Project Plan Report for EC772, Spring 2013

10 April 2013

Ben Humphries

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. We can run the chip with a clock of 500 MHz.

Contents

[1. Introduction 1](#_Toc353362860)

[2 Design 3](#_Toc353362861)

[2.1 Core Top 3](#_Toc353362862)

[2.2 Stream In and Stream Out 3](#_Toc353362863)

[2.3 Key Expansion 5](#_Toc353362864)

[2.4 Cipher Core 7](#_Toc353362865)

[2.4.1 AES Encrypt and Decrypt Pipelines 9](#_Toc353362866)

[2.4.1 AES Encrypt and Decrypt Rounds 9](#_Toc353362867)

[2.5 Physical Design and Layout 11](#_Toc353362868)

[3. Design Verification 15](#_Toc353362869)

[3.1. Verification Design 15](#_Toc353362870)

[3.2. Pre-Synthesis Verification 16](#_Toc353362871)

[3.2.1. Black Box Testing 16](#_Toc353362872)

[3.2.2. White Box Testing 18](#_Toc353362873)

[3.2.3. Visual Inspection Testing 19](#_Toc353362874)

[3.3. Post-Synthesis Verification 20](#_Toc353362875)

[4.1 Labor 21](#_Toc353362876)

[5. Conclusion 22](#_Toc353362877)

[5.1 Accomplishments 22](#_Toc353362878)

[5.2 Uncertainties 22](#_Toc353362879)

[5.3 Ethical considerations 22](#_Toc353362880)

[5.4 Future work 22](#_Toc353362881)

[References 23](#_Toc353362882)

[Appendix A Requirement and Verification Table 24](#_Toc353362883)

# 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, whereas 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 provided by the National Institute of Standards and Technology (Bassham, 2002).

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 (Rijmen et al., 2000). 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 in this design we chose low latency above costs for space, power, etc. This design supports AES in both ECB and CTR modes for additional security. Both ECB and CTR can be encrypted and decrypted fully in parallel. As a consequence both modes have the same latency. The major difference between the modes is the usage of the crypto pipelines and an extra operation after the encryption in case of CTR. 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. When using multiple keys, modes, or initialization vectors, the circuit must be reset when changing keys, modes, or vectors.

## 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 that 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.

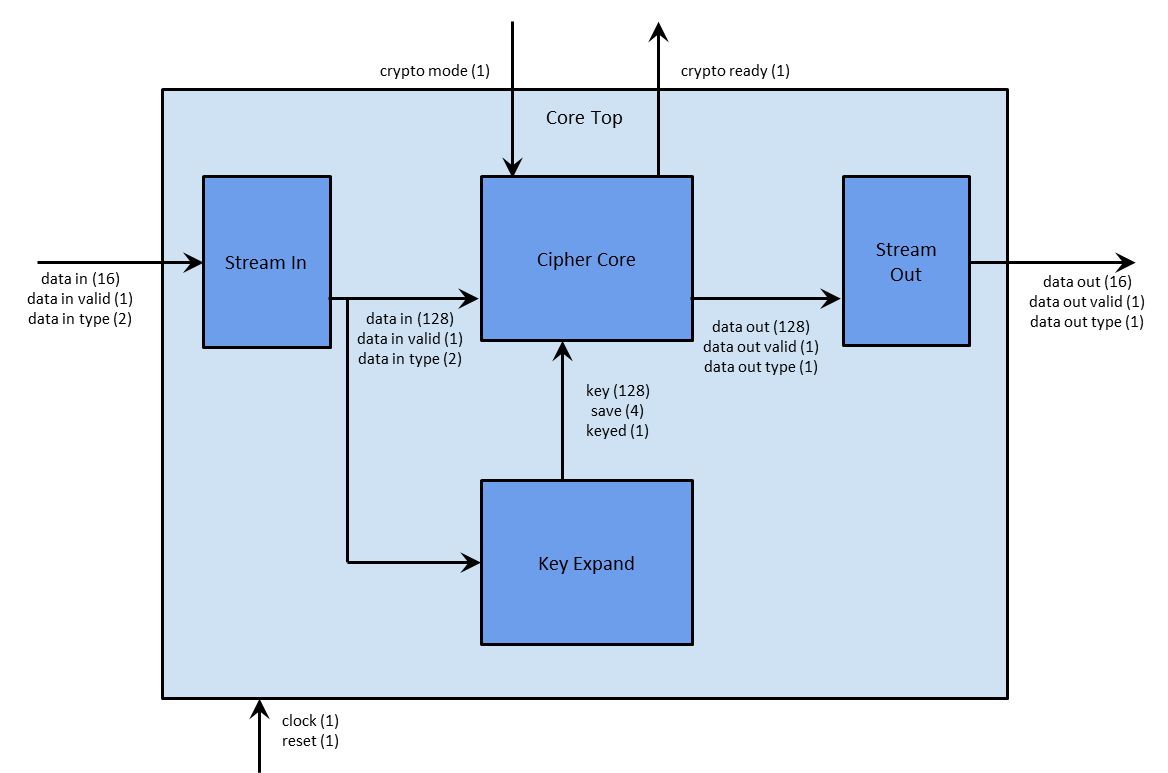


Figure 2. 1: Core Top Module

## 2.2 Stream In and Stream Out

The Stream In module, shown in Figure 2. 2, provides the interface for inputting plaintext, ciphertext, keys, and initialization vectors. The Stream Out, shown in Figure 2. 3, module provides the interface for getting plaintext and ciphertext to peripheral devices outside of our 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 eight 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.

