ENGG4560 Laboratory Tutorial: Asynchronous AES SoC Coprocessor Design

By Radu Muresan, May 2022

Introduction

The references for this implementation are:

[1] NIST, AES_NIST.FIPS.197, November 2001; posted in the Appendix.

[2] M. Mayhew, ASIC Implementation of an AES co-processor, MASc Thesis, 2009, University of Guelph (pp. 46 – 67; posted on ENGG*4560 CourseLink.

[3] J. Wolkerstorfer, E. Oswald, M. Lamberger (2002). An ASIC Implementation of the AES SBoxes. In: Preneel, B. (eds) Topics in Cryptology — CT-RSA 2002. CT-RSA 2002. Lecture Notes in Computer Science, vol 2271. Springer, Berlin, Heidelberg. https://doi-org.subzero.lib.uoguelph.ca/10.1007/3-540-45760-7_6; posted in the Appendix.

[4] Hua Li and Z. Friggstad, "An efficient architecture for the AES mix columns operation," 2005 IEEE International Symposium on Circuits and Systems (ISCAS), 2005, pp. 4637-4640 Vol. 5, doi: 10.1109/ISCAS.2005.1465666; posted in the Appendix.

[5] L. R. Knudsen, M. J. B. Robshaw, The Block Cipher Companion, Springer, 2011.

[6] My project path: C:\engg4560_lab\my_aes_custom_ip\my_first_hps-fpga_base\soc_system.qpt (the working Verilog file is: my_aes128_ip.v); Verilog code posted in the Appendix.

This SoC design implements in Verilog the 128-bits AES encryption algorithm along with the 128-bits cipher key expansion targeting the DE1-SoC development board that we utilize for our ENGG*4560 laboratory. The implementation followed the AES-128 encryption algorithm described in [1] and was verified for all rounds using the numerical examples in [1]. The Verilog implementation of the AES is an asynchronous type implementation, entirely combinational circuitry; no clock is used within the AES module. As the inputs (plaintext and the secret key) to the AES module are changed, the encryption result will be generated to the ciphertext output after the propagation delay through the combinational gates from the start round to the final round.

Integrating the my_aes_128_ip Coprocessor Design into the Project

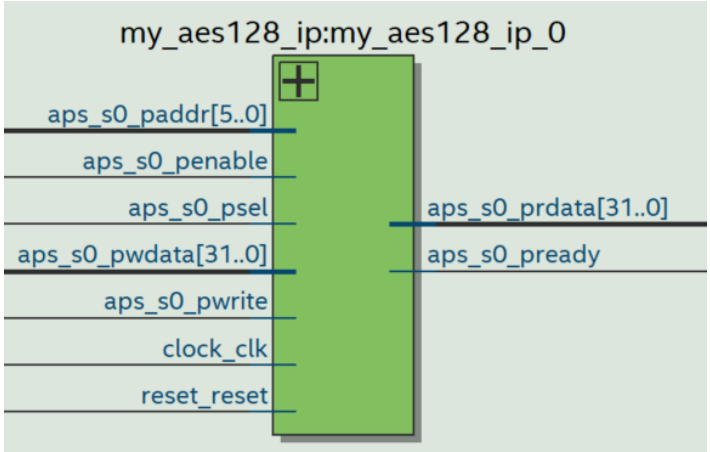
The AES 128-bit coprocessor was designed as a custom APB IP using the Platform Designer software tool. The my_aes_128_ip module is interfaced to the SoC HPS processor using the APB interface protocol. Your ENGG*4560 project needs to fully implement the hardware interface to the my_aes_128_ip module following the laboratory project application requirements. Follow the *ENGG*4560 Laboratory Manual* and *ENGG*4560 Laboratory Project Tutorial: Design of Custom APB IP with the Platform Designer*. The provided AES module is the main coprocessor module; however, you are responsible for implementing

the hardware interface to this module in order to accomplish the project requirements. As a further note, you can use any AES module that you can find on GitHub, or you could implement your own AES module following the AES algorithm and my AES implementation example.

Working on the ENGG*4560 project, it is important for the students to take time and understand the functionality of the AES algorithm and its hardware implementation. Therefore, I will present only the working Verilog code for my AES coprocessor implementation in the Appendix of this tutorial.

AES Coprocessor Hardware Interface Architecture

The figure below shows the my_aes128_ip component with the APB interface signals to the SoC interconnect. Follow the "ENGG4560 Laboratory Tutorial: Design of Custom APB IP with the Platform Designer" to build the APB interface.

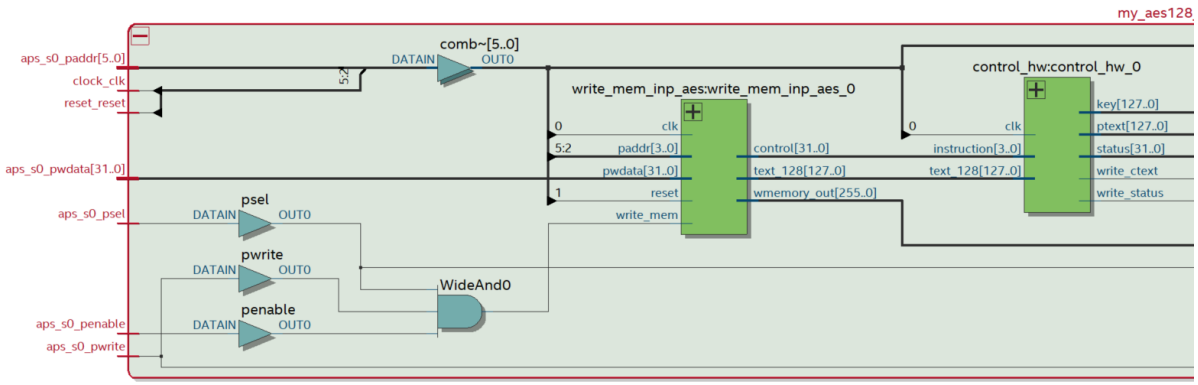


The following two figures present the architectural design of my hardware interface that I designed for the AES coprocessor following the Custom APB IP design process. In the first figure, you can see the APB input signals connections, and in the second figure, you can see the APB output signals connections. The hardware interface contains five main modules that are instantiated as building components. These are:

