# Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms

Bryan Weeks, Mark Bean, Tom Rozylowicz, Chris Ficke

National Security Agency

# 1 Abstract

The National Security Agency (NSA) is providing hardware simulation support and performance measurements to aid NIST in their selection of the AES algorithm. Although much of the Round 1 analysis focused on software, much more attention will be directed towards hardware implementation issues in the Round 2 analysis. As NIST has stated, a common set of assumptions will be essential in comparing the hardware efficiency of the finalists. This paper presents a technical overview of the methods and approaches used to analyze the Round 2 candidate algorithms (MARS, RC6, RIJNDAEL, SERPENT and TWOFISH) in CMOS-based hardware. Both design procedures and architectures will be presented to provide an overview of each of the algorithms and the methods used. To cover a wide range of potential hardware applications, two distinct architectures will be targeted for comparison, specifically a medium speed, small area iterated version and a high speed, large area pipelined version. The standard design approach will consist of creating hardware models using VHDL and an underlying library of cryptographic components to completely describe each algorithm. Once generated, the model can be verified for correctness through simulation and comparison to test vectors, and synthesized to a common CMOS hardware library for performance analysis. Hardware performance data will be collected for a variety of design constraints for each of the algorithms to ensure a wide range of measured data. A summary report of the findings will be presented to demonstrate algorithm performance across a wide range of metrics, such as speed, area, and throughput. This report will provide a common baseline of information, which will enable NIST and the community to compare the hardware performance of the algorithms relative to one another.

# 2 Introduction

The National Security Agency (NSA) agreed to provide technical support to the National Institute of Standards and Technology (NIST) in the form of an analysis of the hardware performance of the Round 2 Advanced Encryption Standard (AES) algorithm submissions. This analysis consisted of the design, coding, simulation and synthesis of the five algorithms using the procedure outlined below. Throughout this evaluation, NSA has taken care to assure that best design practices were used and that all algorithms received equal treatment. No attempt was made to optimize any particular design, but care was taken to find the best configuration for each algorithm. Cross-validation measures during design and simulation were used to overcome the subjective effects of the design process and to ensure that all designs receive the same amount of attention. The results of this analysis should provide an accurate measure of the hardware performance of each algorithm relative to the others. Undoubtedly more optimized (and hence better performing) implementations of these algorithms can be designed, so the individual score of any particular algorithm is not very valuable outside the context of this environment. The point of this analysis is to provide a controlled setting in which a meaningful comparison can be made.

Based on a mathematical description of the Round 2 algorithms, and C code reference models when necessary for clarification, NSA designers fully described each of the algorithm submissions in a hardware modeling language. A review by a team of design engineers followed the initial design stage to reduce the effects of coding style on performance. Using commercially available analysis, simulation and synthesis tools, NSA design engineers have performed simulations to produce performance estimates based on each of the hardware models. In order to provide a wider perspective on the performance of the algorithms, two different architectures or applications were simulated for each algorithm: an iterative version to provide a medium speed operation at minimal area/transistor count, and a pipelined version to provide optimum speed operation, but at the cost of a larger area. This report is a summary of the performance of the Round 2 AES candidate algorithms, and will compare and contrast the results of the analysis.

# 3 Hardware Design Background

## 3.1 Design Guidelines

For this analysis effort, one of the main goals was to provide an unbiased comparison of the algorithms in hardware, specifically in Application Specific Integrated Circuits (ASICs). To that end, the overhead found in typical hardware implementations, such as a robust user-interface, was minimized to reduce the impact on the overall performance of the algorithms. The user-interface is the Input/Output (I/O) connections and logic needed to take the plaintext and key and present them to the algorithm, and take the output ciphertext and present it off the chip. All inputs and control signals were registered in a common interface in order to provide uniformity across all of the algorithms, with fixed setup and hold times identical for all algorithms. A wide variety of architectures could be used to implement a given algorithm. In order to restrict all possible choices and yet capture valuable data points, two fundamental architectures were chosen: iterative and pipelined. All algorithms were designed in each architecture style. There are several variations on these approaches, including multiple copies of an iterative implementation for parallel processing, a partially pipelined implementation, or a combination of these hybrids (multiple copies of a partially pipelined implementation). The approach chosen will depend on the needs of the system, but these variations will likely result in performance within the ranges given by the iterative and fully pipelined implementations. However, these optimizations were beyond the scope of this study.

### 3.1.1 Target Applications

### 3.1.2 Iterative Architecture

The iterated approach to implementing the algorithm focuses on providing a medium to low speed version of the algorithm, with efforts placed on limiting the physical size of the hardware. In this instance of the algorithm, one step is performed per clock period, with the output of the previous step being used as the input to the next step. Data is only placed on the output after the required number of algorithm rounds has been completed.

### 3.1.3 Pipelined Architecture

The pipelined approach to implementing an algorithm centers on providing the highest throughput to the design, sacrificing area to obtain the level of performance needed. In the case of pipelining, all of the steps in computing the algorithm are cascaded into a single design, with each stage feeding the next stage. The latency remains the same as in the iterated case, but the throughput is increased significantly as new data is placed on the output on every clock cycle. Pipelining has been shown to be an effective method of dramatically increasing the throughput capabilities of a given algorithm. However, it comes at the expense of limiting the number of cryptographic modes that can be supported at the maximum throughput rate. For example, since the latency of an encryption cycle remains the same as an iterative case, there is no throughput advantage when using feedback modes such as Cipher Block Chaining (CBC). High performance applications, such as high speed network encryption, will require the increase in throughput, and as a result, often focus on a non-feedback mode of operation such as counter mode to obtain performance.[1]

## 3.2 Parameter Description

There are many design parameters that can be reported for each design implementation. Some parameters will have much more significance in a given application or environment than others. This evaluation reports on these parameters as a method of comparison among the five algorithms, and does not claim that any single parameter has been fully optimized. The following is a description of the parameters being reported. Some have a direct impact or relation to performance metrics (e.g. throughput) and some are simply a function of the algorithm itself (e.g., I/O requirements). Algorithm performance in each of the evaluation categories will be documented for each algorithm submission.

### 3.2.1 Area

As an estimate based on an available MOSIS library, the results of the synthesis area reporting will consist of pre-layout area estimates of the algorithm. Although potentially different from a post-layout estimate, the area reported

by Synopsys will provide a relative comparison of each of the algorithm submissions. Generally, the two varieties of architectures-– iterative and pipeline -– will be on the extremes of area with the iterative being the smallest, and the pipelined being the largest.

### 3.2.2 Throughput

In most cases, throughput is directly proportional to area; as area decreases, throughput decreases. As with area, the iterative and pipelined architectures will report the extremes of throughput. Iterative architectures will have much lower throughput rates since there is a minimum amount of hardware, and it is re-used on multiple clock cycles of execution. Thus, the throughput is limited by the amount of hardware reuse. More specifically, it is limited by the number of rounds in a codebook algorithm. On the other hand, a pipelined architecture dedicates hardware for performing all calculations in any given clock cycle. This maximizes throughput by allowing data to be written and read from the device on every clock. In this case, throughput is a function of the worst-case delay in any one given stage of the algorithm. Throughput will be reported for both iterative and pipelined architectures.

### 3.2.3 Transistor Count

Transistor count is a more specific measure than area and is often more useful. While transistor count is somewhat dependent on the design library being used, it is a useful method of comparing the algorithms since they were compiled using the same library. In addition, the transistor count will be a more useful figure than area when estimating programmable logic implementations since these devices typically report the number of useable gates (which is also directly related to transistor count). Based on the synthesized netlist (from Synopsys), an additional report describing the number of transistors required to implement the algorithm will be provided.

### 3.2.4 Input/Outputs (I/O) Required

With the goal of consistency among algorithms, the I/O was fixed identically for all algorithms. However, since this parameter is highly useful to hardware designs, it will still be reported.

### 3.2.5 Key Setup Time

The key setup time refers to the amount of time required before subkey expansion is ready to execute. Some algorithms use the user-supplied key directly in the subkey expansion thereby reducing the key setup time to zero. Others require some pre-calculation or translation of the key prior to subkey expansion steps. Key setup times will be examined to assess the overhead of each algorithm in establishing a usable key.

### 3.2.6 Algorithm Setup Time

Similar to key setup time, the algorithm setup time reports the minimum amount of time before an algorithm is ready to process data. Time to create look-up tables, etc. will fall in this category. None of the evaluated algorithms contained an algorithm setup time greater than zero.

### 3.2.7 Time to Encrypt One Block

This paramter will address minimum latency times for each of the algorithm submissions. The time to encrypt one block, measured in nanoseconds, is a function of two parameters: the worst-case path delay between any two registers, and the number of rounds in the algorithm.

### 3.2.8 Time to Decrypt One Block

As above, this parameter will address minimum latency times for each of the algorithm submissions. Decryption does not always require identical processing as encryption. Therefore, the time required to decrypt one block is reported.