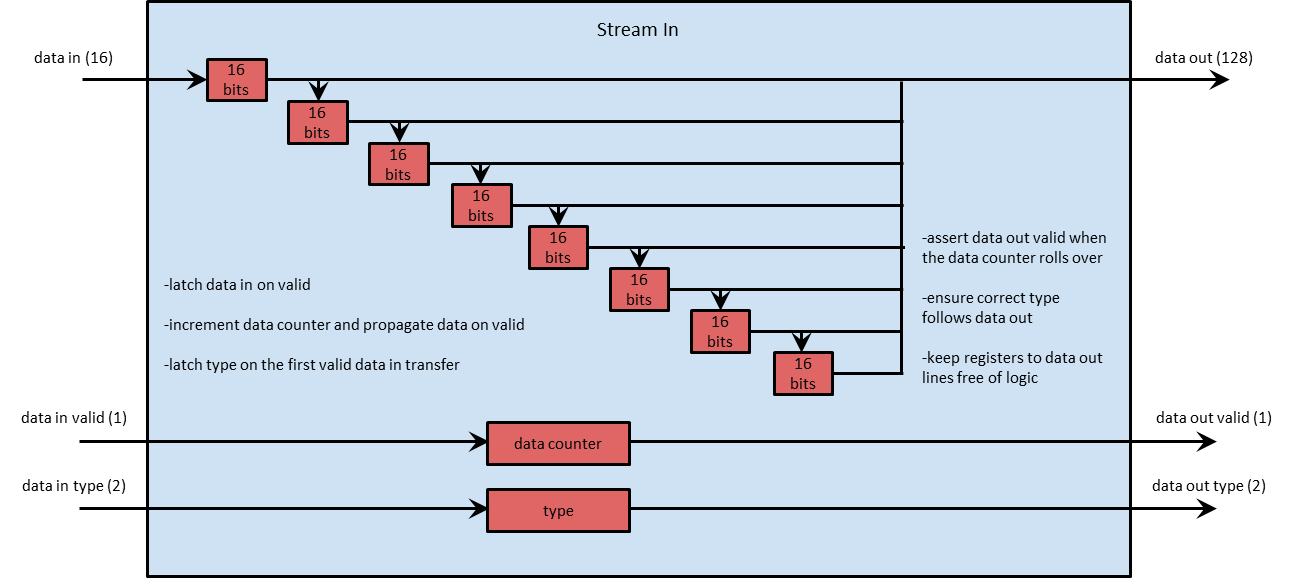


Figure 2. 2: Stream In Module

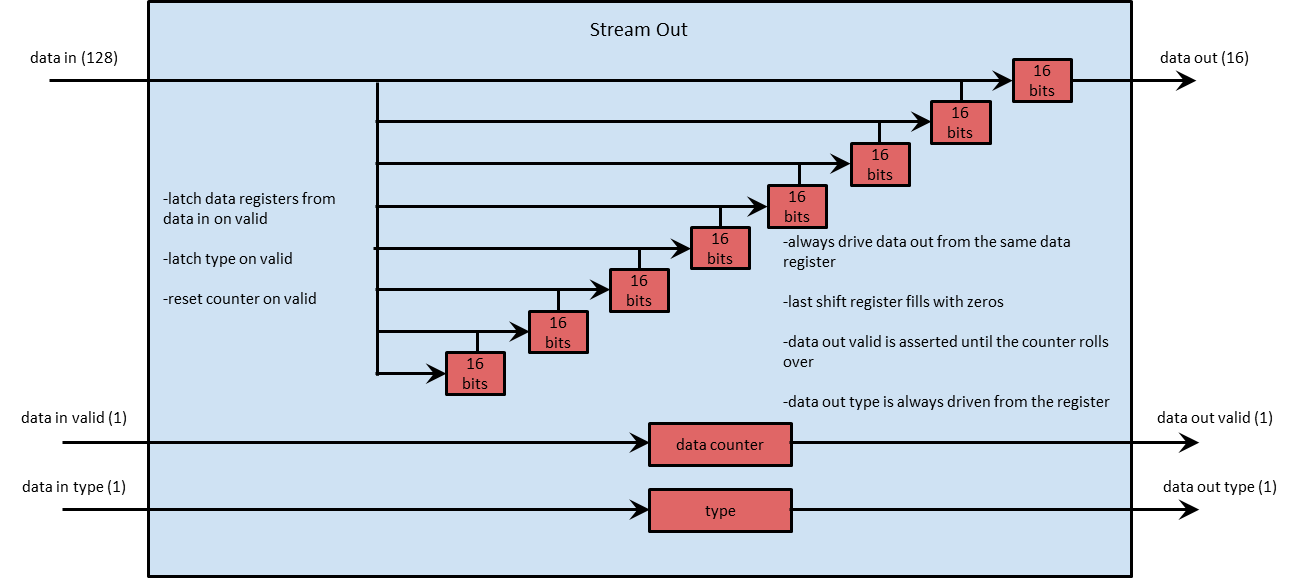


Figure 2. 3: Stream Out Module

Table 1: 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

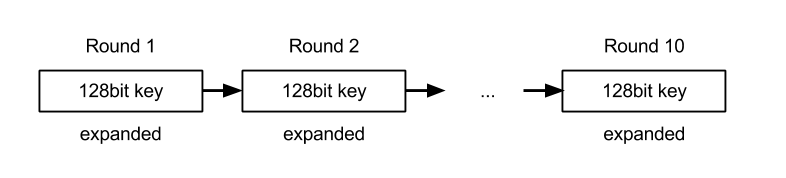
One 128 bit key is provided by the user; however, encrypt and decrypt operations use ten rounds each and each of these ten rounds uses a different key. This requires a schema for the expansion of the 128 bit key as indicated in Figure 2. 4. We show pseudo-code for this expansion operation in Figure 2. 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 ten 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 shown in 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 in addition to the initial key, which is provided by the user. In total 11 keys are used by the encryption and decryption.

Figure 2. 4: The key expansion with a 128 bit key requires the construction of ten 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. 5: Pseudo-code for the key expansion with a 128 bit key.

The key expansion logic can be 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 is provided by the user, and keys 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 implemented 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 held by R0. This computes the 5th word (line W0 in Figure 2. 6). For the implementation of the code line 8 in 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.