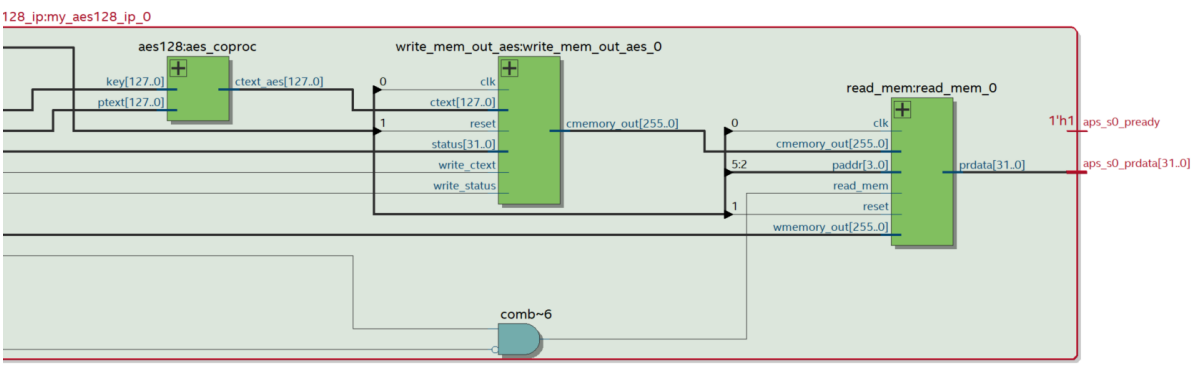
1. write_mem_inp_aes: this is a memory block module that contains 8x32-bit memory locations that are memory-mapped and are used to write the input data to the AES coprocessor, specifically, the control word, the 128-bit plaintext, and the 128-bit key.
2. write_mem_out_aes: this is a memory block module that contains 8x32-bit memory locations that are memory-mapped and are used to write the outputs from the AES coprocessor, specifically, the status word and the 128-bit ciphertext.
3. control_hw: this is the main control module of the hardware interface. I have a set of simple instructions that I designed for controlling the AES coprocessor. These instructions are: IDLE, PTEXT_ld, KEY_ld, start_encryption, CTEXT_read; Note that more instructions can be added as required for the interface functionality. These instructions are the ones that I used to perform simple 128-bit AES encryptions. For my case, I implemented the control module as a simple decoder generating the required signals.
4. aes128: this is the main AES-128 encryption module that performs asynchronous encryptions. When I say that the module performs asynchronous encryptions, it means that the module is designed entirely as a combinational circuitry. The inputs to the module are the 128-bit plaintext and the 128-bit key. Whenever the inputs are changed, the 128-bit ciphertext is generated at the

module's output after the signal propagation delay through all the encryption rounds. The module develops the round keys and performs all the AES rounds in a chaining manner as described in the next AES Coprocessor Section.

5. read_mem: this module reads the memory-mapped locations that are written by the write_mem_inp_aes and write_mem_out_aes modules. The inputs to this module are the memory output from the write modules as seen in the second figure.



...