### 3.2.9 Time to Switch Keys

Originally, this parameter was included as a measure to encompass both key setup time and algorithm setup time overhead. However, since none of the evaluated algorithms contained an algorithm setup time, this parameter is identical to key setup time. Therefore, it will not be reported further in this document.

# 4 Methodology

## 4.1 Standard Design Flow

The design process followed a common methodology used by ASIC designers. The process started with the documentation supplied by the algorithm authors and was completed with a gate-level schematic, which included the performance metrics data. A complete ASIC development would require physical layout and fabrication. These steps were beyond the scope of this effort. However, the performance metrics data obtained here closely matches that which would be found from actual fabrication and testing. Previous efforts using these tools have correlated estimated performance from the schematic to the actual testing. Figure 1 shows the steps in the design flow.

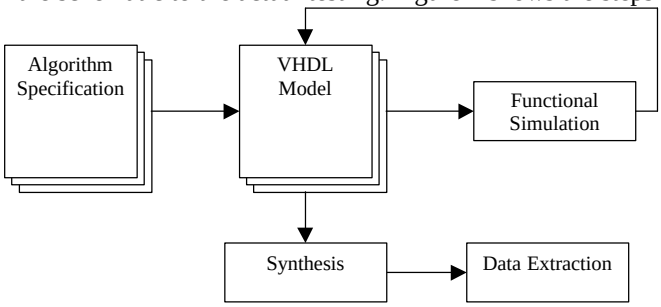


**Figure 1 Standard Design Flow**

### 4.1.1 VHDL code generation

*VHSIC Hardware Description Language (VHDL)*

VHDL modeling is analogous to programming simulations in C code and follows much of the same syntax. However, unlike a behavioral description of the algorithm, VHDL (IEEE 1076) specifies how the algorithm will be implemented in hardware. Using this hardware language, NSA designers fully described the hardware necessary to implement each of the algorithm submissions. Performance metrics, such as speed, area, etc. (see below) can be estimated from the hardware description using available analysis and computer aided design (CAD) tools.

There are different styles in which to code VHDL models, offering various levels of abstraction. For this evaluation, the designers used the register transfer logic (RTL) coding style. For this style, the placement of registers and corresponding logic between registers is chosen by the designer and is determined at the VHDL code level. There are many different methods for identifying an optimized placement of registers. Ideally, there would be an equal amount of logic delay between registers for all stages of the design. However, in order to simplify the design cycle and to be consistent among all algorithms, the designers chose a common placement of registers, even if this placement is not fully optimized. Specifically, the output of each “round” (as defined by the algorithm authors) is registered for both a key schedule round and an algorithm round.

### 4.1.2 Simulation and Verification

NSA followed the design phase with a functional VHDL simulation of the designs using the Synopsys VHDL System Simulator (VSS) to verify the correct operation of the algorithm. The test vectors submitted to NIST for each algorithm were applied to assure that the design was working as intended. Specifically, the Variable Key and Variable Text tests were performed for each algorithm implementation and mode (e.g., iterative encrypt, pipelined decrypt, etc.). The modeled algorithm output was also compared with the C code model supplied to provide an added assurance that the simulation was operating as expected.

### 4.1.3 Code review

NSA had one or more engineers design the VHDL for each algorithm submitted. Initial hardware designs were straightforward implementations of the core algorithm. Following completion of each initial design, an informal group of engineers met to review and provide feedback for the design. Improvements and alternatives to the initial design were examined to determine potential benefits from differing architecture approaches (area compression, pipelining, etc.). Variants of the design that improve the performance of the algorithm were then programmed for comparison.

### 4.1.4 Synthesis

Gate-level synthesis of the algorithm utilized the Synopsys Design Compiler to produce a functionally equivalent schematic in hardware. A MOSIS-specific technology library was used to generate a gate-level schematic of the design and provide more accurate area and timing estimates, as if the design were to be implemented in an integrated circuit (IC). The MOSIS library is based on a publicly available fabrication facility's model of a specific CMOS process, thus giving real performance metrics for an available ASIC line. The VHDL model can be re-targeted to any supported hardware or field programmable gate array (FPGA) design libraries.

The synthesis process can generate a wide range of implementations depending on the constraints provided to the synthesis tool. For example, one implementation may minimize area while another may minimize delay time. In hardware synthesis, the two fundamental parameters are time and area. These parameters are directly related. As delay time decreases, area increases. Timing and area curves that further illustrate this point are shown in subsequent sections. The constraints provided for each algorithm synthesis routine were maintained consistently. Therefore, differences among algorithm synthesis results will be a function of the logic required (algorithm specific) and the synthesis tool's ability to meet the given constraints.

### 4.1.5 Documentation

In addition to a summary report containing performance data, both design notebooks and VHDL documentation will be provided to NIST for evaluation. The design notebooks will contain reporting information for all of the hardware data that was collected, with all algorithms, designs, and architectures represented. The VHDL models and their testbenches (for simulation verification) will also be included.

## 4.2 Synthesis Analysis

### 4.2.1 Function Characterization

Although the hardware design of the algorithms followed a top-down approach, the synthesis portion of the analysis proceeded from the bottom of the design up through the hierarchy. In order to obtain an accurate picture of the performance of the sub-blocks and functions in this type of analysis, a sweep was performed on each of the functional blocks to graphically depict performance versus design constraints. Specifically, the timing constraints, such as output delay and clock frequency, of each of the blocks were varied to observe the performance output of the block. The results of the sweep make up the characterization for that particular block. All subsequent blocks of the hierarchy will be analyzed using these methods.

### 4.2.2 Cryptographic Library

With characterization curves for each of the sub-blocks complete, five speed grade implementations were selected to cover the performance range of the block. A variety of key performance points were selected to reflect requirements for both high speed and small area. Figure 2 shows typical timing and area curves following a sweep of maximum delay time constraints. These curves allow design engineers to select specific implementations of a given function. Specifically, five implementations, or speed grades, were chosen for each function. In this example, the five selected implementations are noted in the figure, and they represent one minimum time delay, one minimum area, and three other points that have desired characteristics such as large area savings for a small increase in delay time.

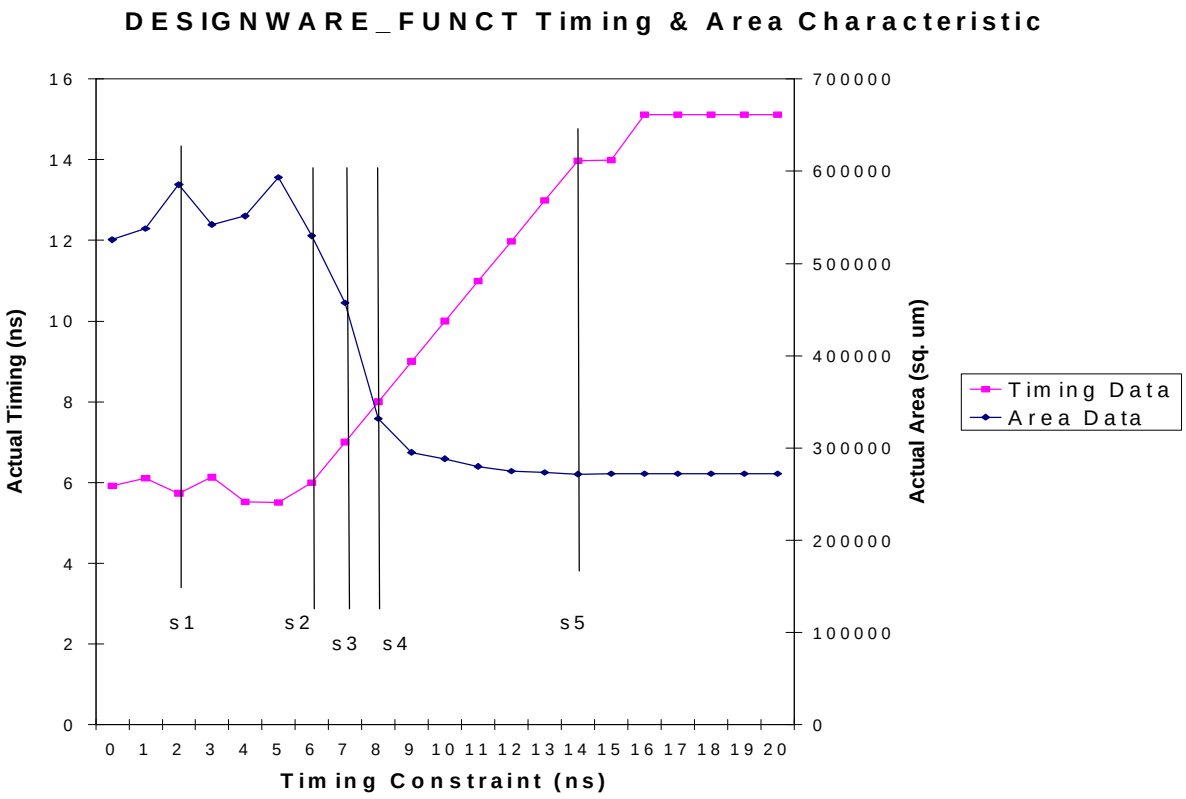


**Figure 2 Sample Function Sweep**

The five implementations were selected only for the functions of each of the algorithms, and then assembled into a cryptographic library. Each library contained implementations of all the functions required to build the given algorithms.