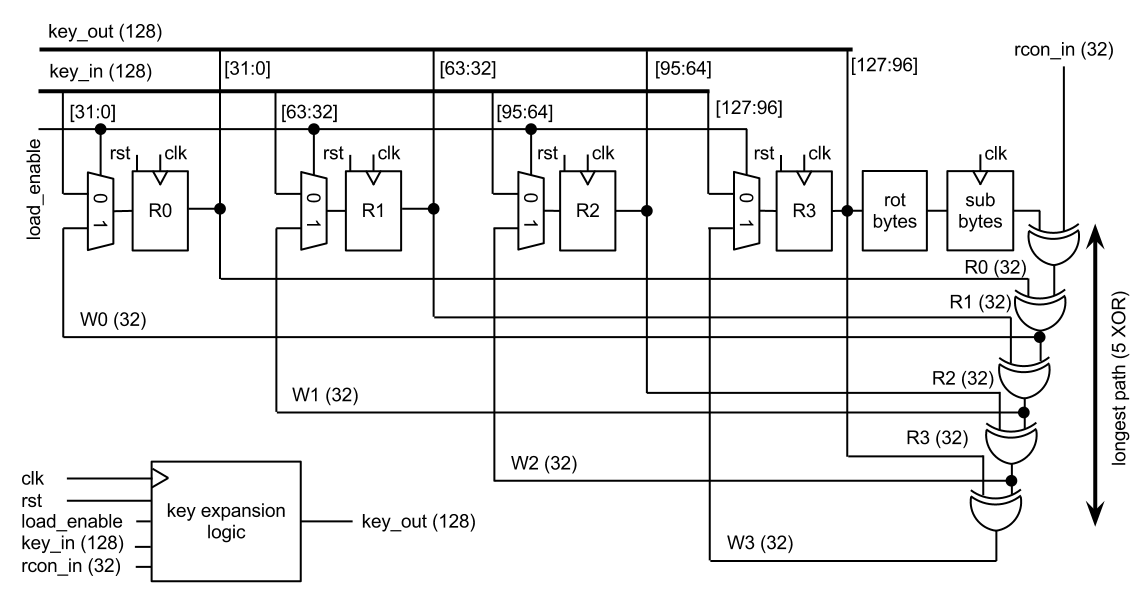


Figure 2. 6: The logic for the key expansion is formulated as a linear feedback shift register.

The control logic for this key expansion module is provided by a state machine shown in Figure 2. 7. The READY state is maintained as long as no key\_in\_valid on the input side is provided. If key\_in\_valid asserts the key is latched into the internal registers R0-R1 and is also provided at the output side with the address key\_addr=1 while rcon\_addr=0 provides the correct address for the constant to be XORed in the 1st round. Then, the loaded key is used to generate the key for the 1nd 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.

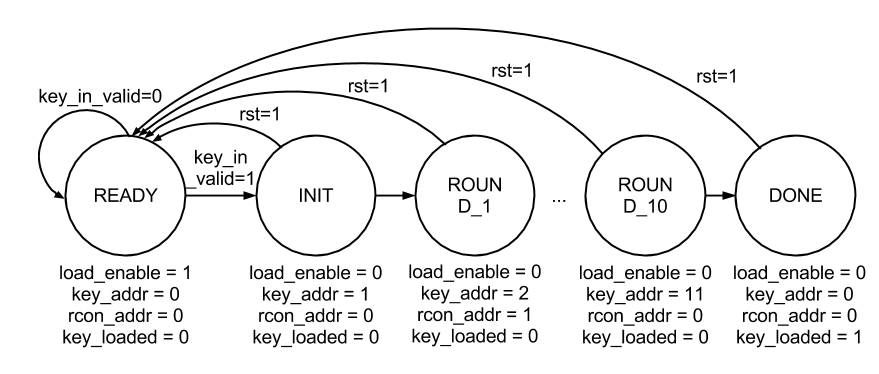


Figure 2. 7: The FSM for the key expansion control.

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 the 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.

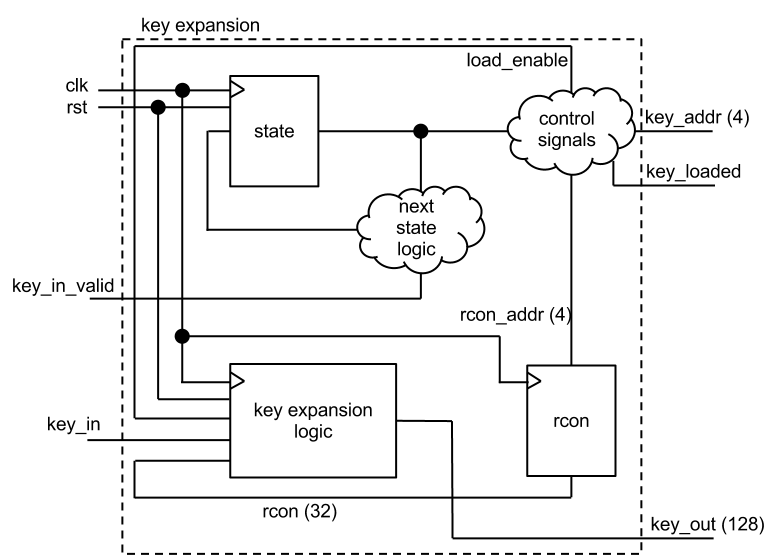


Figure 2. 8: The module for the key expansion.

## 2.4 Cipher Core

The Cipher Core module, shown in Figure 2. 9, contains all of the control and mode logic as well as instantiates the encryption and decryption pipelines. The Cipher Data In section handles feeding the pipelines with data and holds encrypt and decrypt counters. The Cipher Data Out section handles registering and selecting the pipeline output data and calculates 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.