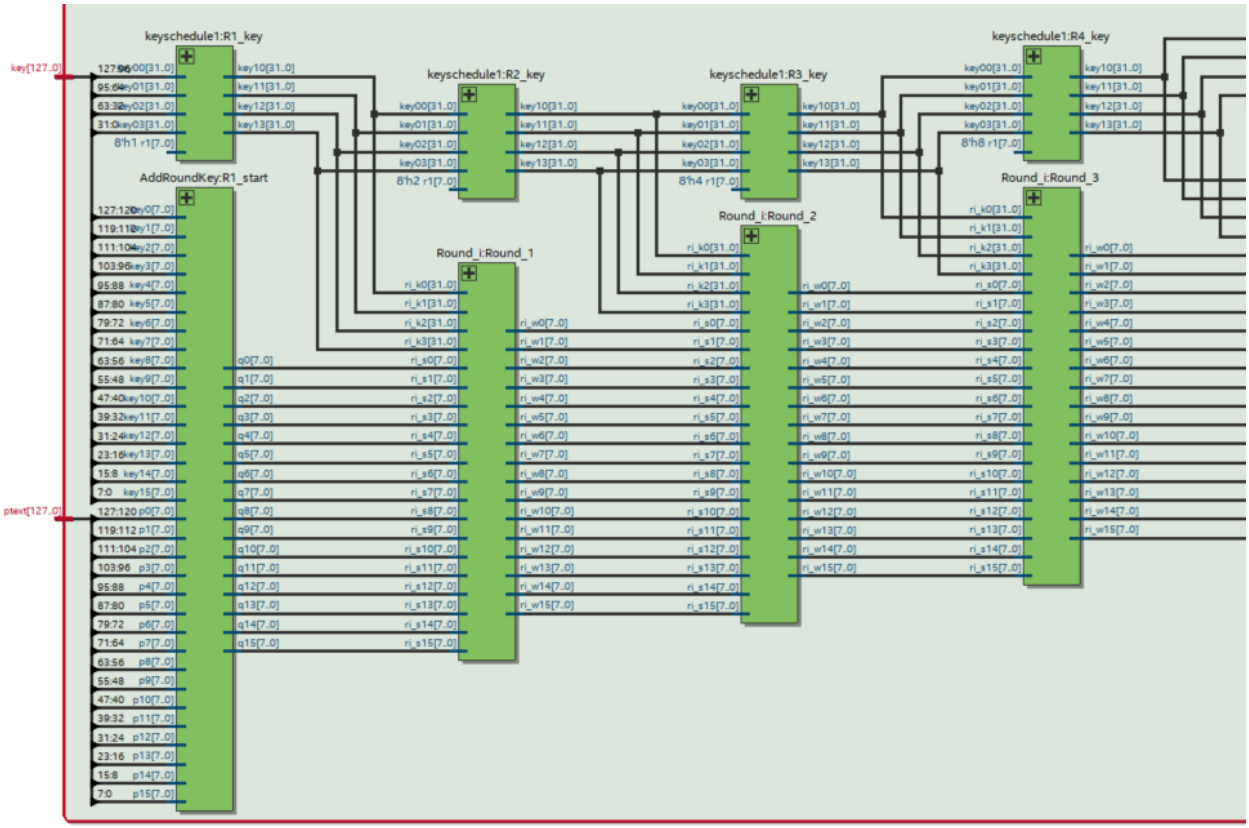


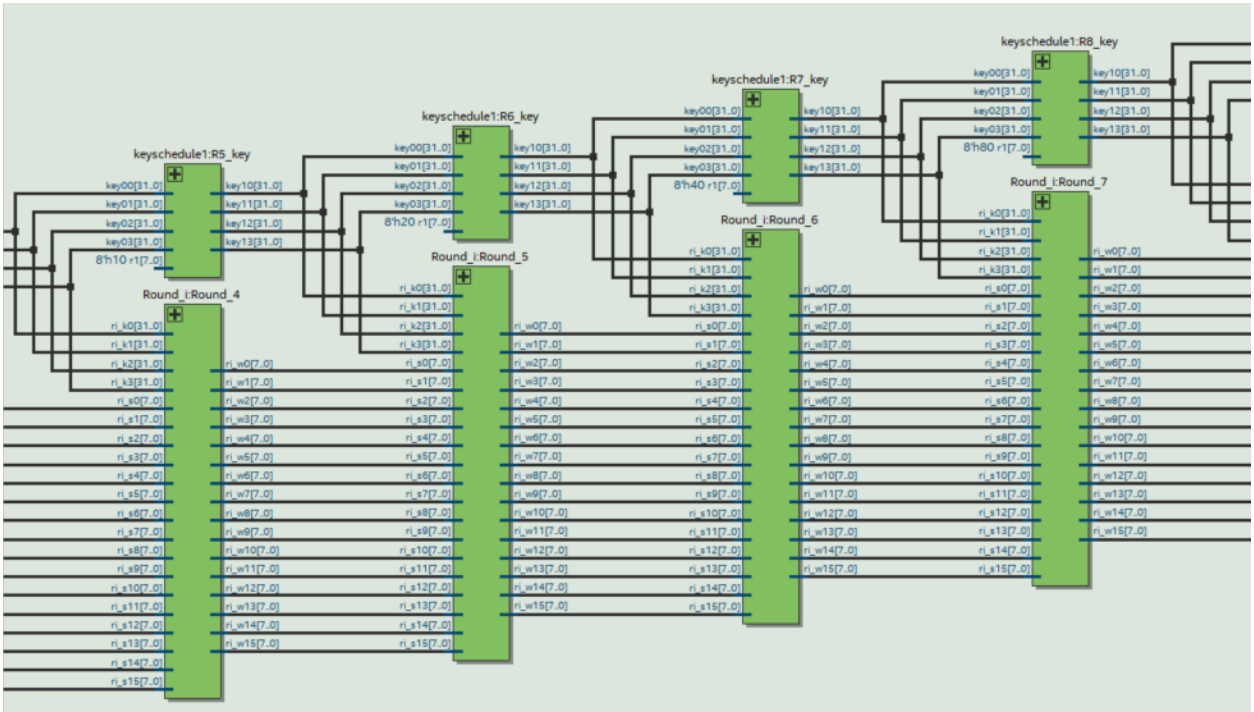
AES Coprocessor Hardware Interface Architecture

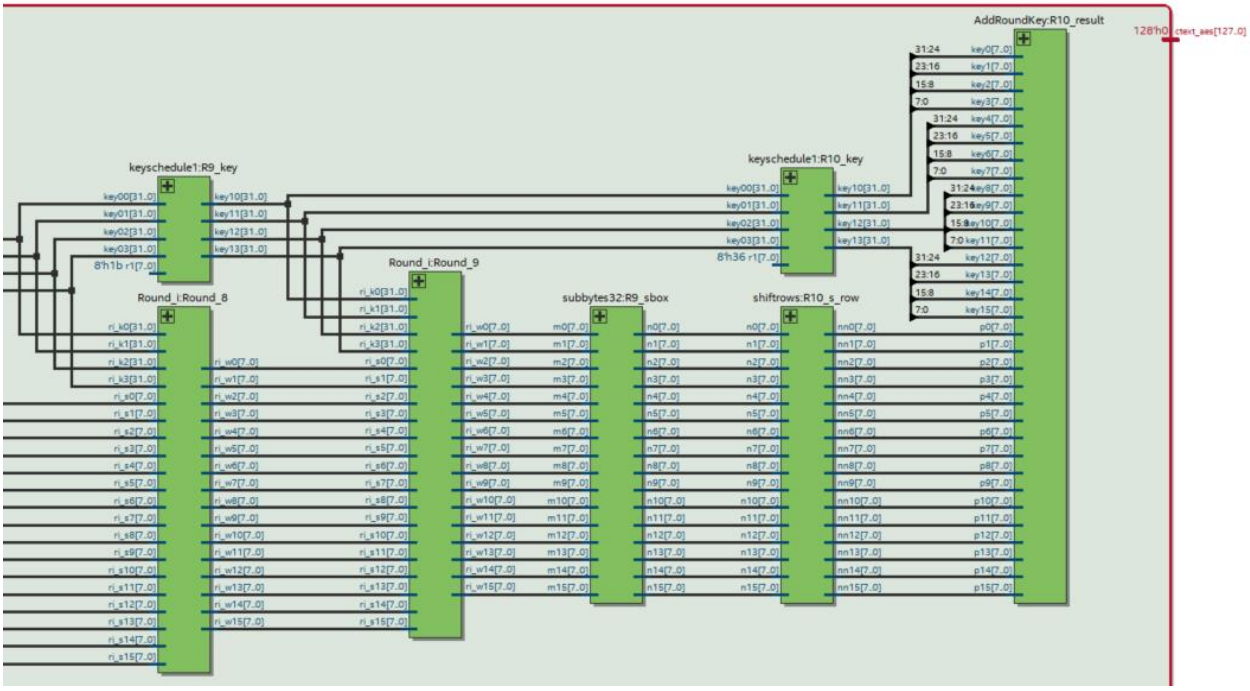
This section will briefly describe the functionality of my AES-128 encryption Verilog module. To understand the implementation, please follow the course material for the AES encryption algorithm plus references [1] to [5].

