Project Plan Report for EC772, Spring 2013

25 February 2013

Ben Humphries

Zhonghan Liu

Michael Pyrch

Florian Raudies

By

Advanced Encryption Standard with Electronic Codebook and Counter Modes

**Abstract**

The goal of this project is to build a chip that can perform streaming AES (Advanced Encryption Standard) encryption and decryption in both ECB (Electronic Codebook) and CTR (Counter) modes. The primary parts of the design include input/output shift logic, key expansion logic, an encryption pipeline, a decryption pipeline, and the control logic for running in different modes. This design emphasizes speed over space by unrolling the design as much as possible and tuning the design for lowest latency through the chip.

TBD – put some actual latency information here.

Contents

[1. Introduction 1](#_Toc349551553)

[2 Design 3](#_Toc349551554)

[2.1 Core Top 3](#_Toc349551555)

[2.2 Stream In and Stream Out 3](#_Toc349551556)

[2.3 Key Expansion 5](#_Toc349551557)

[2.4 Cipher Core 8](#_Toc349551558)

[2.4.1 AES Encrypt and Decrypt Pipelines 9](#_Toc349551559)

[2.4.1 AES Encrypt and Decrypt Rounds 9](#_Toc349551560)

[2.5 Physical Design and Layout 11](#_Toc349551561)

[3. Design Verification 12](#_Toc349551562)

[3.1. Verification Design 12](#_Toc349551563)

[3.2. Pre-Synthesis Verification 13](#_Toc349551564)

[3.2.1. Black Box Testing 13](#_Toc349551565)

[3.2.2. White Box Testing 13](#_Toc349551566)

[3.3. Post-Synthesis Verification 14](#_Toc349551567)

[4.1 Labor 15](#_Toc349551568)

[5. Conclusion 16](#_Toc349551569)

[5.1 Accomplishments 16](#_Toc349551570)

[5.2 Uncertainties 16](#_Toc349551571)

[5.3 Ethical considerations 16](#_Toc349551572)

[5.4 Future work 16](#_Toc349551573)

[References 17](#_Toc349551574)

[Appendix A Requirement and Verification Table 18](#_Toc349551575)

# 1. Introduction

We propose a chip design for the cipher and decipher operation using the Advanced Encryption Standard (AES) with a 128-bit key. This is a symmetric key encryption method. It uses the same key for the cipher and decipher operation. Various features, like nonlinearity and asymmetries add to the strengths of the code against known attacks. Our aim is to have this design be faster compared to software implementations.

The cipher operation is composed of four operations, which are performed in a round or iteration. These are: (i) sub-bytes transform, (ii) shift row transform, (iii) mix columns transform, and (iv) add round key. All four operations are defined as bit-wise operations using the Galois field GF(28). For the 128-bit key packages of 16 Bytes are organized in a 4 × 4 matrix in column-major order, whereas each element has eight bits. The sub-bytes transform takes, first the inverse element using the polynomial m(x) = x8+x4+x3+x+1 and, second, applies a matrix-vector multiplication with a matrix defined by eight cyclic shifts of the polynomial m(x) (Daemen & Rijndael, 1999; p. 11). This sub-bytes transform is applied to each of the 8-bit elements of the 4 × 4 matrix. Due to the computational complexity this operation is realized by a lookup table. For our 128-bit key the table requires 256×8 bit = 256 Bytes. The shift-row transformation shifts the bytes of the 0th row of the 4 × 4 matrix by zero, the 1nd row by one, the 2nd row by two, and the 4th row by four. The mix-column operation is defined by a multiplication with the matrix [02, 03, 01, 01; 01, 02, 03, 01; 01, 01, 02, 03; 03, 01, 01, 02] with the vector [a0, a2, a3, a4] whereas a0 is the 0th column of the 4 × 4 matrix, a1 the 1st column, a2 the 2nd column, and a3 the 3rd column. The matrix entries represent two hex digits or eight bits that are multiplied with the eight bits from the vector components a0, a1, a2, and a3, respectively. The add-round-key operation adds the corresponding part of the key (in column-major order) to the each entry of the 4 × 4 matrix. Note that addition (and subtraction) is a bit-wise XOR operation. For the first nine rounds all four operations are performed. The last round excludes the mix columns operation. Mali et al., (2005) shows a nice flow diagram. The key will be externally provided and expanded on chip.

For the decipher operation the following differences apply. The organization of the rounds is identical, leaving out the inverse mix columns transform in the 10th round. For the inverse sub-bytes transform the inverse table is applied, which gives an overall table size of 512 Bytes. The inverse shift row transform is a cyclic shift of the 0th row by zero, the 1st row by three, the 2nd row by two, and the 3rd row by one. The inverse mix columns transform uses a multiplication with the matrix [0E, 0B, 0D, 09; 09, 0E, 0B, 0D; 0D, 09, 0E, 0B; 0B, 0D, 09, 0E]. The add-round-key operation is the same bit-wise XOR as in the cipher operation. Although having large similarities with the forward transform operations, these will have their own modular implementation.

The key-expansion uses 44 iterations each comprised of an XOR operation and every 4th iteration has an additional sub-bytes transform and cyclic shift of bytes in a 4-byte word. A loop unrolling strategy may be used to match the 10 rounds of the cipher or decipher operation.

An exhaustive test of the entire design is not feasible because of the 22×128 = 1077 possible states. Thus, we plan to test individual modules of a hierarchical design through a scan chain. The entire design will be tested by probing and comparing the results with test data generated by a C reference implementation (Rijmen et al., 2000).

Based on the targeted clock and number of clock cycles we will compare the projected runtime of the chip implementation with the runtime of the C reference implementation. In addition, we will compare our implementation to the runtime of other published results of hardware implementations of AES (e.g. Mali et al., 2005).