### 4.2.3 Block Level Characterization

Continuing with the bottom up approach, the higher level blocks underwent the same performance sweep as described for the function level. The design constraints were varied across the entire range of the block to fully describe the performance curve of the block. At each iteration of the synthesis process, components (e.g., functions) were selected from the cryptographic library based on the required speed and performance. Performance curves at the block level encompassed components of several different speed grades depending on design constraints. At the top level, the characterization curve reflected the performance of the entire design across a wide range of design constraints.

# 5 General Architecture Approach

## 5.1 Top Level Architecture

The design of each algorithm started with a common top level architecture that is well suited for virtually any codebook algorithm. The generalized top level architecture consists of an Interface, an Algorithm block, a Key Schedule block and a Controller. Figure 3 shows a block diagram of the architecture.

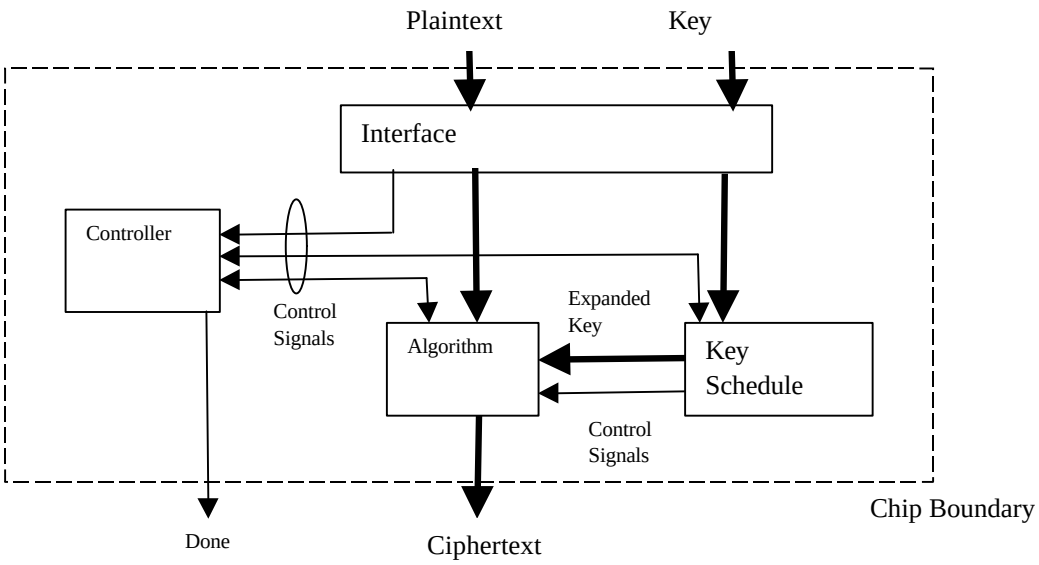


**Figure 3 Top Level Architecture**

The Interface serves to register all data inputs. This is consistent with the hardware design methodology of placing registers at the chip boundary, thus minimizing strict setup-and-hold timing requirements. In some cases, the interface also provides minimum functionality, such as padding keys when appropriate. The Key Schedule performs the generation of subkeys to be used in by the Algorithm block. This includes any required key setup as well as the expansion itself. The Algorithm performs the actual encryption or decryption of data provided from the Interface using the subkeys from the Key Schedule. For iterative implementations, the Algorithm and Key Schedule blocks implement a single round with internal feedback datapaths; whereas the pipelined implementations expand these sections to include as many implementations of a round as required by the particular algorithm. Finally, the Controller provides any necessary control signals for maintaining proper synchronization among the various blocks.

# 6 Algorithm Evaluation

For each of the algorithms, a description of how it was architected for both the pipelined and iterated cases is given. Any nuances of how the rounds were simulated and the key schedule implemented are also given along with specific examples of approaches to reduce redundancy or streamline the design. Each algorithm section then provides block level results of timing constraints versus both chip area and timing in both the iterative and pipelined cases. A table of performance parameters is then provided for four different key sizes, 128 bit key, 192 bit key, 256 bit key, and a hybrid that combines all three key sizes in one key schedule that can be controlled for any particular key size. In some cases, the combined three-in-one key schedule must make compromises to achieve the greater degree of flexibility. Each of the performance parameters is described in more detail in Section 7, along with comparisons across the five algorithms.

Following the architecture for each of the algorithms, each section will provide a summary of the results of the hardware analysis for the individual algorithm. In an effort to save space, the timing and area graphs will be presented for only the combined case which contains all three key sizes in one implementation. Both pipelined and iterated cases will be covered. However, the complete report and design workbooks will contain graphs for all key sizes and contains a much more complete data set. The corresponding tables will capture key performance data points for all key size implementations. *

* Note: At time of publication, not all information was available for every parameter and for every algorithm. Due to some unforeseen difficulties in the amount of time for simulation, some information on area, transistor count and key setup times was not available. This was especially true for simulating the larger blocks in the pipelined cases and for the various key sizes. In addition, certain information for MARS and RC6 was being finalized at time of publication, so is not included in this version. Incomplete data in the following sections are indicated with asterisks.

Complete data for the performance curves and tables of key parameters will be provided on the NIST web site and at the conference.

# 6.1 MARS

## 6.1.1 Architecture

The MARS algorithm requires several different types of rounds[2]. Specifically, there are unkeyed forward mixing, keyed forward transformation, keyed backwards transformation and unkeyed backwards mixing rounds, as well as pre-addition and post-subtraction. The mixture of keyed and unkeyed rounds resulted in the requirement for complex control and data flow operations between the Key Schedule and Algorithm blocks. Specifically, a complex control situation results from the fact that subkeys are required immediately for the pre-addition stage, whereas the next subkeys are not required until the eight unkeyed forward mixing rounds are completed. This architecture presented some unique timing and data synchronization issues.

### 6.1.1.1 Pipelined Key Schedule

As with all pipelined implementations, the subkey from each round must be registered. However, for MARS, the subkeys are not utilized on consecutive clock cycles. Therefore, additional pipelined storage is necessary. The updated key schedule of MARS following AES Round 1 allowed for separating the 40 subkeys into groups of 10. The VHDL model takes advantage of this operation by adding pipelined storage for groups of 10 subkeys, only as long as necessary, rather than creating pipelined storage for all 40 subkeys. This reduced the total number of registers required. Additionally, the pipelined registers are controlled by a latch signal rather than updating on every clock. Again, this reduced the number of registers by removing redundancy.

### 6.1.1.2 Pipelined Algorithm

Relative to the intricacies of the key schedule, the pipelined algorithm implementation is straightforward. It consists of six different types of rounds, each one with its own registered output: one key addition, eight unkeyed forward mixings, eight keyed forward transformations, eight keyed backwards transformations, eight unkeyed backwards mixings and one key subtraction. This makes a total of 34 rounds to complete the algorithm.

### 6.1.1.3 Iterative Key Schedule

The MARS algorithm key schedule generates 10 subkeys at a time. Therefore, the traditional iterative methodology of a single round implementation for generating a single subkey (or set of subkeys as required by a single algorithm round) did not apply. Instead, a single round implementation per 10 subkeys was generated resulting in a key expansion round iterated four times for one encryption cycle. This presents some additional logic overhead for iterative applications in that a “round” generates 10 subkeys simultaneously rather than the exact amount needed by the algorithm at a given stage. (In the case of MARS, two 32-bit subkeys are required per keyed round in the cryptographic core.) In addition to the subkey expansion overhead, there is a storage overhead for the remaining subkeys. Also, decryption requires a full expansion of subkeys prior to beginning data processing. Therefore, the full set of 40 subkeys is stored in registers.

### 6.1.1.4 Iterative Algorithm

The iterative algorithm is consistent with the pipelined algorithm in its relative simplicity when compared to the key schedule. There is a single register for all rounds. The input to the register depends on the round number. For example, the input for the first round of encryption is the key addition round result; for the second round it is the unkeyed forward mixing round result and so on.

Subkeys are presented to the algorithm block as an array of all 40 subkeys. This differs from other iterative algorithm implementations that present only one subkey at a time. The rationale for this design was to eliminate duplicate logic in both the Key Schedule block and Algorithm block. Due to the timing gaps in the application of subkeys and the fact that all 40 subkeys are generated prior to decryption processing, it was considered advantageous to allow a 40 element bus to connect the two blocks.