The AES-128 coprocessor Verilog code is presented in the Appendix my_aes128_coprocessor.v of this tutorial. This section will explain the AES implementation by referring to the code lines in the Appendix. The code presents the APB hardware interface to the AES-128 coprocessor designed as a custom IP module (follow the ENGG4560 Laboratory Tutorial Design of Custom APB IP with the Platform Designer). However, you will need to fully implement the hardware interface to your coprocessor using the memory mapping technique. In addition, your hardware interface must support the project requirements and interface to the AES-128 coprocessor. For example, for the AES-128 encryption coprocessor, you could use the code presented in this tutorial, implement your version, or utilize a different implementation from online resources.

The subsequent three figures present the hardware architecture of my AES-128 coprocessor design. The hardware architecture follows the Cipher pseudocode from Figure 5 and the Key Expansion pseudocode from Figure 11 found in reference [1].


keyschedule1:R1_key
keyschedule1:R2_key
keyschedule1:R3_key
keyschedule1:R4_key
AddRoundKey:R1_start
Round_i:Round_1
Round_i:Round_2
Round_i:Round_3
key[127..0]
ptext[127..0]


keyschedule1:R5_key
keyschedule1:R6_key
keyschedule1:R7_key
keyschedule1:R8_key
Round_i:Round_4
Round_i:Round_5
Round_i:Round_6
Round_i:Round_7



My AES-128 Verilog code can be viewed as two distinct sections:

Section 1: my_aes128_coprocessor (…) main module (code lines 47 to 365). This module can be divided further into four independent subsections, as shown below:

1. Input state data organization (code lines 48 – 85).
2. The Key Expansion algorithm section (code lines 90 – 180). The Key Expansion section follows the pseudocode from Figure 11 of reference [1] and Fig. 3.1 diagram of reference [5].
3. AES-128 cipher algorithm section (code lines 181 – 346). The cipher algorithm follows the pseudocode rounds algorithm (whitening round, rounds 1 to 9, and final round 10) presented in Figure 5 [1]. The rounds are implemented using the required state transformations as independent components (See Figure 5 algorithm [1]). As a note, you can optimize the code by using "for loop." However, you must ensure that your "for loop" is synthesizable.
4. Ciphertext result generation section (code lines 348 to 365).

Section 2: Cipher transformations modules that form the fundamental transformations for the AES-128 cipher algorithm. These modules form the essential components of the cipher rounds. In this section, I implemented seven modules used by the cipher algorithm and one module used by the key expansion algorithm.

1. AddRoundKey (…) module (code lines 369 to 439): this module implements the AddRoundKey () transformation as described in Section 5.1.4 of reference [1]. The module simply implements bitwise xor operations between the round key and the state.
2. subbytes8 (…) module (code lines 442 to 535): this module implements the S-Box operation on an 8-bit input using the arithmetic operations in the finite field GF(28). The S-Box operation is implemented using combinational logic equations. In this implementation, the operations of the

AES-128 cipher in GF(28) field are viewed as an extension of the GF(24) field. The S-Box transformation for an 8-bit input was implemented following Fig. 2 architecture and the logic equations presented in reference [3].

3. subbytes32 (…) module (code lines 537 to 590): this module implements the SubBytes () transformation of the state as described in Section 5.1.1 of reference [1], using the basic subbytes8 (…) module as components.
4. shiftrows (…) module (code lines 592 to 633): this module implements the ShiftRows () transformation as described in Section 5.1.2 of reference [1]. The module performs simple wire connections from the input state to the output state.
5. mixcolums1 (…) module (code lines 635 to 691): this module implements the MixColums () transformation for one column of the state as presented in Section 5.1.3 of reference [1]. The module uses a combinational logic implementation diagram following Fig. 4 and the method illustrated in reference [4]. The entire column transformation is implemented using buffer and xor gates (see module code lines).
6. mixcolums4 (…) module (code lines 693 to 736): this module implements the MixColums () operations for the 4-columns of the state.
7. Keyschedule1 (…) module (code lines 738 to 775): this module implements the general logical equations as applied to an individual Key Expansion round (See Section 5.2 in reference [1]). This module is the primary component for generating the round keys in Section 1 of the code.
8. Round_i (…) module (code lines 777 to 858): this module implements the general round operations of the AES cipher by instantiating the round transformations in order: subbytes32 (), shiftrows (), mixcolums4 (), and AddRoundKey (). This module is instantiated as a component for rounds 1 to 9 in Section 1 of the code. The whitening round (round 0) and round 10 of the AES-128 cipher are implemented by instantiating the components implementing their required operations.

Final Note: You can optimize this code or use it as presented in the Appendix of this tutorial. The code has a large propagation delay due to its chaining rounds implementation style. Synthesizing the code, you will get a warning regarding this large propagation delay. However, the overall propagation delay is not a problem for generating the ciphertext result. I had no issues reading the correct ciphertext in my hardware interface implementation since the HPS processor issues the coprocessor encryption instructions through the C code software driver, and there are many clock cycles between these instructions being sent to the coprocessor.