# 2 Design

For most encryption designs, speed is one of the most important factors; therefore this design chose low latency at all costs of space, power, etc. This design also includes running AES in both ECB and CTR modes for added security. Both ECB and CTR can be encrypted and decrypted fully in parallel, and as a result the latency for either one is the same. The primary difference between the modes is how the crypto pipelines are used and an extra operation after the encryption. The design will only allow for one mode, one key, and one initialization vector to be loaded at a time. The design allows for a single stream of data to be encrypted and decrypted. If multiple keys, modes, or initialization vectors are to be used, the circuit must be reset to accommodate such behavior.

## 2.1 Core Top

The Core Top module, shown in Figure 2. 1, is the top level module for the design. It is the structural Verilog code the houses the Stream In, Stream Out, Cipher Core, and Key Expand modules. The major interfaces at the Core Top level are the data in, data out, crypto mode, crypto ready, clock, and reset.

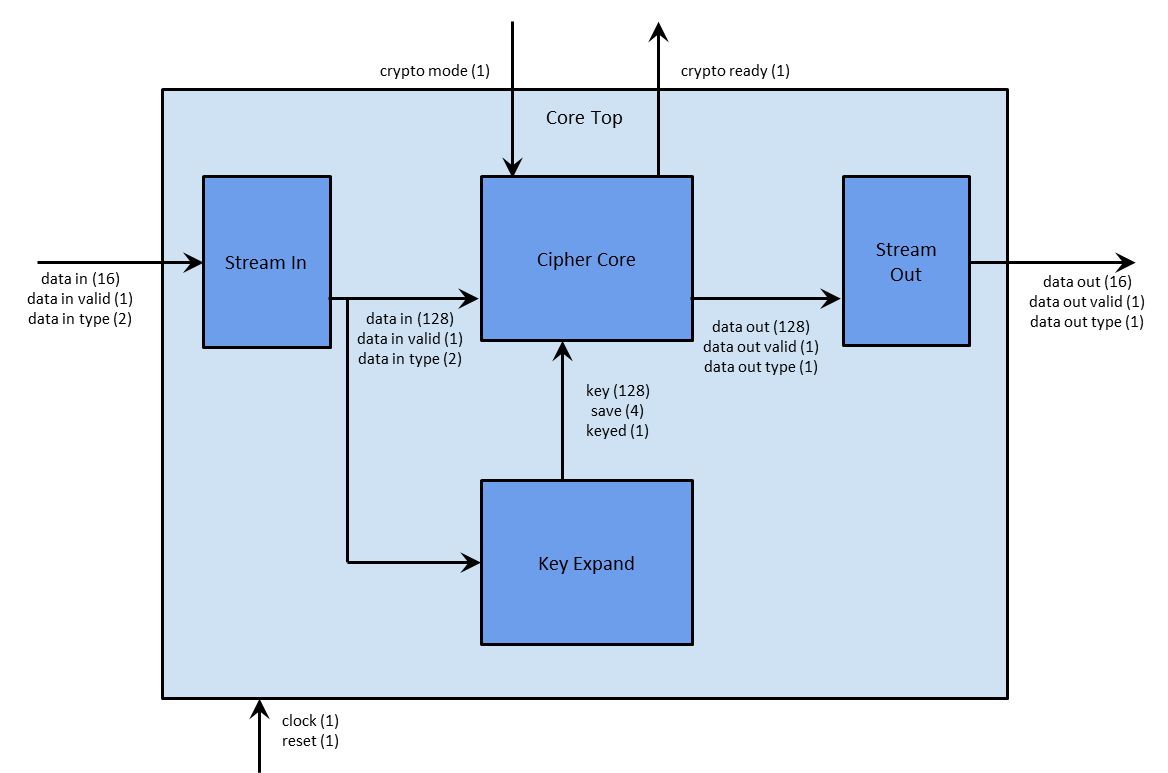


Figure 2. : Core Top Module

## 2.2 Stream In and Stream Out

The Stream In module, shown in Figure 2. 2, provides the interface for getting plaintext, ciphertext, keys, and initialization vectors into the design. The Stream Out, shown in Figure 2. 3, module provides the interface for getting plaintext and ciphertext out of the design. Both interfaces are comprised of a 16 bit data line, a valid signal, and a type signal. Once valid is asserted the data output is valid and should be saved by the receiving side. The type field, 2 bits to the Stream In and 1 bit from the Stream Out, tells the receiver the data type. Table 1 shows the encoding of the type field for both the Stream In and Stream Out sides. All transmitted and received data from these blocks are in 128 bit block sizes; which means that a full block of data is comprised of 8 clocks of data transfer. During these eight clocks, the type field should be kept constant to what the data field is, even if the valid frames are not back to back.