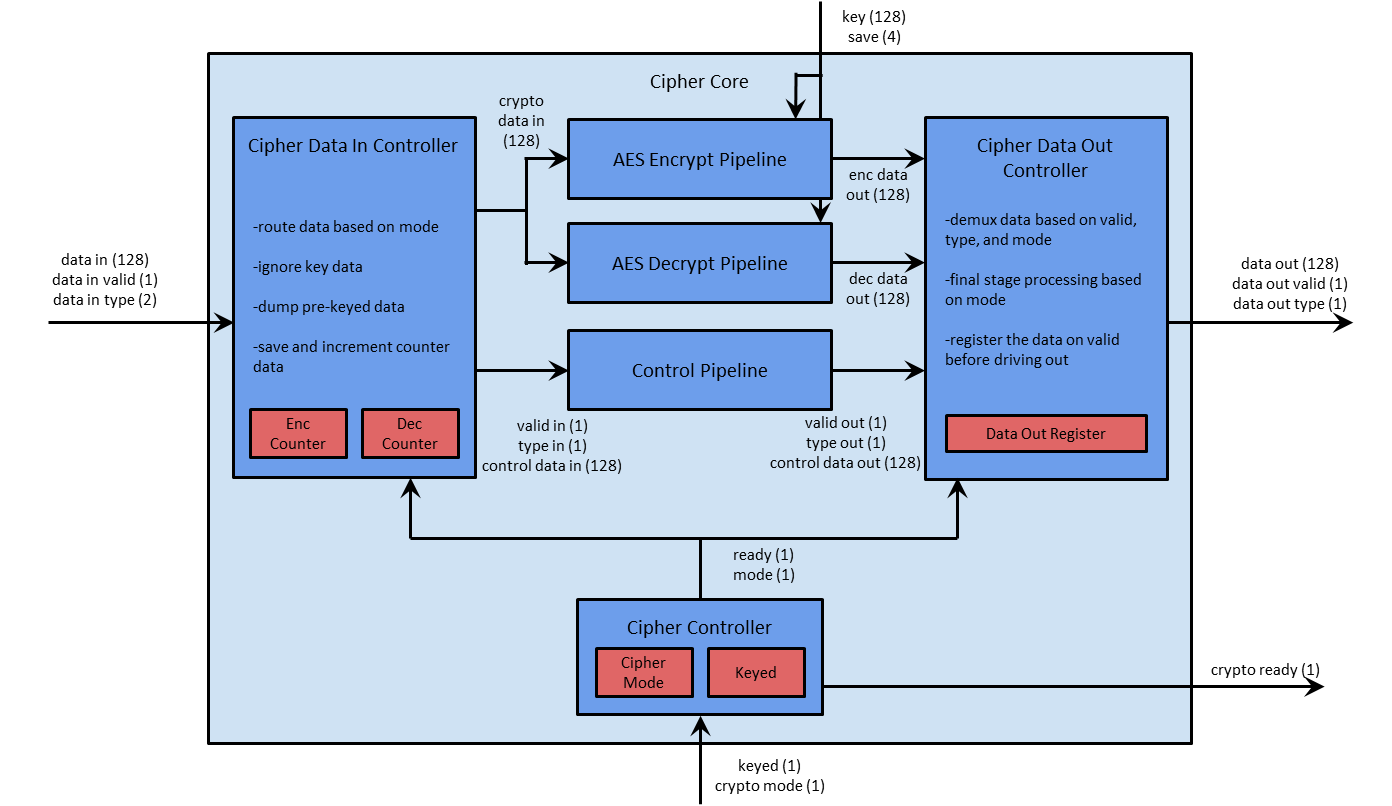


Figure 2. 9: Cipher Top Module

The Cipher Core supports the AES ECB mode. When the Cipher Data In Controller sees valid data, it will forward the valid 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 AES CTR mode is another mode that the Cipher Core supports. 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 a counter 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 sends the result of the XOR operation out. Valid and type signals are send together 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 main modules for encryption and decryption. Each contains 10 rounds of the AES Encrypt or Decrypt Round modules chained back to back. Only 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 exception because the first round module does not include the Inverse Mix Columns transformation.