### 6.1.2 MARS Top Level Results

#### 6.1.2.1 Timing and Area

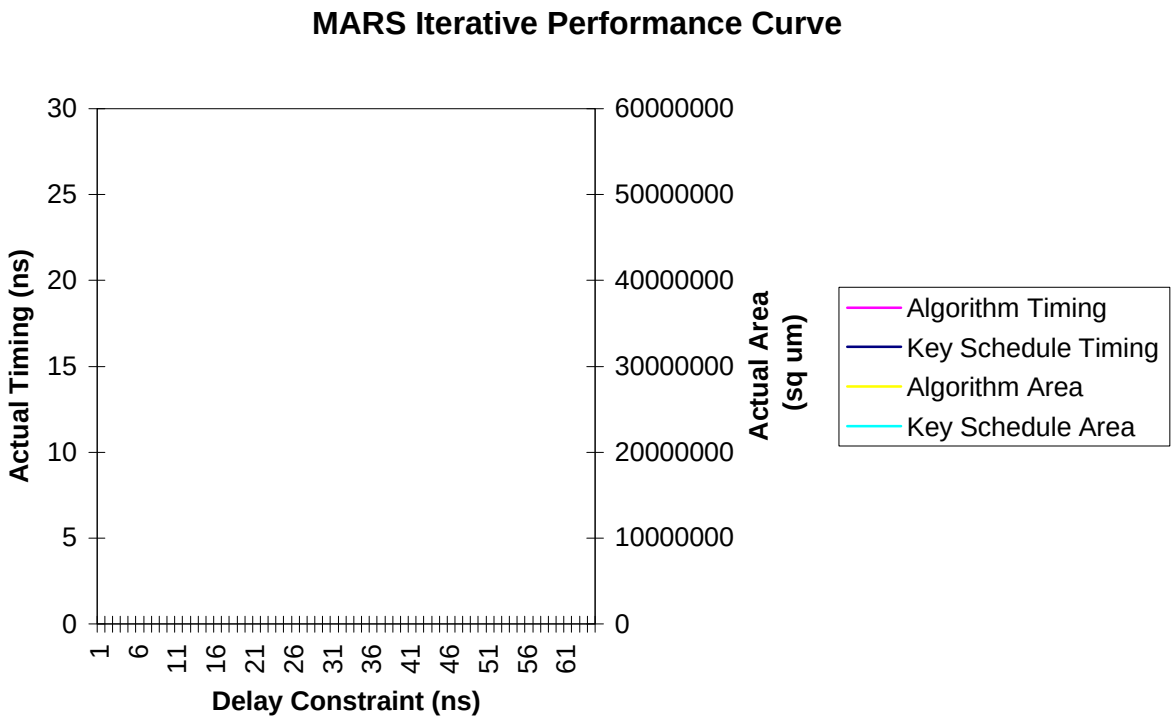


Figure 4

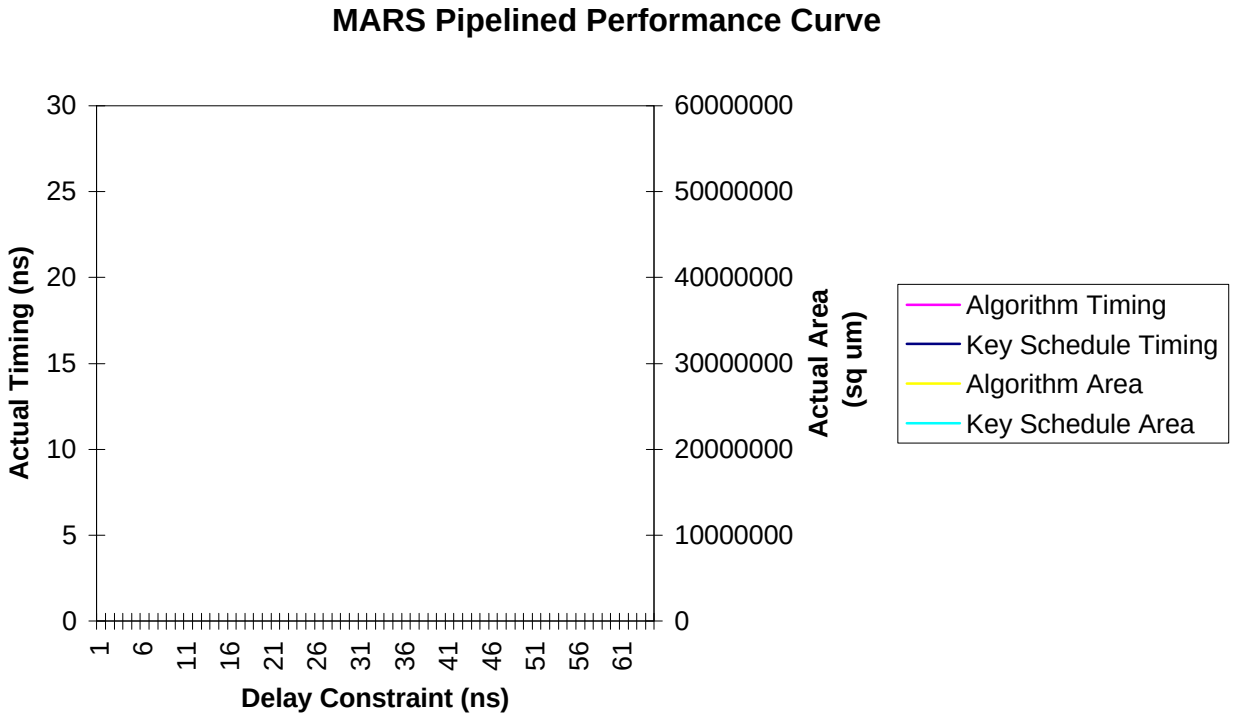


Figure 5

#### 6.1.2.2 Key Parameters

| Parameter | Iterative 3in1 | | Pipelined 3in1 | |
|---|---|---|---|---|
| | Min. | Max. | Min. | Max. |
| Area (um2) | * | * | * | * |
| Transistor Count | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | * | * | * | * |
| Key Setup Time Encrypt (ns) | * | * | * | * |
| Key Setup Time Decrypt (ns) | * | * | * | * |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | * | * | * | * |
| Time to Decrypt One Block(ns) | * | * | * | * |

**Table 1 MARS Summary**

## 6.2 RC6

### 6.2.1 Architecture

The following provides a high level description of the major blocks in the RC6 algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[3].

#### 6.2.1.1 Pipelined Key Schedule

The RC6 key schedule is pipelined using a slightly different method than the other algorithms. Since a significant number of computations for the key schedule are required before any expanded keys are generated, the architecture takes advantage of the run-up by performing the expansion at the start of the pipeline. Only a single copy of the expansion hardware is required, but additional registering is needed to maintain the keys on a time dependent basis, discarding keys from previous stages (i.e., the keys have already been used). Keys are then passed from register to register to follow the data in the pipeline.

#### 6.2.1.2 Pipelined Algorithm

The algorithm "unrolls" the stages of the algorithm into a pipeline, following the algorithm description for function ordering and naming conventions. Combination functions are used to perform cases where distinct operations need to be performed in encrypt and decrypt. For example, the pre-add will contain both addition and subtraction to accommodate both cases. A similar condition exists in the algorithm round function, with slightly different functions needed for encrypt and decrypt. However, synthesis optimization can take advantage of common operations, such as the multiply, to reduce the total number of operators needed.

#### 6.2.1.3 Iterative Key Schedule

The iterative key schedule is designed to perform a single round of expansion per clock. Expanded keys are fed to the algorithm block after the controller initiates a start signal. However, the key setup has been designed to compute single or multiple steps of the run-up in a single clock, depending on the performance needed by the rest of the system. A load cryptovariable (i.e., load key) signal from the controller will initiate the key setup. Once complete, the expansion can be started.

#### 6.2.1.4 Iterative Algorithm

The RC6 iterative algorithm closely reflects the pipelined version. The same round instance is called repeatedly to process input data. The encrypt and decrypt are symmetrical with respect to operations performed in a similar manner (e.g., pre-add, round, post-add), so the same block can be called without additional overhead.