APPENDIX: my_aes128_coprocessor.v Verilog code and references [1], [3], and [4]

```
assign B12 = ptext [31:24];
assign B13 = ptext [23:16];
assign B14 = ptext [15:8];
assign B15 = ptext [7:0];
assign key00 = key [127:96];
assign key01 = key [95:64];
assign key02 = key [63:32];
assign key03 = key [31:0];
assign ctext_aes [127:96] = ctext0;
assign ctext_aes [95:64] = ctext1;
assign ctext_aes [63:32] = ctext2;
assign ctext_aes [31:0] = ctext3;

//round keys generation section generates the 10 round keys for rounds 1 to 10
//the keyschedule algorithm for each round key follows the
//pseudocode for Key Expanaion from Figure 11 of AES_NIST.FIPS.197.pdf document
wire [31:0] key10; wire [31:0] key11;
wire [31:0] key12; wire [31:0] key13;
wire [7:0] ri1;
assign ri1 = 8'b00000001; // constant ri for round 1
keyschedule1 R1_key(
         ri1,
         key00, key01, key02, key03,
         key10, key11, key12, key13
         );
wire [31:0] key20; wire [31:0] key21;
wire [31:0] key22; wire [31:0] key23;
wire [7:0] ri2;
assign ri2 = 8'b00000010; // constant ri for round 2
keyschedule1 R2_key(
         ri2,
         key10, key11, key12, key13,
         key20, key21, key22, key23
         );
wire [31:0] key30; wire [31:0] key31;
wire [31:0] key32; wire [31:0] key33;
wire [7:0] ri3;
assign ri3 = 8'b00000100; // constant ri for round 3
keyschedule1 R3_key(
         ri3,
         key20, key21, key22, key23,
         key30, key31, key32, key33
         );
wire [31:0] key40; wire [31:0] key41;
wire [31:0] key42; wire [31:0] key43;
wire [7:0] ri4;
assign ri4 = 8'b00001000; // constant ri for round 4
keyschedule1 R4_key(
         ri4,
         key30, key31, key32, key33,
         key40, key41, key42, key43
         );
wire [31:0] key50; wire [31:0] key51;
wire [31:0] key52; wire [31:0] key53;
wire [7:0] ri5;
assign ri5 = 8'b00010000; // constant ri for round 5
keyschedule1 R5_key(
         ri5,
         key40, key41, key42, key43,
         key50, key51, key52, key53
         );
wire [31:0] key60; wire [31:0] key61;
wire [31:0] key62; wire [31:0] key63;
wire [7:0] ri6;
assign ri6 = 8'b00100000; // constant ri for round 6
keyschedule1 R6_key(
         ri6,
         key50, key51, key52, key53,
         key60, key61, key62, key63
         );
wire [31:0] key70; wire [31:0] key71;
wire [31:0] key72; wire [31:0] key73;
wire [7:0] ri7;
```

```
/* my_aes128_ip_coprocesor.v; AES 128 encryption coprocessor
Developed by Radu Muresan, March 2022

This AES implementation is an asynchronous, combinational implementation
The module implements the encryption algorithm and the key expansion algorithm
The ciphertext result is generated after the propagation delay through the AES rounds
The AES coprocessor was tested with the NIST example found in AES_NIST.FIPS.197.pdf
Plaintext input:   32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Cipher Key:        2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
Ciphertext result: 39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
The ciphertext result was correctly generated and verified!

Implementation Notes!
The AES-128 cipher algorithm follows the pseudo-code of Figure 5 (pp. 15) presented in
AES_NIST.FIPS.197.pdf document. The implementation was developed by following
the Appendix B example round by round and state by state.
For the AES-128 algorithm, we have the length of the input block, output block, and
the state as 128-bits, and Nb = 4, Nk = 4, and Nr = 10 (See Figure 4, pp. 13).
*/

`timescale 1 ps / 1 ps
//input and output interface ports for the APB interface
module my_aes128_ip (
      input  wire [5:0]  aps_s0_paddr,   // aps_s0.paddr, 6-bits due to APB byte aligning
      input  wire        aps_s0_psel,    //       .psel
      input  wire        aps_s0_penable, //       .penable
      input  wire        aps_s0_pwrite,  //       .pwrite
      input  wire [31:0] aps_s0_pwdata,  //       .pwdata
      output wire [31:0] aps_s0_prdata,  //       .prdata
      output wire        aps_s0_pready,  //       .pready
      input  wire        clock_clk,      //  clock.clk
      input  wire        reset_reset     //  reset.reset
   );

/* This module implements the memory mapped hardware interface to
 the AES coprocessor module.
 the hardware interface must be implemented as part of the project requirements
*/

   assign aps_s0_pready = 1'b1; // no wait states required
   ...

// Follow the laboratory manual information for implementing the hardware interface

endmodule

//This is the AES coprocessor module code implementation
module my_aes128_coprocessor ( input wire [127:0] ptext,
                input wire [127:0] key,
                output wire [127:0] ctext_aes
                );

//organize the initial state; see Figure 3 of AES_NIST.FIPS.197.pdf document
wire [7:0] B0; wire [7:0] B1; wire [7:0] B2; wire [7:0] B3;
wire [7:0] B4; wire [7:0] B5; wire [7:0] B6; wire [7:0] B7;
wire [7:0] B8; wire [7:0] B9; wire [7:0] B10; wire [7:0] B11;