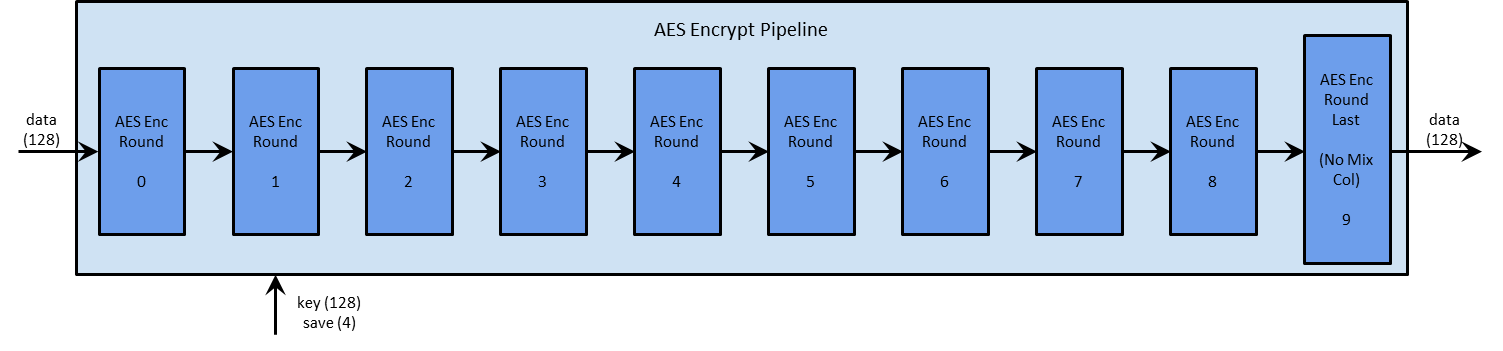


Figure 2. 10: AES Encrypt Pipeline Module

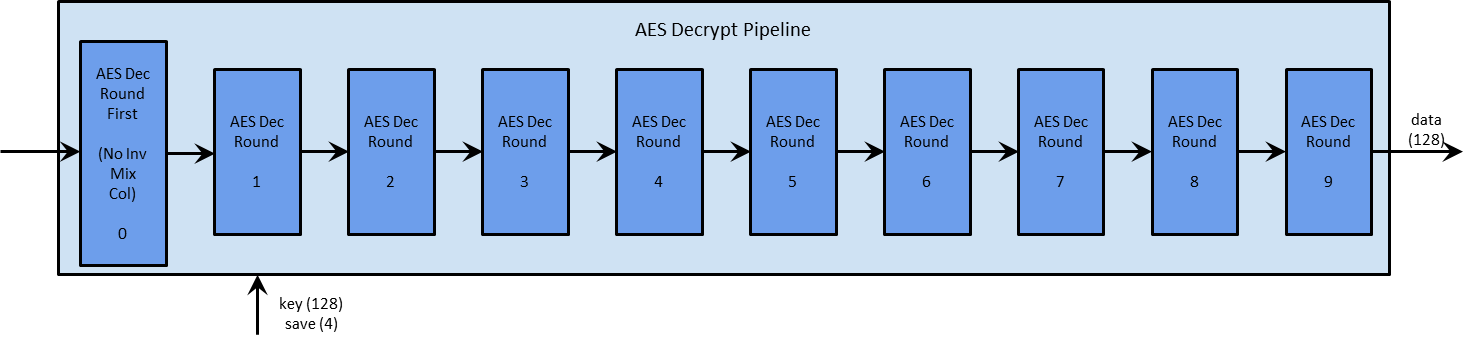


Figure 2. 11: AES Decrypt Pipeline Module

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. Key and address buses are used to distribute the expanded key. In each round the expanded key indicated by the address is, which is the round number, is latched. This allows each round to hold its own expanded 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 connected 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.

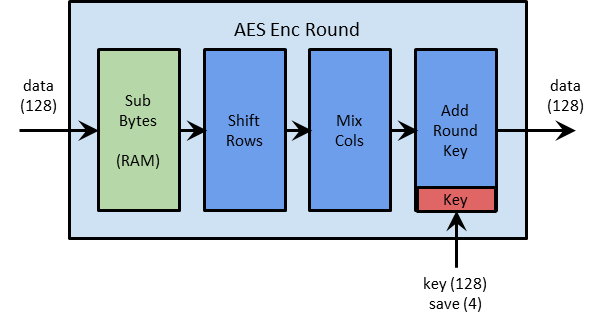


Figure 2.12: AES Encrypt Round

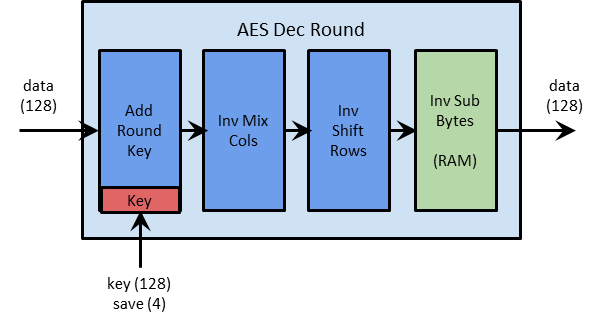


Figure 2.13: AES Decrypt Round

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.

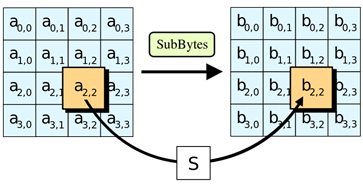


Figure 2.14: Sub Bytes Operation

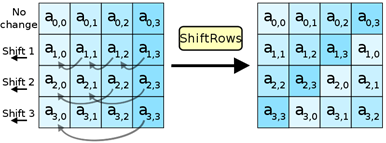


Figure 2.15: Shift Rows Operation

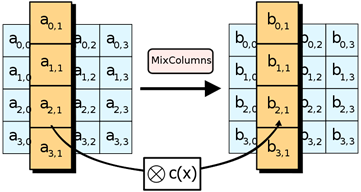


Figure 2.16: Mix Columns Operation

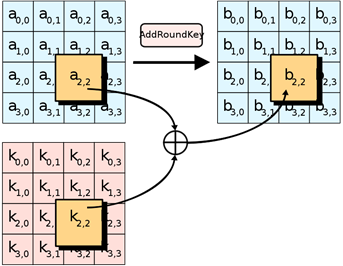


Figure 2.17: Add Round Key Operation

## 2.5 Physical Design and Layout

For the synthesis of our design, we use the cadence encounter (CE) tool. This tool takes the Verilog description and a standard cell library, in our case the Nangate library, and compiles a netlist of basic gates and registers.

|  |
| --- |
| Warning : Ignoring unsynthesizable construct. [VLOGPT-37]  : Initial in file '../rtl/inv\_sbox.v' on line 22, column 10.  : The following constructs will be ignored:  - initial block  - final block  - program block  - property block  - sequence block  - covergroup  - gate drive strength  - system task enable  - reg declaration with initial value  - specify block.  initial $readmemh("inv\_sbox.dat", inv\_sbox);  |  Warning : Ignoring unsynthesizable construct. [VLOGPT-37]  : Call to system task '$readmemh' in file '../rtl/inv\_sbox.v' on line 22, column 21.  initial $readmemh("rcon.dat", rcon); |

Figure 2.18: Warning of cadence encounter about unsynthesizable construct.