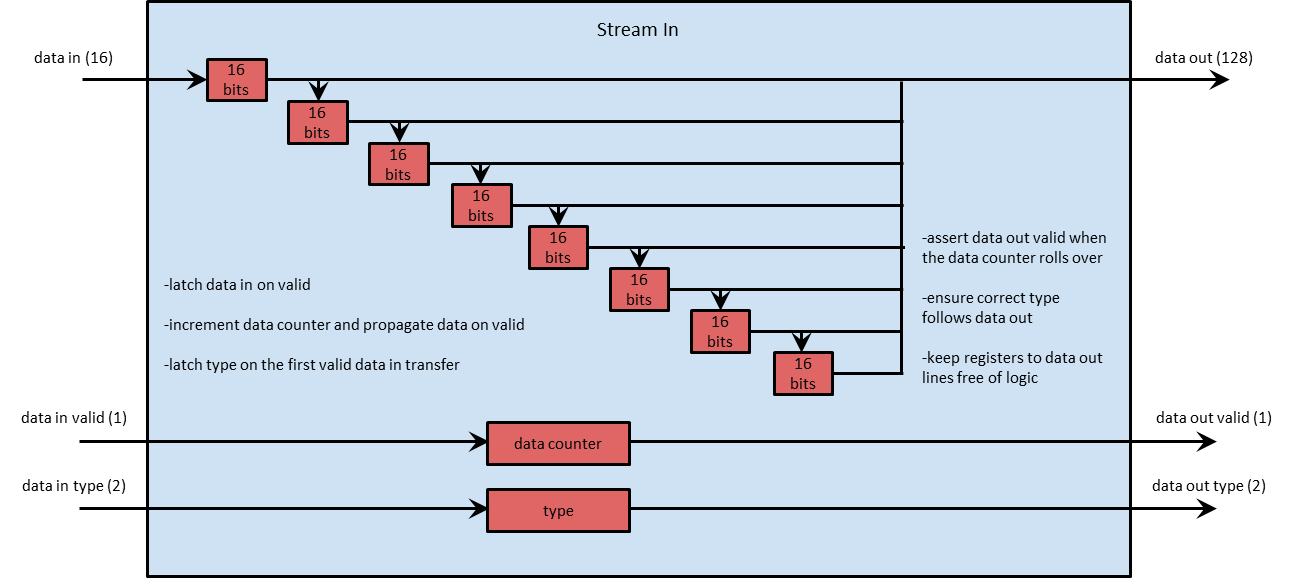


Figure 2. : Stream In Module

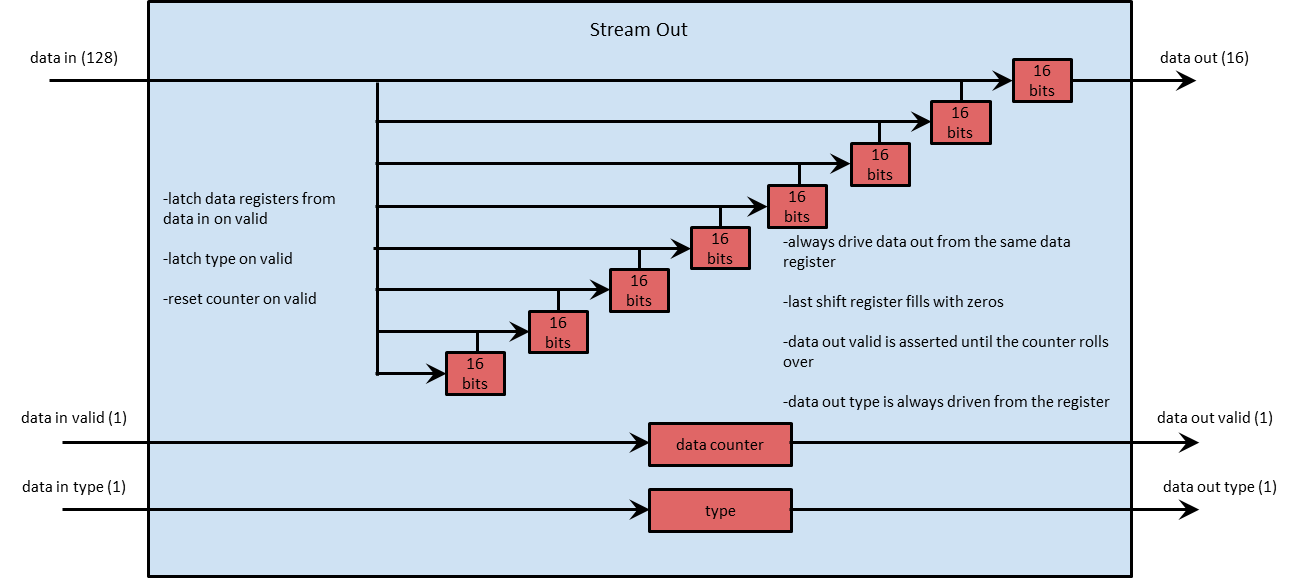


Figure 2. : Stream Out Module

Table : Stream In and Stream Out Type Encoding

|  |  |  |
| --- | --- | --- |
|  | | |
| **Value** | **Stream In Encoding** | **Stream Out Encoding** |
| 00 | Plaintext | Ciphertext |
| 01 | Ciphertext | Plaintext |
| 10 | Key | N/A |
| 11 | Initialization Vector | N/A |

## 2.3 Key Expansion

One 128 bit key is provided by the user; however, encrypt and decrypt operations use ten rounds each and each of these ten rounds uses a different key. This requires a schema for the expansion of the 128 bit key as indicated in Figure 2. 4. We show pseudo-code for this expansion operation in Figure 2. 5. During the initial phase the user provided key is copied into four word-sized (32bit) subsequent fields of an array, which lines 2-5 show. In nine subsequent iterations all entries of multiples of four include a sub bytes, a byte-wise left, cyclic rotate operation, and an XOR operation with a constant from rcon. This operation is shows line 8. The three other operations from lines 8-10 are XORs between the word from the prior line and the one of this line from the prior round. This extracts 10 round keys.

