to their maximum ability for modes of operation which require feedback (CFB, OFB, etc.) However we believe that for many applications which require high encryption rates, non-feedback modes (or modified feedback modes such as interleaved CFB [3, Section 9.12]) will be the modes of choice. Note that the Counter Mode grew out of the need for high speed encryption of ATM networks which required parallelization of the encryption algorithm.

7 Conclusions

The importance of the Advanced Encryption Standard and the significance of high throughput implementations of the AES finalists has been examined. A design methodology was established which in turn led to the architectural requirements for a target FPGA. The core operations of the AES finalists were identified and multiple architecture options were discussed. The implementation of each architecture option for each of the AES finalists was analyzed to determine their suitability for hardware implementation. Based on the implementation results, the best speed-optimized implementations were identified for each AES finalist in both non-feedback and feedback modes. Upon comparison, it was determined that the Serpent algorithm yielded the best performance in both modes, where best performance was defined strictly as the highest throughput. The Serpent algorithm outperforms its nearest competitor by a factor of 2.2 in non-feedback mode and by a factor of 1.5 in feedback mode.

8 Acknowledgement

We would like to thank Pawel Chodowiec and Kris Gaj from George Mason University for their helpful discussion and the VHDL code modules that were provided to assist in the implementation of some of the AES finalists. We would also like to thank Alan Martello from the University of Pittsburgh for his public-domain VHDL code module that was used in implementation of the AES finalists.

References

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

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

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

[4] R. Doud, "Hardware Crypto Solutions Boost VPN," *EETimes*, pp. 57–64, April 1999.

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

[6] 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.

[7] E. Biham, "A Fast New DES Implementation in Software," in *Fast Software Encryption. 4th International Workshop, FSE'97 Proceedings*, (Berlin), pp. 260–272, Springer-Verlag, 1997. Lecture Notes in Computer Science Volume 1267.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," in *accepted for CCECE'99*, (Edmonton, Alberta, Canada), 1999.

[14] C. Phillips and K. Hodor, "Breaking the 10k FPGA Barrier Calls For an ASIC-Like Design Style," *Integrated System Design*, 1996.

[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] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall, "Twofish: A 128-Bit Block Cipher," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[17] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[18] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6$^{\mathrm{TM}}$ Block Cipher," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

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

[20] B. Chetwynd, "Universal Block Cipher Module: Towards a Generalized Architectures for Block Ciphers," Master's thesis, Worcester Polytechnic Institute, Worcester, MA, November 1999.

[21] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," in *IEEE Design & Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[22] C. Paar, "Optimized Arithmetic for Reed-Solomon Encoders," in *1997 IEEE International Symposium on Information Theory*, (Ulm, Germany), p. 250, June 29 – July 4 1997.

[23] P. Alfke, "Xilinx M1 Timing Parameters." electronic mail personal correspondance, December 1999.