### 6.2.2 RC6 Top Level Results

#### 6.2.2.1 Timing and Area

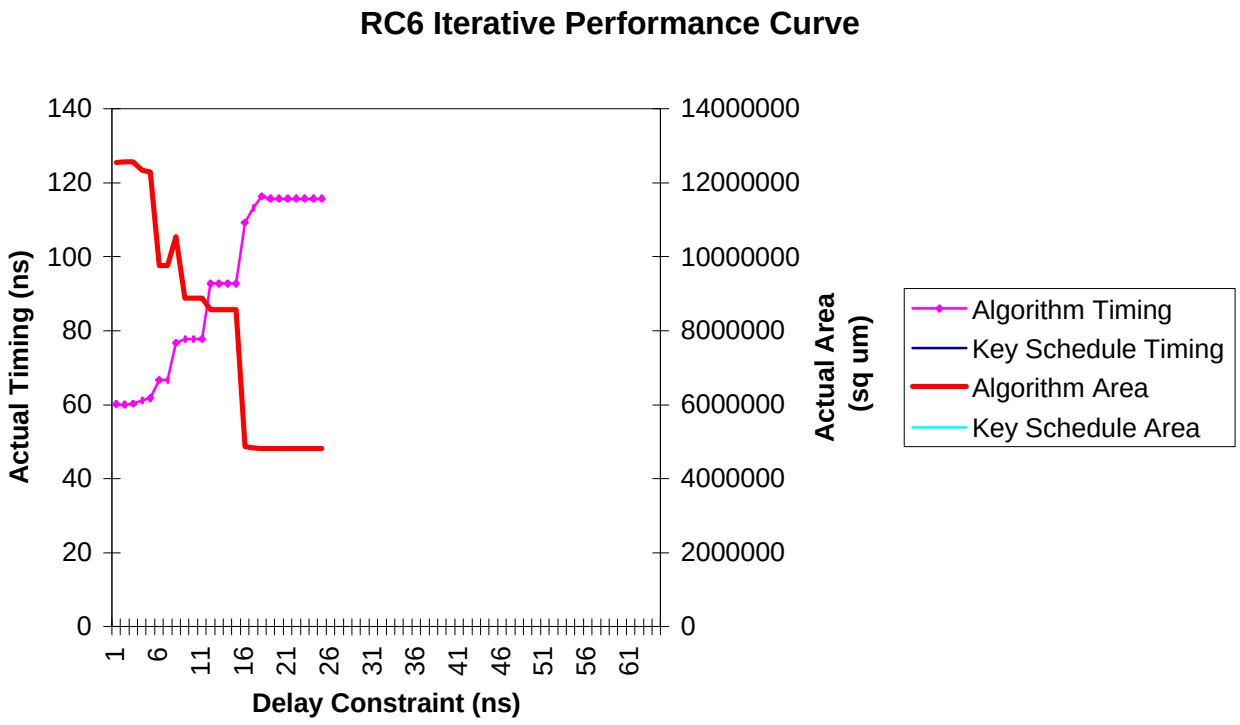


**Figure 6**

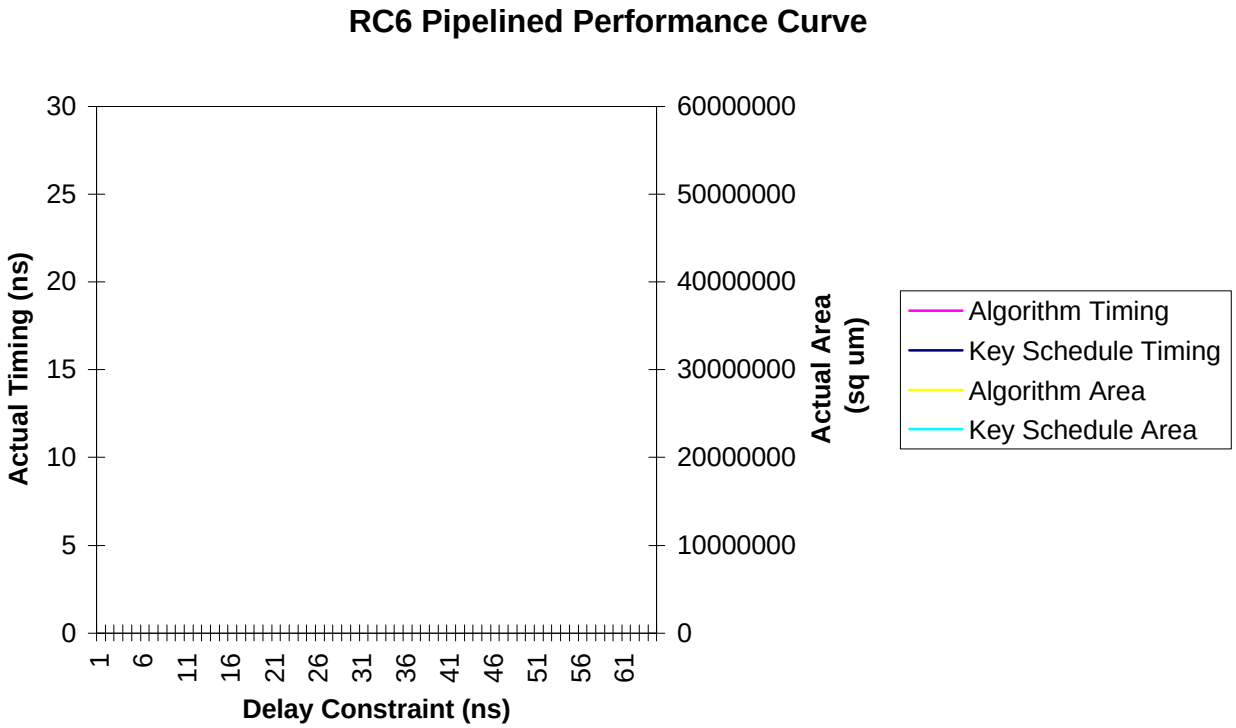


**Figure 7**

#### 6.2.2.2 Key Parameters

| Parameter | Iterative 3in1 | | Pipelined 3in1 | |
|---|---|---|---|---|
| | Min. | Max. | Min. | Max. |
| Area (um2) | * | * | * | * |
| Transistor Count | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | * | * | 1192.00 | 2171.00 |
| Key Setup Time Encrypt (ns) | * | * | * | * |
| Key Setup Time Decrypt (ns) | * | * | * | * |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | * | * | 1179.2 | 2146.8 |
| Time to Decrypt One Block(ns) | * | * | 1179.2 | 2146.8 |

**Table 2 RC6 Summary**

## 6.3 RIJNDAEL

### 6.3.1 Architecture

The following provides a high level description of the major blocks in the RIJNDAEL algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[4].

#### 6.3.1.1 Pipelined Key Schedule

The RIJNDAEL key schedule is based on a sliding window approach as described in the algorithm specification. Multiple key sizes are based on the n-1 element and the n-k element (32 bit word organized), where k is 4,6, or 8, depending on key size. The key expansion is a linear combination of the elements, so a similar function can be used on the decrypt function to "unexpand" the keys in a reverse direction. Such an approach allows for an increase in the key agility without sacrificing significant amounts of area to store all of the expanded keys.

The encryption expansion can start immediately, with the first words of the initial key being used as expanded key. The setup time for this case is zero. During the decryption, the key is expanded to the last key, stored, and then the pipeline is run to create the previous expanded key until the last decrypt key is generated, which is the initial key. Keys are generated at a rate of four 32 bit words per round, regardless of key size, to keep up with the requirements of the algorithm block. Additional registers are used to maintain sufficient previous keys to generate the next four words of expanded key.

Keys are pulled from the bank of registers which make up the sliding window. S-Boxes are re-used, without a performance penalty, to minimize the size impact of having additional S-Boxes.

#### 6.3.1.2 Pipelined Algorithm

The RIJNDAEL algorithm pipeline consists of a sequential mapping of the steps of the algorithm to registered stages in hardware. Each stage reflects a single round of the algorithm. The primary advantage to pipelining in this manner is the significant increase in throughput. RIJNDAEL was architected such that both the encrypt and decrypt functions could be performed with the same pipeline. This approach needed a static pipeline that could perform both functions, so the algorithm round functions contained in the package will serve a dual role by providing cases for encrypt and decrypt within the same function. The pipeline structure reflects changes in direction, such as requiring a pre-add on the encryption (first round) versus decryption requiring a post-add on the last round.

#### 6.3.1.3 Iterative Key Schedule

The iterative version of the key schedule focuses on reducing the area of the key expansion, so only a single copy of the expansion is maintained. For encryption, as in the pipelined case, the expansion starts immediately, with no key setup required. The keys are expanded every round, producing the four 32 bit words of key required. As each new key is created and stored, the old key is overwritten.

In the case of decryption, the algorithm requires a setup time to effectively run the algorithm to the last key. This serves as the starting point for all decryptions using that key. This value will also be stored so it can be referenced on each new decryption to eliminate key setup for every new decryption.

#### 6.3.1.4 Iterative Algorithm

The algorithm block uses the same functionality as described in the pipeline but does not re-use some of the combination functions used to construct the pipeline. Instead, the function calls are made explicitly, depending on encryption/decryption to provide the widest possible range of hardware re-use. The function calls in the encrypt and decrypt directions are not symmetrical. The algorithm processes the state data on each round, performing only one step of the algorithm per round.