wire [7:0] B12; wire [7:0] B13; wire [7:0] B14; wire [7:0] B15;
wire [31:0] key00; wire [31:0] key01; wire [31:0] key02; wire [31:0] key03;
wire [31:0] ctext0; wire [31:0] ctext1; wire [31:0] ctext2; wire [31:0] ctext3;

//organize the input and output signals
assign B0 = ptext [127:120];
assign B1 = ptext [119:112];
assign B2 = ptext [111:104];
assign B3 = ptext [103:96];
assign B4 = ptext [95:88];
assign B5 = ptext [87:80];
assign B6 = ptext [79:72];
assign B7 = ptext [71:64];
assign B8 = ptext [63:56];
assign B9 = ptext [55:48];
assign B10 = ptext [47:40];
assign B11 = ptext [39:32];
```

```
assign ri7 = 8'b01000000; // constant ri for round 7
keyschedule1 R7_key(
        ri7,
        key60, key61, key62, key63,
        key70, key71, key72, key73
        );
wire [31:0] key80; wire [31:0] key81;
wire [31:0] key82; wire [31:0] key83;
wire [7:0] ri8;
assign ri8 = 8'b10000000; // constant ri for round 8
keyschedule1 R8_key(
        ri8,
        key70, key71, key72, key73,
        key80, key81, key82, key83
        );
wire [31:0] key90; wire [31:0] key91;
wire [31:0] key92; wire [31:0] key93;
wire [7:0] ri9;
assign ri9 = 8'b00011011; // constant ri for round 9
keyschedule1 R9_key(
        ri9,
        key80, key81, key82, key83,
        key90, key91, key92, key93
        );
wire [31:0] key100; wire [31:0] key101;
wire [31:0] key102; wire [31:0] key103;
wire [7:0] ri10;
assign ri10 = 8'b00110110; // constant ri for round 10
keyschedule1 R10_key(
        ri10,
        key90, key91, key92, key93,
        key100, key101, key102, key103
        );

//AES-128 has one whitening round (round 0) plus 9 regular rounds (rounds 1 to 9)
//and a final round (round 10)

//transfer the ptext to R0 - whitening round

wire [7:0] r0_w0; wire [7:0] r0_w1; wire [7:0] r0_w2; wire [7:0] r0_w3;
wire [7:0] r0_w4; wire [7:0] r0_w5; wire [7:0] r0_w6; wire [7:0] r0_w7;
wire [7:0] r0_w8; wire [7:0] r0_w9; wire [7:0] r0_w10; wire [7:0] r0_w11;
wire [7:0] r0_w12; wire [7:0] r0_w13; wire [7:0] r0_w14; wire [7:0] r0_w15;

//perform the round 0 which is just the AddRoundKey

AddRoundKey R1_start (key00[31:24], key00[23:16], key00[15:8], key00[7:0],
                key01[31:24], key01[23:16], key01[15:8], key01[7:0],
                key02[31:24], key02[23:16], key02[15:8], key02[7:0],
                key03[31:24], key03[23:16], key03[15:8], key03[7:0],
                B0, B1, B2, B3, B4, B5, B6, B7,
                B8, B9, B10, B11, B12, B13, B14, B15,
                r0_w0, r0_w1, r0_w2, r0_w3, r0_w4, r0_w5, r0_w6, r0_w7,
                r0_w8, r0_w9, r0_w10, r0_w11, r0_w12, r0_w13, r0_w14, r0_w15
                );

//next is round 1
wire [7:0] r1_w0; wire [7:0] r1_w1; wire [7:0] r1_w2; wire [7:0] r1_w3;
wire [7:0] r1_w4; wire [7:0] r1_w5; wire [7:0] r1_w6; wire [7:0] r1_w7;
wire [7:0] r1_w8; wire [7:0] r1_w9; wire [7:0] r1_w10; wire [7:0] r1_w11;
wire [7:0] r1_w12; wire [7:0] r1_w13; wire [7:0] r1_w14; wire [7:0] r1_w15;

Round_i Round_1 (r0_w0, r0_w1, r0_w2, r0_w3, r0_w4, r0_w5, r0_w6, r0_w7,
                r0_w8, r0_w9, r0_w10, r0_w11, r0_w12, r0_w13, r0_w14, r0_w15,
                key10, key11, key12, key13,
                r1_w0, r1_w1, r1_w2, r1_w3, r1_w4, r1_w5, r1_w6, r1_w7,
                r1_w8, r1_w9, r1_w10, r1_w11, r1_w12, r1_w13, r1_w14, r1_w15
                );
//next is round 2
wire [7:0] r2_w0; wire [7:0] r2_w1; wire [7:0] r2_w2; wire [7:0] r2_w3;
wire [7:0] r2_w4; wire [7:0] r2_w5; wire [7:0] r2_w6; wire [7:0] r2_w7;
wire [7:0] r2_w8; wire [7:0] r2_w9; wire [7:0] r2_w10; wire [7:0] r2_w11;
wire [7:0] r2_w12; wire [7:0] r2_w13; wire [7:0] r2_w14; wire [7:0] r2_w15;
```

```
220
221   Round_i Round_2 (r1_w0, r1_w1, r1_w2, r1_w3, r1_w4, r1_w5, r1_w6, r1_w7,
222                    r1_w8, r1_w9, r1_w10, r1_w11, r1_w12, r1_w13, r1_w14, r1_w15,
223                    key20, key21, key22, key23,
224                    r2_w0, r2_w1, r2_w2, r2_w3, r2_w4, r2_w5, r2_w6, r2_w7,
225                    r2_w8, r2_w9, r2_w10, r2_w11, r2_w12, r2_w13, r2_w14, r2_w15
226                    );
227
228   //next is round 3
229   wire [7:0] r3_w0; wire [7:0] r3_w1; wire [7:0] r3_w2; wire [7:0] r3_w3;
230   wire [7:0] r3_w4; wire [7:0] r3_w5; wire [7:0] r3_w6; wire [7:0] r3_w7;
231   wire [7:0] r3_w8; wire [7:0] r3_w9; wire [7:0] r3_w10; wire [7:0] r3_w11;
232   wire [7:0] r3_w12; wire [7:0] r3_w13; wire [7:0] r3_w14; wire [7:0] r3_w15;
233
234   Round_i Round_3 (r2_w0, r2_w1, r2_w2, r2_w3, r2_w4, r2_w5, r2_w6, r2_w7,
235                    r2_w8, r2_w9, r2_w10, r2_w11, r2_w12, r2_w13, r2_w14, r2_w15,
236                    key30, key31, key32, key33,
237                    r3_w0, r3_w1, r3_w2, r3_w3, r3_w4, r3_w5, r3_w6, r3_w7,
238                    r3_w8, r3_w9, r3_w10, r3_w11, r3_w12, r3_w13, r3_w14, r3_w15
239                    );
240   //next is round 4
241   wire [7:0] r4_w0; wire [7:0] r4_w1; wire [7:0] r4_w2; wire [7:0] r4_w3;