Our design contained the “initial readmemh(“file.dat”,data);” construct, which did not synthesize. Figure 2.18 shows the warning of the CE tool. This warning also lists other unsynthesizable constructs, such as initial blocks, program blocks, or the initialization of a reg with a value. Initially, we thought that the readmemh command would be synthesized into a RAM; however, this is not the case and it the Nangate library did not specifically provide a RAM. We re-wrote the code to be synthesizable, buffering the address for a multiplexer whose inputs are directly driven by power and ground. Figure 2.19 shows an example of the unsynthesizable code side by side with the synthesizable construct.

|  |  |
| --- | --- |
| Unsynthesizeable construct | Synthesizable construct (rcon.v) |
| module rcon (  input clk,  input [3:0] addr,  output reg [31:0] dout);  reg[31:0] rcon [0:15];  initial begin  $readmemh(“rcon.dat”, rcon)  end  always @(posedge clk)  dout <= rcon[addr];  endmodule  From rcon.dat  01000000  02000000  … | module rcon(  input clk,  input rst,  input [3:0] addr,  output reg [31:0] dout);  reg [3:0] addr\_r;  always @(posedge clk)  if (rst) addr\_r <= 0;  else addr\_r <= addr;  always @(\*)  case (addr\_r)  4'h0: dout = 32'h01000000;  4'h1: dout = 32'h02000000;  4'h2: dout = 32'h04000000;  …  endcase  endmodule |

Figure 2. : Unsynthesizeable and synthesizable construct for a “memory”.

The CE tool provides delay and area information, based on the wiring and standard cells from the Nangate library. Figure 2.20 shows the area information for our design. Most of the area (81%) is used for logic. The area for the chip is specified in µm² and, thus, is 0.24mm². For a squared design the design is of area 0.48mm × 0.48mm.

|  |
| --- |
| Type Instances Area Area %  ---------------------------------------  sequential 5971 27353.844 11.4  inverter 34765 18495.246 7.7  buffer 8 6.384 0.0  logic 157272 195000.344 81.0  ---------------------------------------  total 198016 240855.818 100.0 |

Figure 2.20: Area for layout of design using cadence encounter with the Nangate 45nm technology.

Figure 2.21 shows the timing report. As constraint for the synthesis we specified a clock period of 2,000ps or 2ns, which is a clock frequency of 500 MHz. The critical path for timing occurs in the cipher core for the decoding pipeline between the clock of the data\_in address in round eight for the inverse sbox and the data out signal of the inverse sbox in round nine with a time of 1,642ps, which leaves a slack of 358ps. Thus, we could choose a clock faster than the 500 MHz.

|  |
| --- |
| ============================================================  Generated by: Encounter(R) RTL Compiler v09.10-s242\_1  Generated on: Apr 07 2013 10:00:47 PM  Module: core\_top  Technology library: NangateOpenCellLibrary revision 1.0  Operating conditions: typical (balanced\_tree)  Wireload mode: enclosed  Area mode: timing library  ============================================================  Pin Type Fanout Load Slew Delay Arrival  (fF) (ps) (ps) (ps)  ------------------------------------------------------------------------  (clock clk\_input) launch 0 R  u\_cipher\_core  u\_dec\_pipeline  u\_round\_8  u\_inv\_sub\_bytes  u\_inv\_sbox\_5  din\_r\_reg[3]/CK 0 0 R  din\_r\_reg[3]/QN DFF\_X1 16 34.3 82 +144 144 R  g14057/A2 +26 169  g14057/ZN NOR2\_X1 10 19.6 37 +57 226 F  g14056/A +20 246  g14056/ZN INV\_X1 6 12.7 33 +54 300 R  g13962/A1 +18 318  g13962/ZN NOR2\_X1 2 3.6 26 +16 334 F  g13773/B1 +14 348  g13773/ZN OAI21\_X1 1 2.0 21 +36 383 R  g13772/A +12 395  g13772/ZN INV\_X1 1 1.8 7 +10 405 F  g13676/A +12 416  g13676/ZN AOI221\_X1 1 1.9 43 +71 488 R  g13611/A +11 499  g13611/ZN OAI211\_X1 1 1.8 20 +36 535 F  g13600/A +12 547  g13600/ZN AOI221\_X1 1 1.9 43 +76 623 R  g13575/A2 +11 634  g13575/ZN NAND4\_X1 1 1.6 20 +36 671 F  g13572/A1 +11 682  g13572/ZN NOR4\_X1 1 2.5 50 +58 740 R  u\_inv\_sbox\_5/dout[5]  u\_inv\_sub\_bytes/dout[45]  u\_add\_rkey/din[109]  g2144/A +16 756  g2144/Z XOR2\_X1 9 25.1 130 +172 927 R  u\_add\_rkey/dout[109]  u\_inv\_mix\_cols/din[109]  u\_inv\_mix\_bytes\_0/i2[5]  g623/B +29 957  g623/Z XOR2\_X1 5 15.0 84 +125 1082 R  g614/A +22 1104  g614/Z XOR2\_X1 4 12.3 71 +109 1213 R  u\_gf\_e/i2[7]  g94/B +23 1236  g94/Z XOR2\_X1 1 4.0 33 +67 1303 R  u\_gf\_e/o2[4]  g3/A +25 1328  g3/S FA\_X1 1 3.9 17 +99 1427 F  g1742/A +25 1452  g1742/S FA\_X1 1 1.2 11 +113 1565 R  u\_inv\_mix\_bytes\_0/o2[4]  u\_inv\_mix\_cols/dout[108]  u\_round\_8/dout[108]  u\_round\_9/din[108]  u\_inv\_sub\_bytes/din[108]  u\_inv\_sbox\_d/din[4]  g14113/A1 +8 1573  g14113/ZN AND2\_X1 1 1.4 8 +30 1602 R  din\_r\_reg[4]/D DFF\_X1 +9 1611  din\_r\_reg[4]/CK setup 0 +31 1642 R  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  (clock clk\_input) capture 2000 R  ------------------------------------------------------------------------  Timing slack : 358ps  Start-point : u\_cipher\_core/u\_dec\_pipeline/u\_round\_8/u\_inv\_sub\_bytes/u\_inv\_sbox\_5/din\_r\_reg[3]/CK  End-point : u\_cipher\_core/u\_dec\_pipeline/u\_round\_9/u\_inv\_sub\_bytes/u\_inv\_sbox\_d/din\_r\_reg[4]/D |

Figure 2. : Timing report of synthesis using cadence encounter with the Nangate 45nm technology.