### 6.3.2 RIJNDAEL Top Level Results

#### 6.3.2.1 Timing and Area

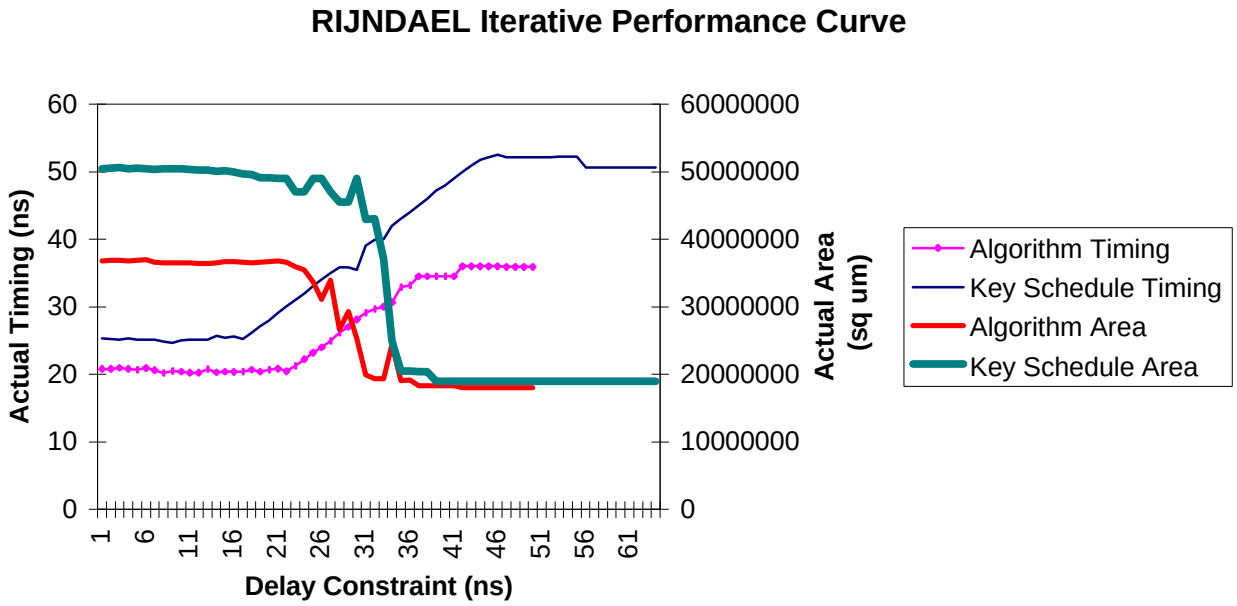


**Figure 8**

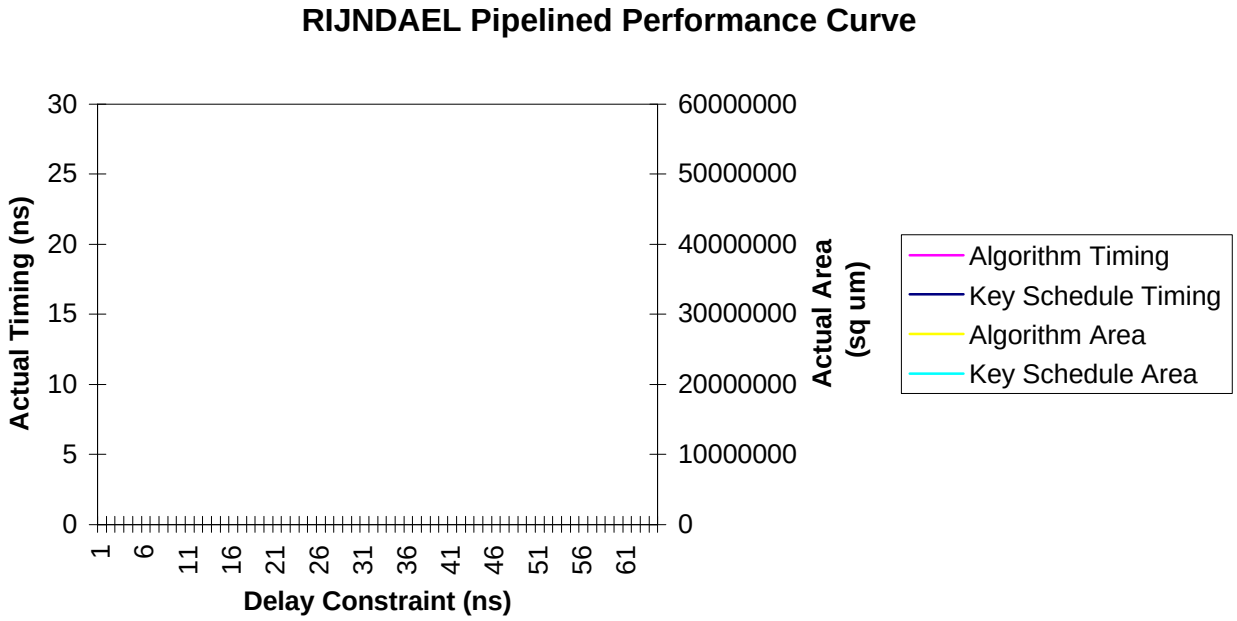


**Figure 9**

#### 6.3.2.2 Key Parameters

| Parameter | Iterative 3in1 | | Pipelined 3in1 | |
|---|---|---|---|---|
| | Min. | Max. | Min. | Max. |
| Area (um2) | 37034346.00 | 81661400.00 | * | * |
| Transistor Count | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | 371.06 | 519.48 | 4060.00 | 5163.00 |
| Key Setup Time Encrypt (ns) | 0.00 | 0.00 | 0.00 | 0.00 |
| Key Setup Time Decrypt (ns) | 246.4 | 344.96 | 0 | 277.92 |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | 493.8 | 346.36 | 247.4 | 346.36 |

**Table 3 RIJNDAEL Summary**

## 6.4 SERPENT

### 6.4.1 Architecture

The following provides a high level description of the major blocks in the SERPENT algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[5].

#### 6.4.1.1 Pipelined Key Schedule

The SERPENT algorithm implements a simple expansion function for the key scheduling. The exclusive-or based function allows for quick computation and does not require key setup in the encrypt direction. Pipelining is maximized as this approach utilizes a sliding window approach, where only a small number of previous expanded keys are needed to compute the next sub-keys. However, for decryption, a key setup time is required to compute the starting point for the key expansion, which is the last set of W registers. The decrypt pipeline computes the previous set of W registers based on the current set, as the exclusive-or based expansion can be reversed easily. To save storage in this design, the keys are computed at each stage, with the decrypt case requiring a block of logic at the beginning to find the last subkeys.

The SERPENT pipelined key schedule provides two successive keys to each round of the algorithm on expansion. The algorithm will select the correct key based on the current encryption/decryption mode. The additional key allows for the rounds that require two keys to operate.

#### 6.4.1.2 Pipelined Algorithm

The pipelined SERPENT algorithm block contains a structural model of the unraveled rounds of the algorithm. Four distinct functions are needed to implement both the encrypt and decrypt operations. The core algorithm round functions are the same for 30 rounds of the algorithm, with an internal mux/demux to select the encrypt or decrypt mode. The first two rounds of encrypt and last two rounds of decrypt distinguish the cases where the pipeline is re-routed. The encrypt will bypass the two special rounds of the decrypt while the decrypt will bypass the two special rounds of encrypt. The latency will remain the same as no extra rounds are added. The pipeline will select and re-route based on the current mode of encryption or decryption.

#### 6.4.1.3 Iterative Key Schedule

The SERPENT iterative key schedule uses a single copy of the expansion function to generate the sub-keys, one at a time. Area can be significantly reduced using the same hardware repeatedly. Additional key setup will be required in the decrypt direction to allow for the run-up to the last key of the expansion.

#### 6.4.1.4 Iterative Algorithm

The iterative algorithm uses the same functions as the pipeline, with the same round instance referenced repeatedly to perform the main processing of the algorithm. The special case rounds are selected by the state machine within the iterative block to determine encrypt/decrypt direction, and consequently, which pre/post add functions to perform.