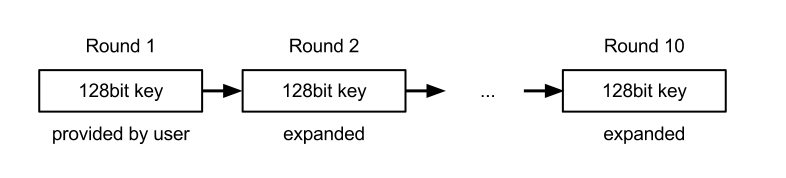


Figure 2. : The key expansion with a 128 bit key requires the construction of nine additional keys.

|  |
| --- |
| 1 KeyExpansion(byte Key[16], word W[40]) // key is provided and W is returned  2 W[0] = (Key[0], Key[1], Key[2], Key[3]); // 4 bytes  3 W[1] = (Key[4], Key[5], Key[6], Key[7]);  4 W[2] = (Key[8], Key[9], Key[10], Key[11]);  5 W[3] = (Key[12], Key[13], Key[14], Key[15]);  6 for (r = 1; r < 10; r++)  7 // RotBytes is a bit-wise cyclic leftward shift, SubBytes addresses the Sbox, and rcon are constants.  8 W[4\*r+0] = W[4\*(r-1)+0] ^ SubBytes[RotBytes[4\*r-1]] ^ rcon[r-1];  9 W[4\*r+1] = W[4\*(r-1)+1] ^ W[4\*r+0];  10 W[4\*r+2] = W[4\*(r-1)+2] ^ W[4\*r+1];  11 W[4\*r+3] = W[4\*(r-1)+3] ^ W[4\*r+2];  12 end |

Figure 2. : Pseudo-code for the key expansion with a 128 bit key.

The key expansion can be formulated designed as a linear feedback shift register with an initial load stage. Figure 2. 6 shows the schematics of such a register. The load\_enable line switches between the initial load phase for the first round key which provided by the user and keys which are generated in subsequent rounds. After the initialization R1 holds each 1st word, R2 each 2nd word, and R3 each 4th word. This corresponds to the code lines 2-5 in Figure 2. 5. Then code line 7 from Figure 2. 5 is computed by taking the value from R3 and applying the rot bytes and sub bytes transform. The result is XORed with the round-specific constant rcon and the value hold by R0 to compute the 5th word (line W0 in Figure 2. 6). For code line 8 from Figure 2. 5 we take the output result W0 and XOR it with the contents of R1 to compute the word W1. The words W2 and W3 are computed in the same way. All four words are fed back into the registers R0-R3 for the next round. Note that the longest path in this design contains 5XORs and one memory access to compute the sub bytes.

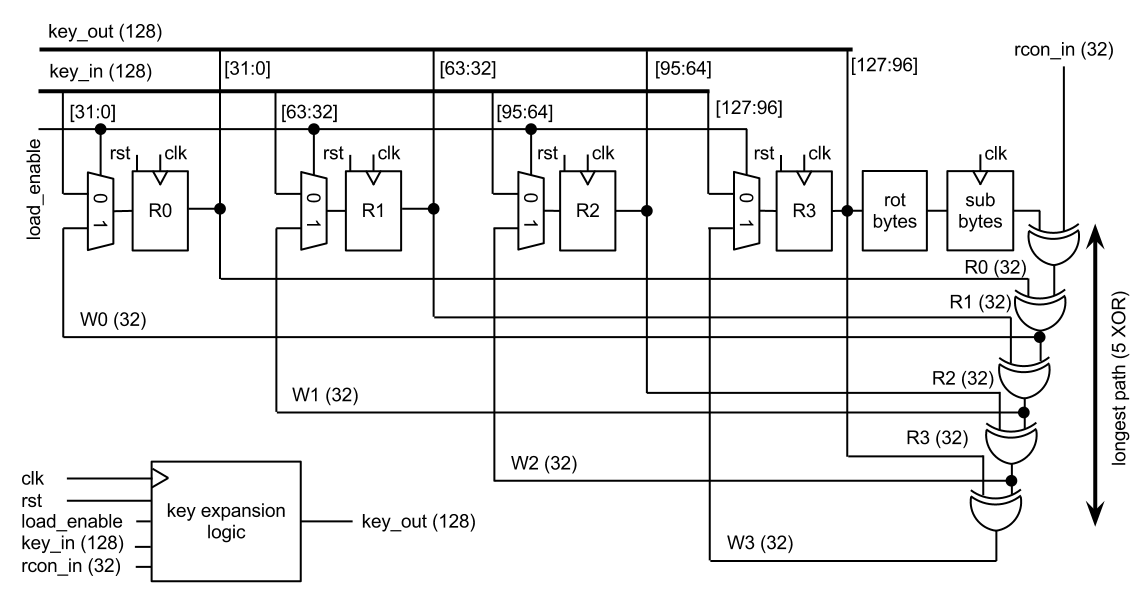


Figure 2. : The logic for the key expansion is formulated as a linear feedback shift register.

The control for this key expansion module is provided by a state machine. Figure 2. 7 shows the state machine. As long as no key\_in\_valid on the input side is provided the READY state is maintained. If key\_in\_valid asserts the key is latched into the internal registers R0-R1 and is also provided at the output side under the address key\_addr=1 while rcon\_addr=0 provides the correct address for the constant to be XORed in the 1st round. In the 1st round the loaded key is used to generate the key for the 2nd round. Address signals are set accordingly, see Figure 2. 7. After the 10th round the DONE state is reached. In this DONE state the key loaded signal assert, which indicates the completion of the key loading and expansion. An assertion of the reset signal rst always sets the FSM back to the READY state regardless of the current state.