# Design Verification

## Verification Design

Verification of our design will be used to prove the functionality of the AES encrypt/decrypt chip. For verification we use two types of testing: Black-box testing and white-box testing. Black box testing tests the functionality without knowledge of the internal implementation. Black box testing only knows the primary inputs and outputs of the device under test (DUT). White box testing is used to test the internal implementation of the hardware. White box testing requires a complete understanding of the architecture for an effective test implementation.

The test design will implement two different architectures. Shown below in Figure 3.1 is the architecture for the Known Answer Test suite. This suite uses known test vectors and their expected outputs from NIST (Bassham, 2002) to verify the design.

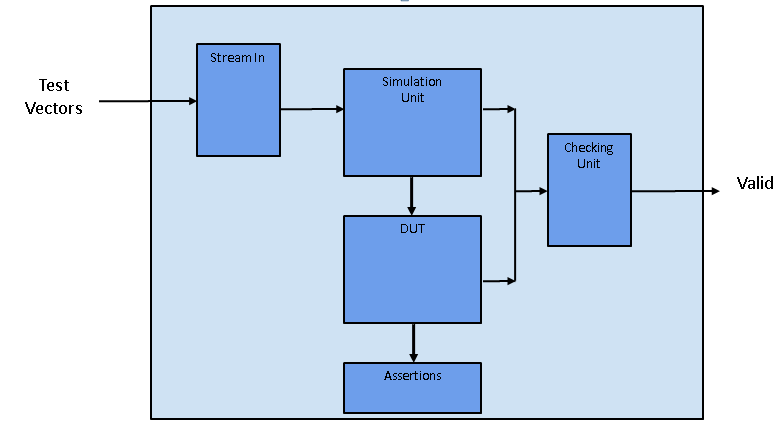


Figure 3.1: Architecture for verification test bench using the Known Answer Test vectors from NIST.

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

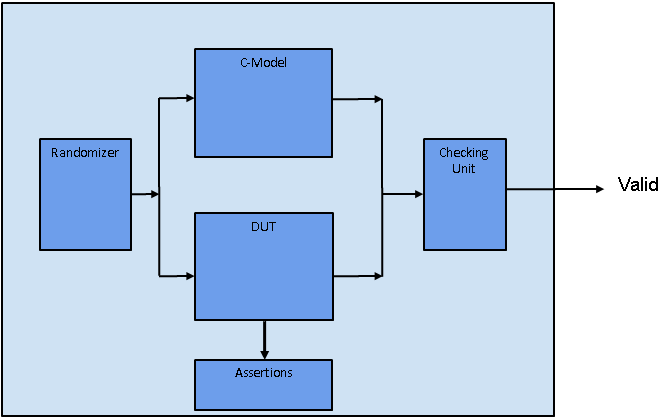


Figure 3.2: Design for randomized testing. A software model is run in parallel with the DUT to compute the expected output.

## 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 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 (Bassham, 2002) 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.

Before the entire chip can be verified using the above black-box design, individual module units will be tested to ensure proper functionality. Although this may seem like a form of white-box testing since testing will be done on modules inside the design and not at the chips boundaries, this is still a form of black-box testing done at a different level of abstraction. The testing of some major hardware modules will test the inputs and outputs for correct functionality. If bugs are encountered in this stage a form of white-box testing must be implemented on the corresponding module to investigate the cause of the bug and to provide a fix.

The two figures, Figure 3.2.1 and 3.2.2 below show a test vector being entered into the chip. The test vector is a vector from the Known Answer Test, KAT. From Figure 3.2.1 the reset signal is initially high. To bring the system out of reset and enable interaction with the system the reset signal is de-asserted. After some time the three signals assert: *data\_in\_type*, *data\_in*, and *data\_in\_valid*. The *data\_in\_type* signal asserts to 0x2, which signals to the system that the data on the input is the key.

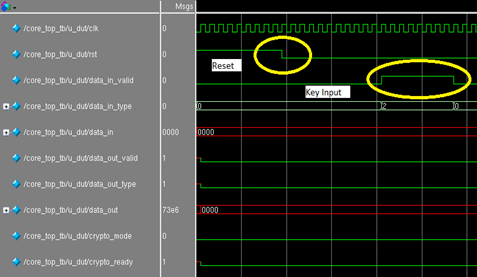


Figure 3.2.1: A test vector is entered into the system after reset. Here the system is brought out of reset and a key is entered. In this case the key is 0x0.

Figure 3.2.2 shows the next part in the test. After the key is entered into the system, the system will expand the key and assert the *crypto\_ready* signal to show to that it is ready for data. When this signal is asserted the *data\_in\_type* signal is set to the appropriate value, 0x0, to represent data, and data is put on the *data\_in* lines. After the data is entered into the system the AES chip will compute the output and a few clocks later the *data\_out\_signal* will be asserted and the output will be sent through the *data\_out* lines.

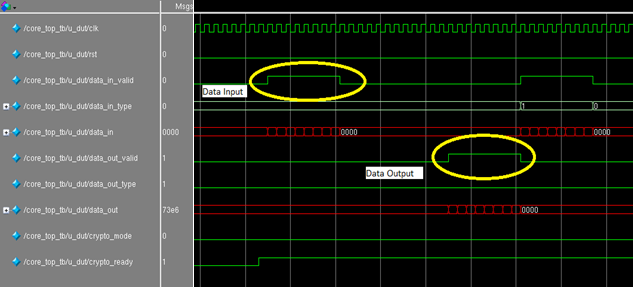


Figure 3.2.2: A test vector is entered into the system after reset. The *data\_in\_valid* signal is asserted and the data is entered into the system. The data output is shown a few clocks later.