242   wire [7:0] r4_w4; wire [7:0] r4_w5; wire [7:0] r4_w6; wire [7:0] r4_w7;
243   wire [7:0] r4_w8; wire [7:0] r4_w9; wire [7:0] r4_w10; wire [7:0] r4_w11;
244   wire [7:0] r4_w12; wire [7:0] r4_w13; wire [7:0] r4_w14; wire [7:0] r4_w15;
245
246   Round_i Round_4 (r3_w0, r3_w1, r3_w2, r3_w3, r3_w4, r3_w5, r3_w6, r3_w7,
247                    r3_w8, r3_w9, r3_w10, r3_w11, r3_w12, r3_w13, r3_w14, r3_w15,
248                    key40, key41, key42, key43,
249                    r4_w0, r4_w1, r4_w2, r4_w3, r4_w4, r4_w5, r4_w6, r4_w7,
250                    r4_w8, r4_w9, r4_w10, r4_w11, r4_w12, r4_w13, r4_w14, r4_w15
251                    );
252   //next is round 5
253   wire [7:0] r5_w0; wire [7:0] r5_w1; wire [7:0] r5_w2; wire [7:0] r5_w3;
254   wire [7:0] r5_w4; wire [7:0] r5_w5; wire [7:0] r5_w6; wire [7:0] r5_w7;
255   wire [7:0] r5_w8; wire [7:0] r5_w9; wire [7:0] r5_w10; wire [7:0] r5_w11;
256   wire [7:0] r5_w12; wire [7:0] r5_w13; wire [7:0] r5_w14; wire [7:0] r5_w15;
257
258   Round_i Round_5 (r4_w0, r4_w1, r4_w2, r4_w3, r4_w4, r4_w5, r4_w6, r4_w7,
259                    r4_w8, r4_w9, r4_w10, r4_w11, r4_w12, r4_w13, r4_w14, r4_w15,
260                    key50, key51, key52, key53,
261                    r5_w0, r5_w1, r5_w2, r5_w3, r5_w4, r5_w5, r5_w6, r5_w7,
262                    r5_w8, r5_w9, r5_w10, r5_w11, r5_w12, r5_w13, r5_w14, r5_w15
263                    );
264   //next is round 6
265   wire [7:0] r6_w0; wire [7:0] r6_w1; wire [7:0] r6_w2; wire [7:0] r6_w3;
266   wire [7:0] r6_w4; wire [7:0] r6_w5; wire [7:0] r6_w6; wire [7:0] r6_w7;
267   wire [7:0] r6_w8; wire [7:0] r6_w9; wire [7:0] r6_w10; wire [7:0] r6_w11;
268   wire [7:0] r6_w12; wire [7:0] r6_w13; wire [7:0] r6_w14; wire [7:0] r6_w15;
269
270   Round_i Round_6 (r5_w0, r5_w1, r5_w2, r5_w3, r5_w4, r5_w5, r5_w6, r5_w7,
271                    r5_w8, r5_w9, r5_w10, r5_w11, r5_w12, r5_w13, r5_w14, r5_w15,
272                    key60, key61, key62, key63,
273                    r6_w0, r6_w1, r6_w2, r6_w3, r6_w4, r6_w5, r6_w6, r6_w7,
274                    r6_w8, r6_w9, r6_w10, r6_w11, r6_w12, r6_w13, r6_w14, r6_w15
275                    );
276   //next is round 7
277   wire [7:0] r7_w0; wire [7:0] r7_w1; wire [7:0] r7_w2; wire [7:0] r7_w3;
278   wire [7:0] r7_w4; wire [7:0] r7_w5; wire [7:0] r7_w6; wire [7:0] r7_w7;
279   wire [7:0] r7_w8; wire [7:0] r7_w9; wire [7:0] r7_w10; wire [7:0] r7_w11;
280   wire [7:0] r7_w12; wire [7:0] r7_w13; wire [7:0] r7_w14; wire [7:0] r7_w15;
281
282   Round_i Round_7 (r6_w0, r6_w1, r6_w2, r6_w3, r6_w4, r6_w5, r6_w6, r6_w7,
283                    r6_w8, r6_w9, r6_w10, r6_w11, r6_w12, r6_w13, r6_w14, r6_w15,
284                    key70, key71, key72, key73,
285                    r7_w0, r7_w1, r7_w2, r7_w3, r7_w4, r7_w5, r7_w6, r7_w7,
286                    r7_w8, r7_w9, r7_w10, r7_w11, r7_w12, r7_w13, r7_w14, r7_w15
287                    );
288   //next is round 8
289   wire [7:0] r8_w0; wire [7:0] r8_w1; wire [7:0] r8_w2; wire [7:0] r8_w3;
290   wire [7:0] r8_w4; wire [7:0] r8_w5; wire [7:0] r8_w6; wire [7:0] r8_w7;
291   wire [7:0] r8_w8; wire [7:0] r8_w9; wire [7:0] r8_w10; wire [7:0] r8_w11;
292   wire [7:0] r8_w12; wire [7:0] r8_w13; wire [7:0] r8_w14; wire [7:0] r8_w15;
```

```
293
294   Round_i Round_8 (r7_w0, r7_w1, r7_w2, r7_w3, r7_w4, r7_w5, r7_w6, r7_w7,
295                    r7_w8, r7_w9, r7_w10, r7_w11, r7_w12, r7_w13, r7_w14, r7_w15,
296                    key80, key81, key82, key83,
297                    r8_w0, r8_w1, r8_w2, r8_w3, r8_w4, r8_w5, r8_w6, r8_w7,
298                    r8_w8, r8_w9, r8_w10, r8_w11, r8_w12, r8_w13, r8_w14, r8_w15
299                    );
300   //next is round 9
301   wire [7:0] r9_w0; wire [7:0] r9_w1; wire [7:0] r9_w2; wire [7:0] r9_w3;
302   wire [7:0] r9_w4; wire [7:0] r9_w5; wire [7:0] r9_w6; wire [7:0] r9_w7;
303   wire [7:0] r9_w8; wire [7:0] r9_w9; wire [7:0] r9_w10; wire [7:0] r9_w11;
304   wire [7:0] r9_w12; wire [7:0] r9_w13; wire [7:0] r9_w14; wire [7:0] r9_w15;
305
306   Round_i Round_9 (r8_w0, r8_w1, r8_w2, r8_w3, r8_w4, r8_w5, r8_w6, r8_w7,
307                    r8_w8, r8_w9, r8_w10, r8_w11, r8_w12, r8_w13, r8_w14, r8_w15,
308                    key90, key91, key92, key93,
309                    r9_w0, r9_w1, r9_w2, r9_w3, r9_w4, r9_w5, r9_w6, r9_w7,
310                    r9_w8, r9_w9, r9_w10, r9_w11, r9_w12, r9_w13, r9_w14, r9_w15
311                    );
312   //next is round 10, the final round
313
314   wire [7:0] r10_n0; wire [7:0] r10_n1; wire [7:0] r10_n2; wire [7:0] r10_n3;
315   wire [7:0] r10_n4; wire [7:0] r10_n5; wire [7:0] r10_n6; wire [7:0] r10_n7;
316   wire [7:0] r10_n8; wire [7:0] r10_n9; wire [7:0] r10_n10; wire [7:0] r10_n11;
317   wire [7:0] r10_n12; wire [7:0] r10_n13; wire [7:0] r10_n14; wire [7:0] r10_n15;
318   subbytes32 R9_sbox (r9_w0, r9_w1, r9_w2, r9_w3, r9_w4, r9_w5, r9_w6, r9_w7,
319                       r9_w8, r9_w9, r9_w10, r9_w11, r9_w12, r9_w13, r9_w14, r9_w15,