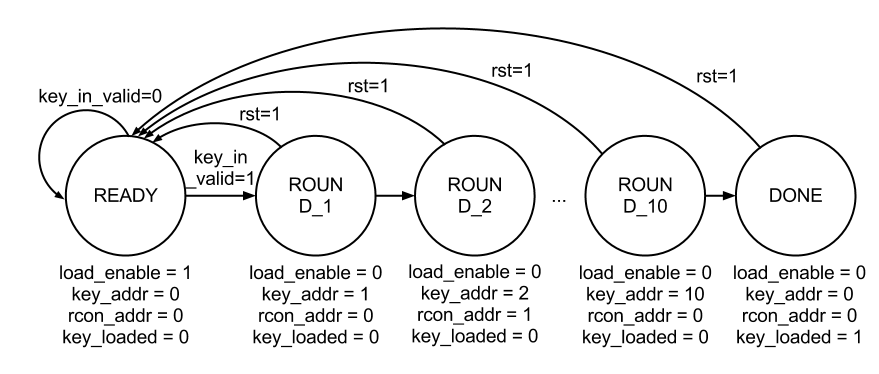


Figure 2. : The FSM for the key expansion control.

The key expansion logic and the control flow are embedded into the key expansion module, which Figure 2. 8 shows. A state registers holds the current state of the FSM. The control signals for the key address (key\_addr), the r-constant address (rcon\_addr), and the key loaded (key\_loaded) are generated based on this current state. The next-state-logic depends on the current state and the input signal key\_in\_valid which asserts if a valid input key is provided (at key\_in). The box rcon contains a memory with the ten constants that we use for the expansion.

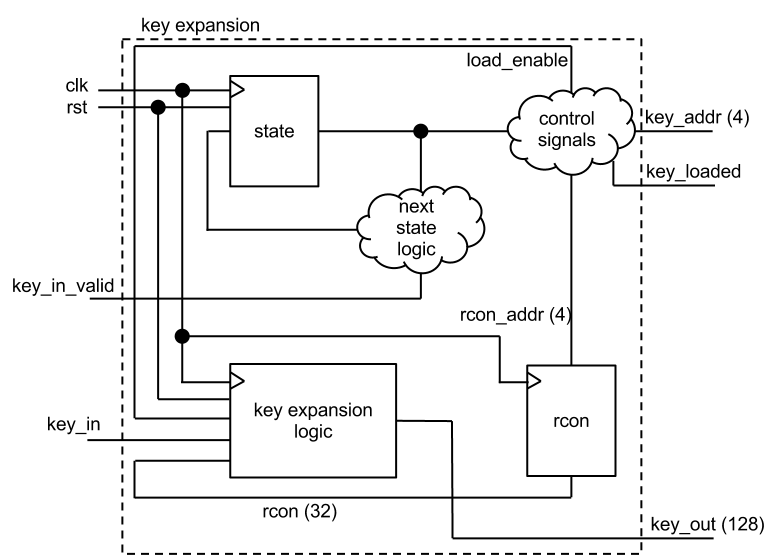


Figure 2. : The module for the key expansion.

## 2.4 Cipher Core

The Cipher Core module, shown is Figure 2. 9, contains all of the control and mode logic as well as instantiating the encryption and decryption pipelines. The Cipher Data In section handles feeding the pipelines with data and also holds encrypt and decrypt counters. The Cipher Data Out section handles registering and selecting the pipeline output data as well as calculating the final stages for the counter mode. The Cipher Controller section handles latching the keyed signal from the Key Expand module and the crypto mode from the external interface. The Cipher Controller determines when the device is ready to handle crypto data by asserting crypto ready signal to the external interface.