Figure 3.2.1 and 3.2.2 show the AES chip performing the AES encryption with ECB mode for a test vector from the KAT suite. The KAT suite contains 156 test vectors for 128-bit AES encryption with ECB. Each test follows the same format described above. For each test vector the reset is asserted then de-asserted to reset the system back to a known state. This procedure is also required when entering a new key. For larger chunks of data with the same key the system does not need to perform a reset for each 128-bit data chunk, as long as the key is the same.

### 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.

The two figures below, Figure 3.2.3 and 3.2.4, show the implementation of the chip in the encryption pipeline. The figures show the data traveling through the encryption pipeline from the data input to the data output. In Figure 3.2.3 the chip is brought out of reset and a key is entered into the system similar to the procedure shown in Figure 3.2.1.

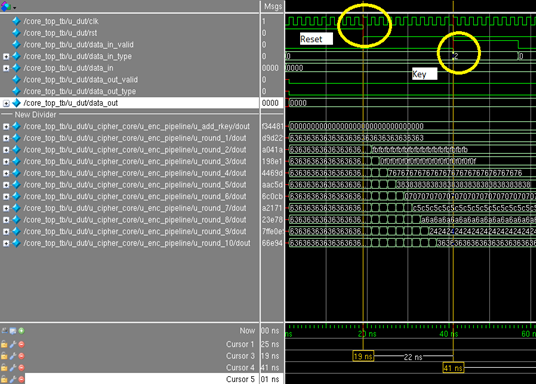


Figure 3.2.3: This figure shows the implementation of the encryption pipeline. The system is first brought of reset and a key is entered shown here.

Figure 3.2.4 shows the data being entered into the system on the *data\_in* signal. The signal *dout* from *u\_add\_rkey*, *and u\_round\_1* through *u\_round\_10* show the data traveling through the encryption pipeline. After the data reaches the last round it is sent to the output and the *data\_out\_valid* signal is asserted.

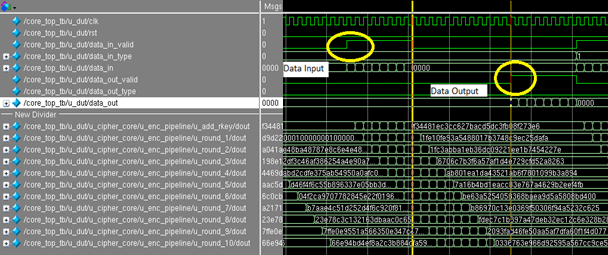


Figure 3.2.4: This figure shows the implementation of the encryption pipeline. Data is entered into the system and each round shows the intermediate data after it has been computed for that round. After 12 clock cycles the data is sent to the output.

### Visual Inspection Testing

Part of the motivation of using both ECB and CTR modes was to provide more secure communication patterns by ensuring dispersion of the data. This property is easily detected when encrypting an image of data, such as Figure 3.3.1. An example from Wikipedia shows the image when run through encryption in both ECB and CTR modes, as seen in Figures 3.3.2 and 3.3.3 respectively. Another way of proving that the design is correct, it should be able to pass a visual inspection test of encrypting an image in both ECB and CTR that look similar to the ones found in Wikipedia. Figures 3.3.4 and 3.3.5 show the original image when encrypted using the design for ECB and CTR modes respectively. Clearly the design’s ecrypted images are similar enough to the ones from Wikipedia that our design would pass a visual inspection test. There are some differences in the ECB versions, which is due to how the pixels were extracted and packed into raw data before encryption. There are 3 bytes of data per pixel, and in the Wikipedia version, this data was simply packed back to back for encryption and any odd bytes were handled at the end of the image. This is why there are diagonal lines running through the background of the picture. The design’s ECB version has straight lines in the background of the picture. This is because each three byte pixel of raw data was zero-padded at the end in order to make the scripting of converting from an image to raw data easier.

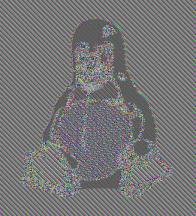
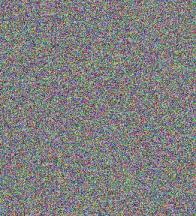
 

Figure 3.3.1: Original input image. Figure 3.3.2: Wikipedia ECB image. Figure 3.3.3: Wikipedia CTR image.

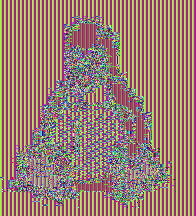
 

Figure 3.3.4: This design’s ECB image. Figure 3.3.5: This design’s CTR image.

## 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.

## 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.

Bassham, L.E. (2002). The advanced encryption standard algorithm validation suite (AESAVS). National Institute of Standards and Technology. http://csrc.nist.gov/groups/STM/cavp/ (Accessed March 2013)

Wikipedia (2013). Block cipher modes of operation. http://en.wikipedia.org/wiki/Block\_cipher\_modes\_of\_operation (Accessed April 2013)

# Appendix A Requirement and Verification Table

The table below shows the requirements for the AES encrypt/decrypt chip. Requirements that the chip should perform are shown in the left column, verification methods are shown in the middle column, and if the AES chip has passed the corresponding requirement is shown in the right column.

|  |  |  |
| --- | --- | --- |
| **Table X System Requirements and Verifications** | |  |
| Requirement | Verification | Verification status  (Y or N) |
| 1. The AES core shall use the data\_in input for both the key and the data. |  | N |
| 1. The AES core shall input the 128-bit key first, before any data input    1. Any data input before a valid key is provided will not be processed by the chip. |  | N |
| 1. The AES core shall provide handshaking with the data\_in\_valid input    1. The data\_in\_valid input will tell the AES core the data is valid on the data\_in input. |  | N |
| 1. The AES core shall only process data after the crypto\_ready output is asserted. |  | N |
| 1. The AES core shall process the data on the data\_in input with a key that was provided after the last reset. |  | N |
| 1. The AES core shall use a new key only after a global reset has been asserted. |  | N |
| 1. More TBD |  |  |