### 6.4.2 SERPENT Top Level Results

#### 6.4.2.1 Timing and Area

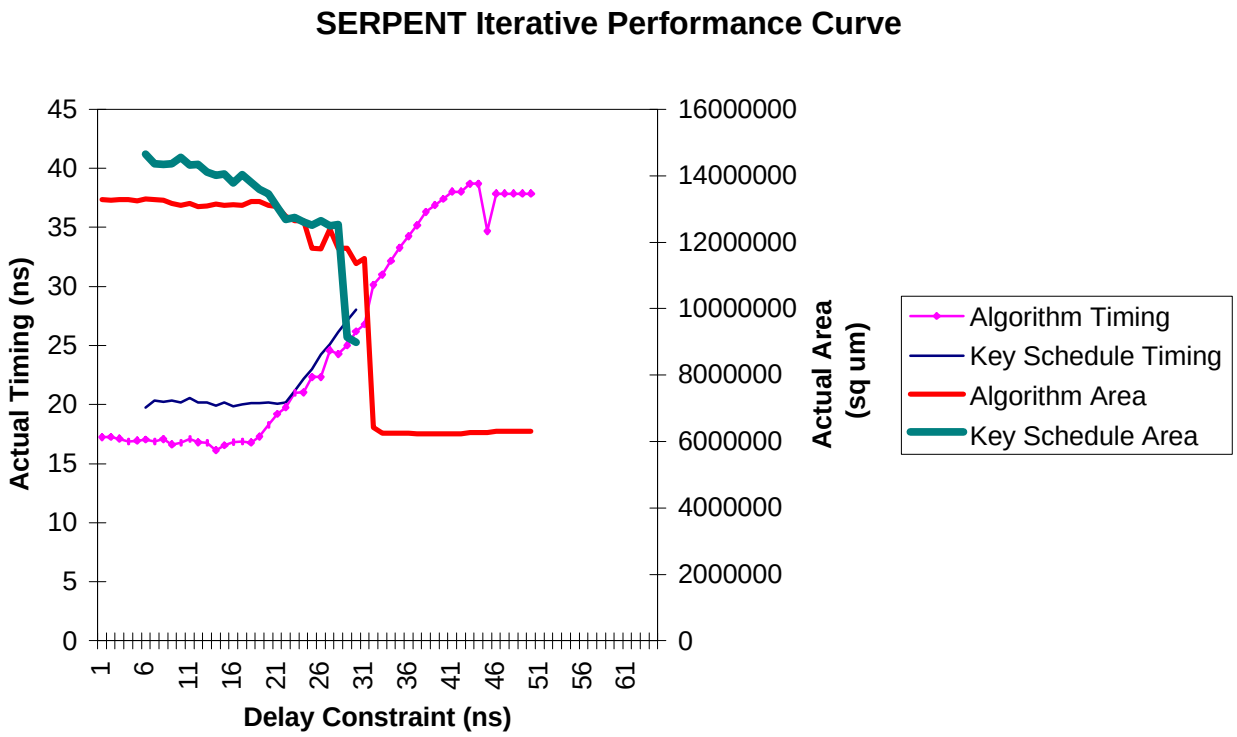


**Figure 10**

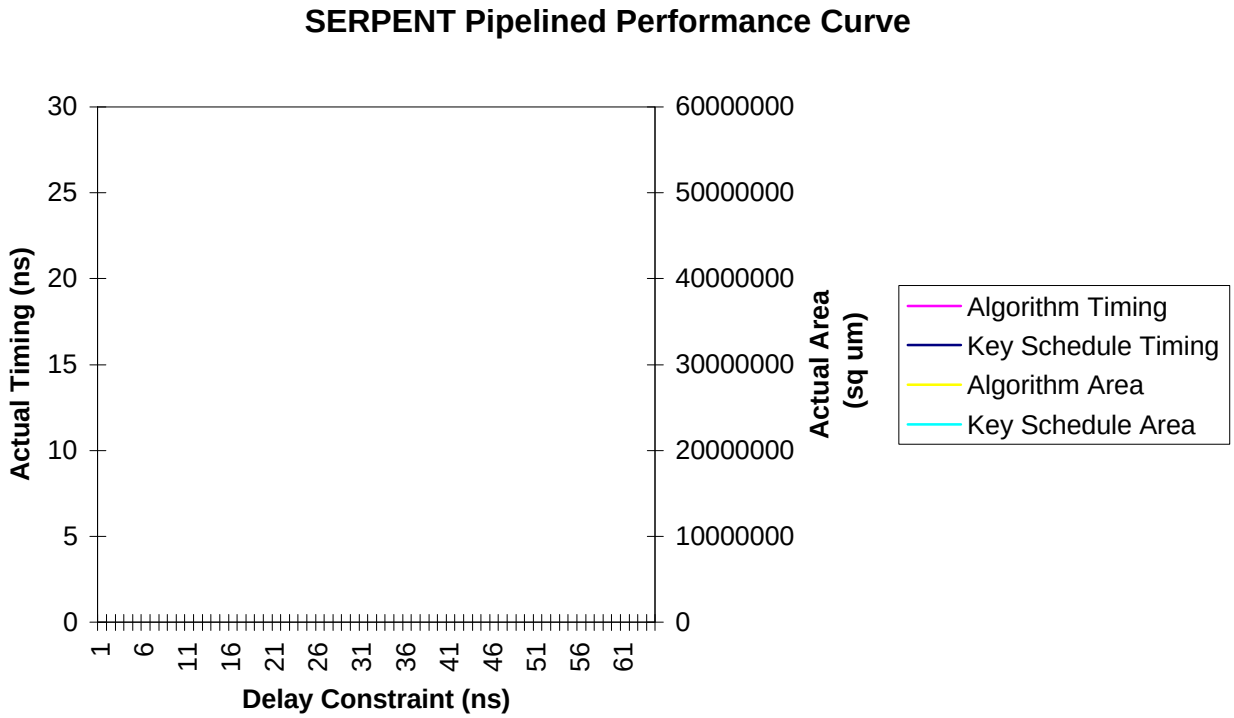


**Figure 11**

#### 6.4.2.2 Key Parameters

| Parameter | Iterative 3in1 | | Pipelined 3in1 | |
|---|---|---|---|---|
| | Min. | Max. | Min. | Max. |
| Area (um2) | * | * | * | * |
| Transistor Count | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | * | 202.33 | 5298.01 | 8030.11 |
| Key Setup Time Encrypt (ns) | 19.77 | * | 6.74 | 11.76 |
| Key Setup Time Decrypt (ns) | 672.18 | * | 212.55 | 365.58 |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | 632.64 | * | 510.08 | 773.12 |
| Time to Decrypt One Block(ns) | 632.64 | * | 510.08 | 773.12 |

**Table 4 SERPENT Summary**

## 6.5 TWOFISH

The following provides a high level description of the major blocks in the TWOFISH algorithm. Details of the components, sweeps, and their implementations can be found in the design workbook[6].

### 6.5.1 Architecture

A useful property of the TWOFISH architecture was the relatively large amount of re-use of design blocks. Both the Key Schedule and Algorithm utilized many of the same functions. While this does not result directly in a direct increase in performance, since key expansion and encryption are performed in parallel, it does simplify the hardware coding process. As stated, common coding techniques and processes were used for developing each algorithm design resulting in areas available for improvement in a more highly optimized design. In the case of TWOFISH, a smaller design could be created by taking advantage of the function re-use. However, as with most hardware trade-offs, this area optimization would come at the expense of performance and complex control mechanisms.
Another feature of TWOFISH is the lack of initial key runup prior to subkey expansion. In addition, the key schedule is not a feed-forward design. Each round of key schedule is independent of the previous round. This unique characteristic allowed for a Key Schedule that does not require a setup time for either encryption or decryption.
In the TWOFISH algorithm, the first step of encryption is a pre-whiten function, In hardware, this is simply an exclusive-OR. The pre-whiten step is performed during the same clock cycle as the first subkey expansion which generates the pre-whiten subkey. This was possible because the XOR function did not create a critical path concern since the main algorithm rounds incorporate an integer addition that is more complex. The result was the ability to load data and key in the same clock cycle, thereby reducing the overall time for encryption by one clock cycle.

#### 6.5.1.1 Pipelined Key Schedule

In order to allow for either encryption or decryption, both pre-add and post-add subkeys are generated during the first pipeline stage. The post-add key is buffered through a pipelined delay until needed in the final processing step. Also, since one of the input parameters is the round number which is fixed for a given pipelined round there is an optimization or pre-calculation in each pipelined round.

#### 6.5.1.2 Pipelined Algorithm

The algorithm is an efficient unrolling of stages because encryption and decryption are nearly identical. In addition, the symmetry allows for similar processing in both the encrypt and decrypt directions.

#### 6.5.1.3 Iterative Key Schedule

As in the pipelined key schedule, the iterative design requires buffering of the post-add subkey until it is needed in the final processing step. However, this buffering is not required to be implemented in pipelined stages. The key schedule round is generalized such that the round number is not a fixed constant as in the pipelined case. This does not allow synthesis optimization of each round, but does save area since only one hardware block is instantiated.

#### 6.5.1.4 Iterative Algorithm

The differences between encryption and decryption are minor such that the additional hardware to support either process in a single round adds minimal area.