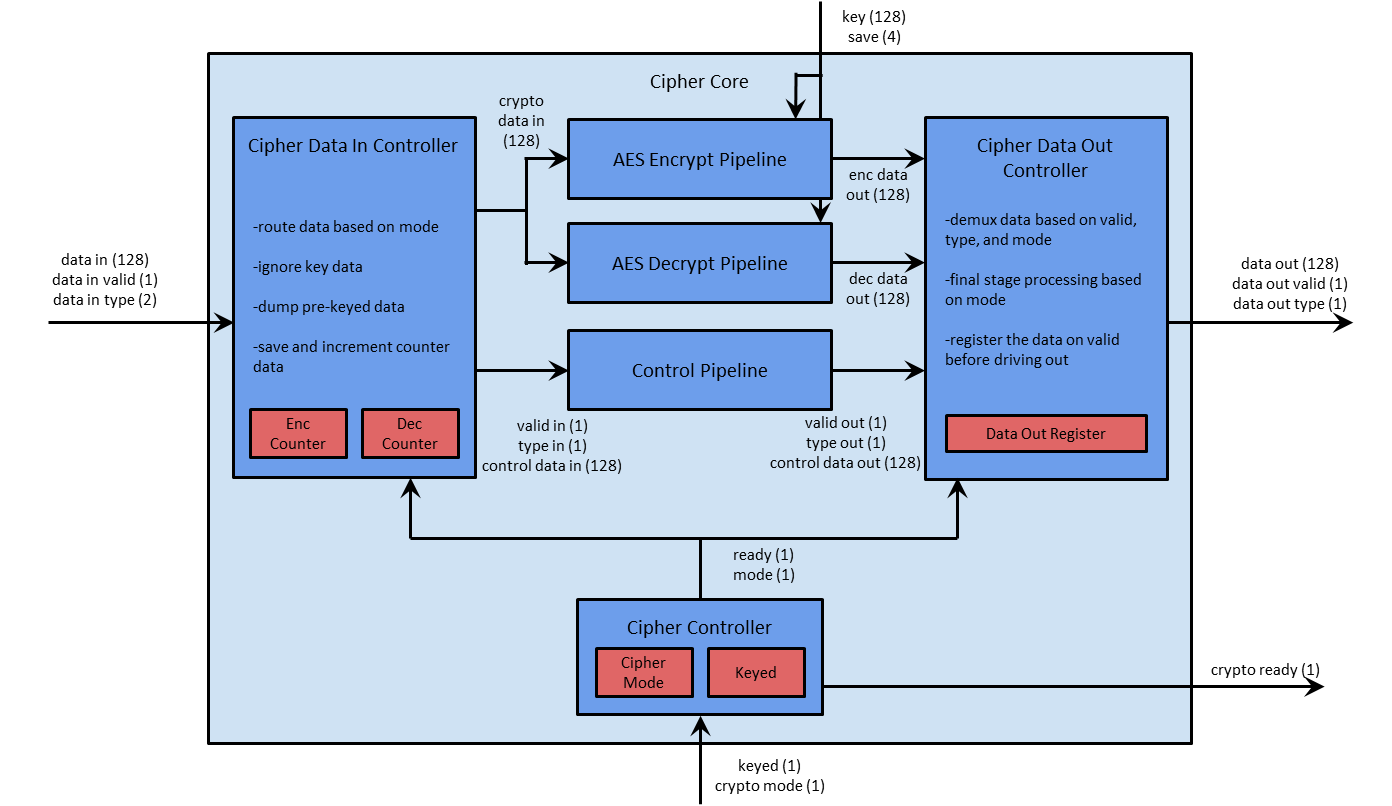


Figure 2. : Cipher Top Module

The first mode that the Cipher Core handles is AES ECB mode. When the Cipher Data In Controller sees valid data it will forward that data to both the Encrypt and Decrypt Pipelines and forward the valid and type signals to the Control Pipeline. The control data in the Control Pipeline will be driven with zeros. When the Cipher Data Out Controller sees a valid signal from the Control Pipeline it will check the type signal to determine which data, encrypt or decrypt, pipeline to drive the output data with. It also forwards the valid and type signals along with the output data.

The second mode that the Cipher Core handles is AES CTR mode. When the Cipher Data In Controller sees valid data it will forward that data to the control data in the Control Data Pipeline along with the type and valid signals. The Cipher Data In Controller will then drive both the Encrypt and Decrypt Pipelines with either the Encrypt Counter or Decrypt Counter depending on the data in type setting. It will also increment the counter used by one. When the Cipher Data Out Controller sees a valid signal from the Control Pipeline it will XOR the Control Pipeline data with the Encrypt Pipeline data and drive that data out. It also forwards the valid and type signals along with the output data.

### 2.4.1 AES Encrypt and Decrypt Pipelines

The AES Encrypt Pipeline and AES Decrypt Pipelines, shown in Figure 2. 10 and Figure 2. 11 respectively, are the primary calculating modules of this design. They are each made up of 10 rounds of the AES Encrypt or Decrypt Round modules chained back to back. The one caveat to this is the last round of the Encrypt Pipeline is not the same module as the others because it does not include the Mix Columns transformation. The Decrypt Pipeline has a similar caveat because the first round module does not include the Inverse Mix Columns transformation.