### 6.5.2 TWOFISH Top Level Results

#### 6.5.2.1 Timing and Area

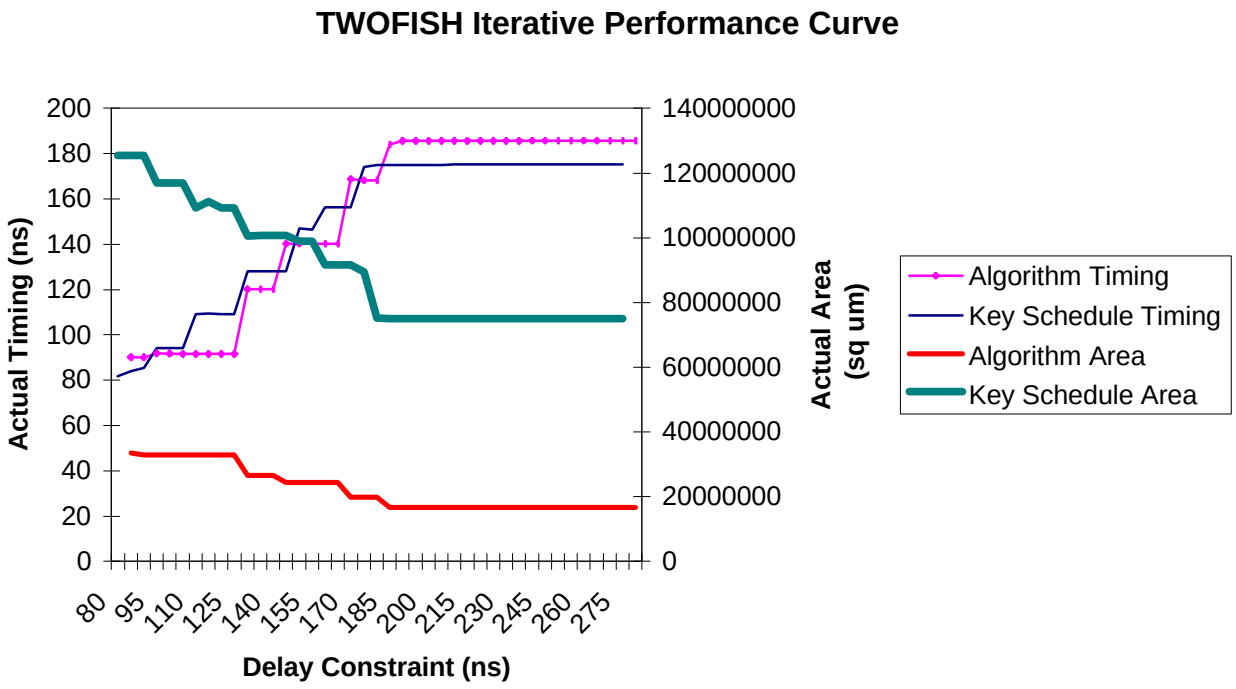


**Figure 12**

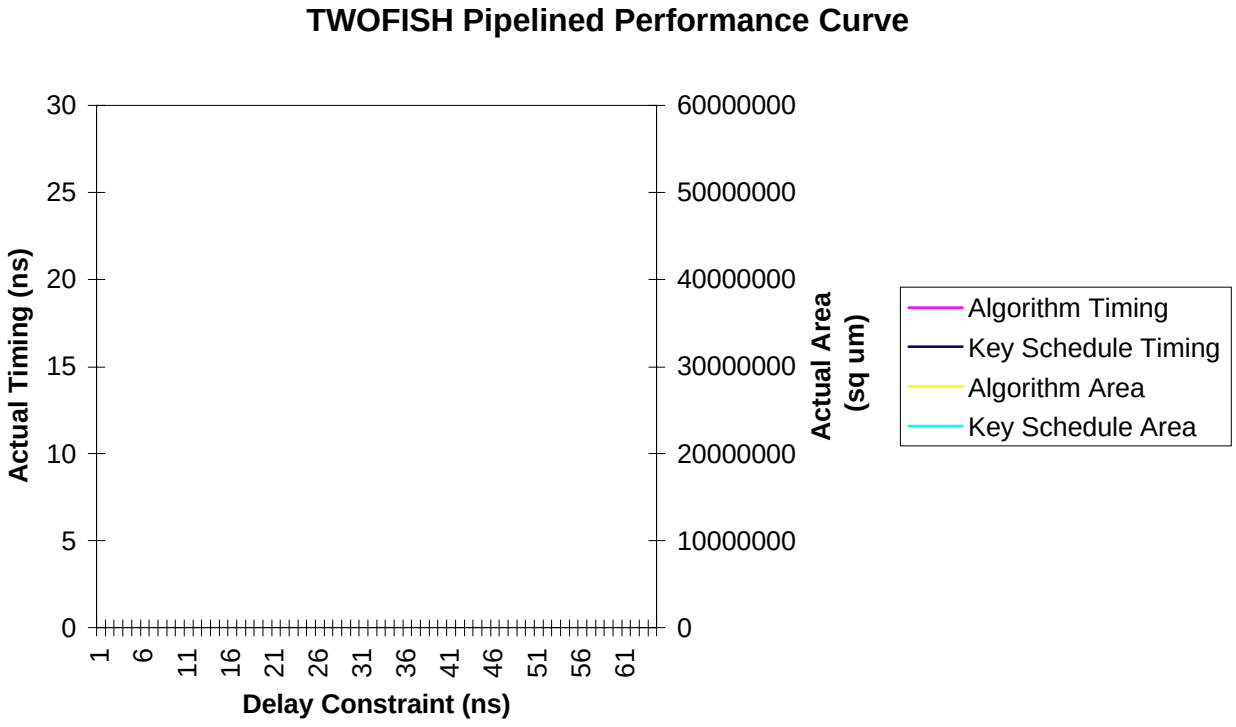


**Figure 13**

#### 6.5.2.2 Key Parameters

| Parameter | Iterative 3in1 | | Pipelined 3in1 | |
|---|---|---|---|---|
| | Min. | Max. | Min. | Max. |
| Area (um2) | 91686840 | 158300076 | * | * |
| Transistor Count | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | 38.29 | 79.00 | * | 1445.55 |
| Key Setup Time Encrypt (ns) | 0 | 0 | 0 | 0 |
| Key Setup Time Decrypt (ns) | 0 | 0 | 0 | 0 |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | 1620.18 | 3342.6 | 1593.9 | * |
| Time to Decrypt One Block(ns) | 1620.18 | 3342.6 | 1593.9 | * |

**Table 5 TWOFISH Summary**

# 7 Performance Results

A table summarizing the results and performance metrics is given below for algorithm comparison. These comparison values are given only for the combined key size implementation, which implements a selectable 128 bit, 192 bit, and 256 bit key in the same implementation.

| Parameter | Algorithm | | | | |
|---|---|---|---|---|---|
| | MARS | RIJNDAEL | RC6 | SERPENT | TWOFISH |
| Area (um2) | * | * | * | * | * |
| Transistor Count | * | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | * | 519 | * | 202 | 79 |
| Key Setup Time (ns) | * | * | * | * | * |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | * | 494 | * | 633 | 1620 |
| Time to Decrypt One Block(ns) | * | 494 | * | 633 | 1620 |

**Table 6 Iterated Summary**

| Parameter | Algorithm | | | | |
|---|---|---|---|---|---|
| | MARS | RIJNDAEL | RC6 | SERPENT | TWOFISH |
| Area (um2) | * | * | * | * | * |
| Transistor Count | * | * | * | * | * |
| Input/Outputs Required | 520 | 520 | 520 | 520 | 520 |
| Throughput (Mbps) | * | 5163 | 2171 | 8030 | 1445 |
| Key Setup Time (ns) | * | * | * | * | * |
| Algorithm Setup Time (ns) | 0 | 0 | 0 | 0 | 0 |
| Time to Encrypt One Block (ns) | * | 247 | 1179 | 510 | 1594 |
| Time to Decrypt One Block(ns) | * | 247 | 1179 | 510 | 1594 |

**Table 7 Pipelined Summary**

# 8 Summary

This paper has presented an overview of the methods and architectures used for the AES hardware comparison. The primary characteristics used for design tradeoffs in hardware engineering are area and timing. As such, each algorithm was examined from the standpoint of minimum area (iterative architecture) and maximum throughput

(pipelined architecture). Further, statistics and data based on area and timing were emphasized and illustrated for each algorithm.
The results (in Section 7) show vital parameters for both the iterative and pipelined architectures of each algorithm that can be used to evaluate relative performance. The designs were not optimized for any one parameter, but rather they serve as a good testbench scoring of all the algorithms relative to one another, given the same commonly used hardware design practices and procedures. Key performance data points to highlight are minimum transistor count and maximum throughput. *
It should be emphasized that any data point based on a single parameter (e.g. transistor count or throughput) is a relatively narrow view of the algorithm's overall performance or rating. For this reason, there was no attempt to rank algorithms in order. Rather, it is left to the cryptographic community to establish a consensus of the most important parameters – in combination or alone – and to draw appropriate conclusions from the data provided herein.

* Note: Because incomplete information was available at publication time, additional results will be updated and provided to the community through NIST as the parameter information is filled in for all algorithms.

# 9 Acknowledgements

The authors would like to thank others at NSA for providing support to this project and in editing this report, namely Mr. Jeff Ingle. The authors also appreciate the opportunity NIST gave in encouraging this analysis, since they feel that high performance implementations for high-speed networks are an important aspect of the AES competition.

# 10 References

[1] W. Semancik, L. Mercer, T. Hoehn, G. Rowe, M. Smith-Luther, R. Agee, D. Fowlkes, and J. Ingle , "Cell Level Encryption for ATM Networks and Some Results from Initial Testing," , *DoD Fiber Optics Conference*, March 1994.

[2] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, M. Peyravian, D. Safford and N. Zunic, "Mars – a candidate cipher for AES," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

[3] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6™ Block Cipher," *First Advanced Encryption Standard (AES) Conference*, Ventura, CA, 1998.

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

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

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