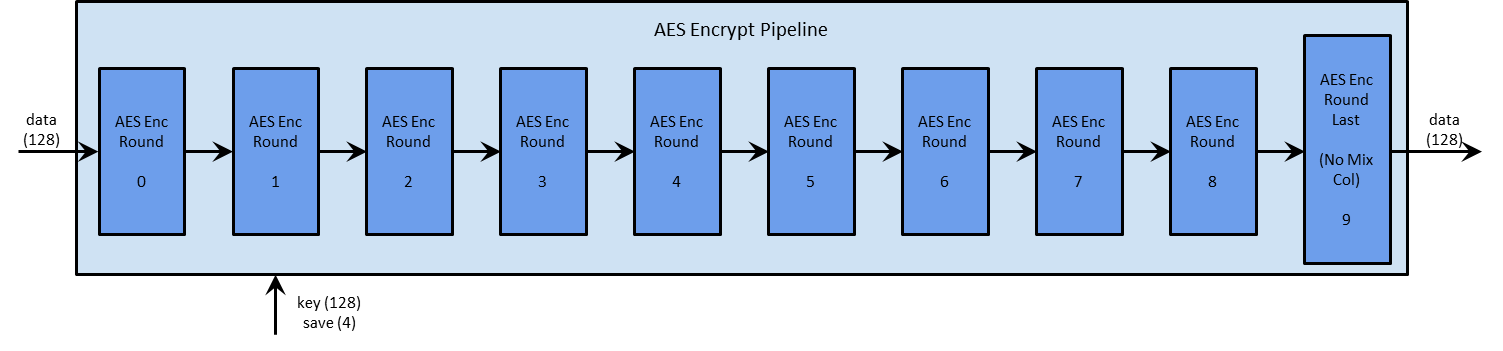


Figure 2. : AES Encrypt Pipeline Module

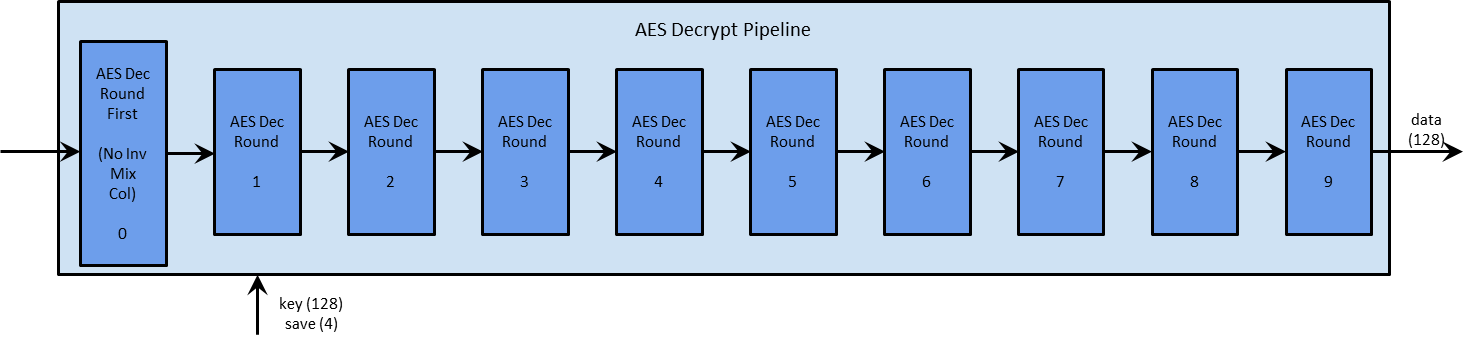


Figure 2. : AES Decrypt Pipeline Module

The interfaces to the pipelines are exactly the same. They have a 128 bit data in block and a 128 data out block as well as a 128 bit key data and key address lines. The data in is the data to be encrypted / decrypted and the data out is the data that was just encrypted / decrypted. The key and address buses is how the expanded key is distributed throughout the system. Each round will latch the key data when the address is that rounds number, allowing each round to hold its own partial key in local registers. Both encrypt and decrypt pipeline rounds hold the same key.

### 2.4.1 AES Encrypt and Decrypt Rounds

The AES Encrypt and Decrypt Rounds, shown in Figure 2. 12 and Figure 2. 13 respectively, are composed of four modules that are wired together by a 128 bit bus. The Encrypt Round is composed of the Substitute Bytes RAM, Shift Rows wiring, Mix Columns logic, and Add Round Key logic. The Decrypt Round is composed of the Inverse Substitute Bytes RAM, Inverse Shift Rows wiring, Inverse Mix Columns logic, and Add Round Key logic. Each operation and its inverse are very similar, except that they offset each other. For both the Encrypt and Decrypt Rounds, each clock will compute one full round of data calculation with the data driving from the output of a RAM to the address input of the next round.

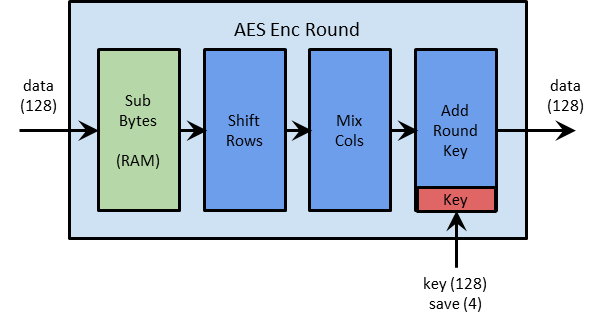


Figure 2. : AES Encrypt Round

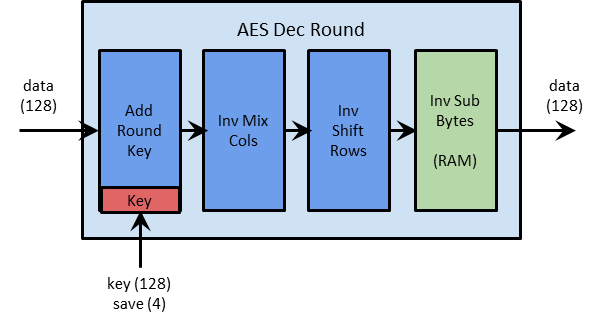


Figure 2. : AES Decrypt Round

The Sub Bytes and Inverse Sub Bytes, shown in general form in Figure 2. 14, are implemented as 16 byte addressable, byte output RAMs. Each byte of input data will drive one byte of output data. The Shift Rows and Inverse Shift Rows, shown in general form in Figure 2. 15, are implemented as simply a single re-routing of data lines with no actual logic. The Mix Columns and Inverse Mix Columns, shown in general form Figure 2. 16, are implemented as four word-wise transformations. For the Mix Columns the logic requires 3 XORs and 1 Mux as a longest path. For the Inverse Mix Columns the logic requires 5 XORs and 1 Mux as a longest path. The Add Round Key, shown in Figure 2. 17, is the same for both encryption and decryption and is simply a bit-wise XOR of the data and the local key schedule.

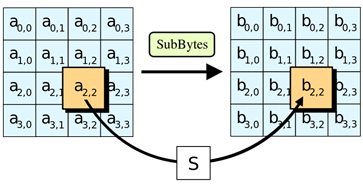


Figure 2. : Sub Bytes Operation

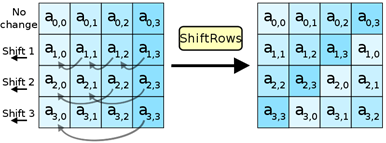


Figure 2. : Shift Rows Operation

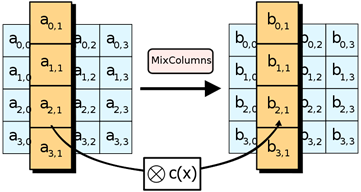


Figure 2. : Mix Columns Operation

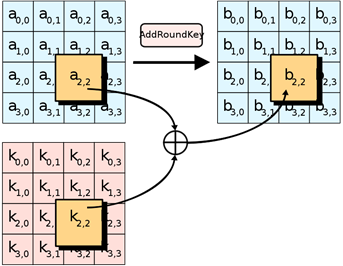


Figure 2. : Add Round Key Operation

## 2.5 Physical Design and Layout

TBD

# Design Verification

## Verification Design

Verification of our design will be used to prove the functionality of the AES encrypt/decrypt chip. For verification two types of testing will be done: Black-box testing and white-box testing. Black box testing is a form of testing that is used to test the functionality without any knowledge of the internal implementation. Black box testing has only knowledge of the primary inputs and outputs. White box testing is used to test the internal implementation of the hardware. A complete understanding of the architecture must be known to implement white box testing effectively.

The test design will implement two different architectures. Shown below in figure 3.1 is the architecture for using Known Answer Test suite of vectors. This uses known test vectors and their expected outputs to be used for verification.

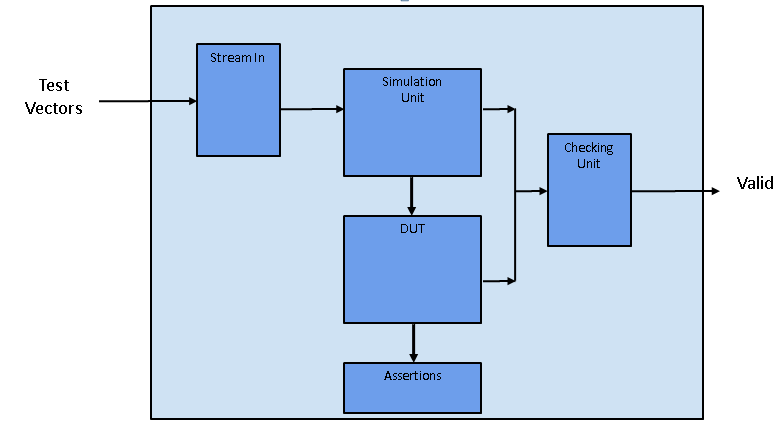


Figure 3. : Architecture of verification test bench using the Known Answer Test vectors.

To increase the coverage achieved in the verification step randomized testing will be implemented. Randomized testing uses a slightly different architecture than figure 3.1. To perform randomized testing a software model of the AES encryption will be used as the golden model to calculate the expected outputs. Figure 3.2 shows the architecture for randomized testing.

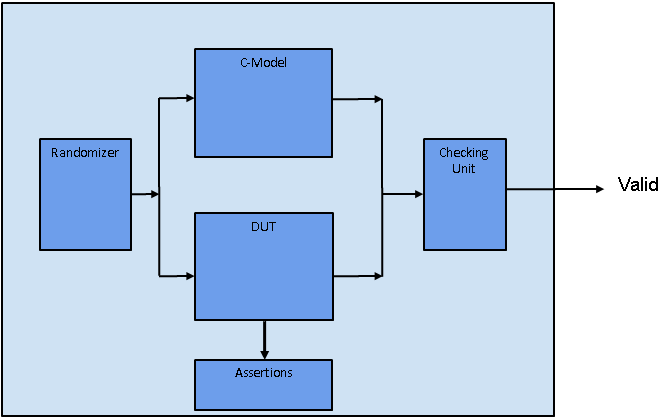


Figure 3. :Design for randomized testing. A software model is run in parallel with the DUT to compute the expected output.

## Pre-Synthesis Verification

### Black Box Testing

Black box testing will be implemented by constructing a test bench that feeds inputs to the simulation unit. In figure 3.1 the simulation unit is responsible for providing inputs to the Device Under Test, DUT, following the correct protocol. The simulation unit will then provide the expected ciphertext/plaintext to a checking unit that will compare the expected output to the actual output produced by the DUT. The DUT will be the hardware model of the AES chip. This will process the inputs given by the simulation unit and produce the actual ciphertext/plaintext. A Known Answer Test, KAT, provided by NIST provides a set of test vectors and outputs to use for verification. Along with the KAT, more test vectors can be used for corner cases and directed tests.

### White Box Testing

White box testing will be implemented by using assertions within the AES RTL model. Assertions will be used to test the inner functionality of components. Such assertions will be used to test the correct functionality of signal protocols, mutually exclusive signals, etc. All assertions will not be synthesizable and not be used in the synthesized net-list. The assertions will only be used during simulation for white box testing.

## Post-Synthesis Verification

Timing verification will be used for static timing analysis, STA, at the gate level. This verification step is performed after the design is synthesized and a net-list is created. Timing verification can be used to help maximally optimize the design clock as well as verify that there are no setup and hold violations in the pre-layout net-list.

TBD:

## 4.1 Labor

# 5. Conclusion

The conclusion may contain the following sections or others of your choosing.

## 5.1 Accomplishments

## 5.2 Uncertainties

## 5.3 Ethical considerations

## 5.4 Future work

# References

Daemen, J. & Rijmen, V. (1999). AES Proposal: Rijndael. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf (Accessed Feb 2013)

Rijmen, V., Bosselaers, A., & Barreto, P. (2000). C reference code of AES. http://www.efgh.com/software/rijndael.htm (Accessed Feb 2013)

Mali, M., Novak, F., Biasizoo, A. (2005). Hardware implementation of AES algorithm. Journal of Electrical Engineering 56(9-10), 265-269.

# Appendix A Requirement and Verification Table

TBD: Need to fill out.

An appendix is a good place for the Requirement and Verification Table from your design review. Below is a starter table. Including these details here will help to avoid lengthy and tedious narrative descriptions in the main text, which may not be of immediate interest to your imagined audience of company managers and professionals. Any requirement that is not verified should be explained either in the main text or the appendix. Note that both the pagination and the numbering of figures, tables, and equations continues from main text to appendices.

|  |  |  |
| --- | --- | --- |
| **Table X System Requirements and Verifications** | |  |
| Requirement | Verification | Verification status  (Y or N) |
| 1. Requirement    1. Subrequirement    2. Subrequirement    3. Subrequirement | 1. Verification    1. Subverification    2. Subverification    3. Subverification |  |
| 1. Requirement    1. Subrequirement    2. Subrequirement    3. Subrequirement | 1. Verification    1. Subverification    2. Subverification    3. Subverification |  |
|  |  |  |
|  |  |  |