320                       r10_n0, r10_n1, r10_n2, r10_n3, r10_n4, r10_n5, r10_n6, r10_n7,
321                       r10_n8, r10_n9, r10_n10, r10_n11, r10_n12, r10_n13, r10_n14, r10_n15);
322
323   wire [7:0] r10_nn0; wire [7:0] r10_nn1; wire [7:0] r10_nn2; wire [7:0] r10_nn3;
324   wire [7:0] r10_nn4; wire [7:0] r10_nn5; wire [7:0] r10_nn6; wire [7:0] r10_nn7;
325   wire [7:0] r10_nn8; wire [7:0] r10_nn9; wire [7:0] r10_nn10; wire [7:0] r10_nn11;
326   wire [7:0] r10_nn12; wire [7:0] r10_nn13; wire [7:0] r10_nn14; wire [7:0] r10_nn15;
327
328   shiftrows R10_s_row(r10_n0, r10_n1, r10_n2, r10_n3, r10_n4, r10_n5, r10_n6, r10_n7,
329            r10_n8, r10_n9, r10_n10, r10_n11, r10_n12, r10_n13, r10_n14, r10_n15,
330            r10_nn0, r10_nn1, r10_nn2, r10_nn3, r10_nn4, r10_nn5, r10_nn6, r10_nn7,
331            r10_nn8, r10_nn9, r10_nn10, r10_nn11, r10_nn12, r10_nn13, r10_nn14, r10_nn15);
332
333   wire [7:0] r10_w0; wire [7:0] r10_w1; wire [7:0] r10_w2; wire [7:0] r10_w3;
334   wire [7:0] r10_w4; wire [7:0] r10_w5; wire [7:0] r10_w6; wire [7:0] r10_w7;
335   wire [7:0] r10_w8; wire [7:0] r10_w9; wire [7:0] r10_w10; wire [7:0] r10_w11;
336   wire [7:0] r10_w12; wire [7:0] r10_w13; wire [7:0] r10_w14; wire [7:0] r10_w15;
337
338   AddRoundKey R10_result (key100[31:24], key100[23:16], key100[15:8], key100[7:0],
339                    key101[31:24], key101[23:16], key101[15:8], key101[7:0],
340                    key102[31:24], key102[23:16], key102[15:8], key102[7:0],
341                    key103[31:24], key103[23:16], key103[15:8], key103[7:0],
342                    r10_nn0, r10_nn1, r10_nn2, r10_nn3, r10_nn4, r10_nn5, r10_nn6, r10_nn7,
343                    r10_nn8, r10_nn9, r10_nn10, r10_nn11, r10_nn12, r10_nn13, r10_nn14, r10_nn15,
344                    r10_w0, r10_w1, r10_w2, r10_w3, r10_w4, r10_w5, r10_w6, r10_w7,
345                    r10_w8, r10_w9, r10_w10, r10_w11, r10_w12, r10_w13, r10_w14, r10_w15
346                    );
347
348   //the following assign statements write the final ciphertext result to the output
349
350   assign ctext0[31:24] = r10_w0;
351   assign ctext0[23:16] = r10_w1;
352   assign ctext0[15:8] = r10_w2;
353   assign ctext0[7:0] = r10_w3;
354   assign ctext1[31:24] = r10_w4;
355   assign ctext1[23:16] = r10_w5;
356   assign ctext1[15:8] = r10_w6;
357   assign ctext1[7:0] = r10_w7;
358   assign ctext2[31:24] = r10_w8;
359   assign ctext2[23:16] = r10_w9;
360   assign ctext2[15:8] = r10_w10;
361   assign ctext2[7:0] = r10_w11;
362   assign ctext3[31:24] = r10_w12;
363   assign ctext3[23:16] = r10_w13;
364   assign ctext3[15:8] = r10_w14;
365   assign ctext3[7:0] = r10_w15;
```

```
366
367   endmodule
368
369   module AddRoundKey (
370           input wire [7:0] key0,
371           input wire [7:0] key1,
372           input wire [7:0] key2,
373           input wire [7:0] key3,
374           input wire [7:0] key4,
375           input wire [7:0] key5,
376           input wire [7:0] key6,
377           input wire [7:0] key7,
378           input wire [7:0] key8,
379           input wire [7:0] key9,
380           input wire [7:0] key10,
381           input wire [7:0] key11,
382           input wire [7:0] key12,
383           input wire [7:0] key13,
384           input wire [7:0] key14,
385           input wire [7:0] key15,
386           input wire [7:0] p0,
387           input wire [7:0] p1,
388           input wire [7:0] p2,
389           input wire [7:0] p3,
390           input wire [7:0] p4,
391           input wire [7:0] p5,
392           input wire [7:0] p6,
393           input wire [7:0] p7,
394           input wire [7:0] p8,
395           input wire [7:0] p9,
396           input wire [7:0] p10,
397           input wire [7:0] p11,
398           input wire [7:0] p12,
399           input wire [7:0] p13,
400           input wire [7:0] p14,
401           input wire [7:0] p15,
402           output wire [7:0] q0,
403           output wire [7:0] q1,
404           output wire [7:0] q2,
405           output wire [7:0] q3,
406           output wire [7:0] q4,
407           output wire [7:0] q5,
408           output wire [7:0] q6,
409           output wire [7:0] q7,
410           output wire [7:0] q8,
411           output wire [7:0] q9,
412           output wire [7:0] q10,
413           output wire [7:0] q11,
414           output wire [7:0] q12,
415           output wire [7:0] q13,
416           output wire [7:0] q14,
417           output wire [7:0] q15
418           );
419   //This module implements the AddRoundKey () transformation
420   //on the state through XOR bitwise operations
421   //(see Figure 10 in AES_NIST.FIPS.197.pdf document)
422   assign q0 = key0 ^ p0;
423   assign q1 = key1 ^ p1;
424   assign q2 = key2 ^ p2;
425   assign q3 = key3 ^ p3;
426   assign q4 = key4 ^ p4;
427   assign q5 = key5 ^ p5;
428   assign q6 = key6 ^ p6;
429   assign q7 = key7 ^ p7;
430   assign q8 = key8 ^ p8;
431   assign q9 = key9 ^ p9;
432   assign q10 = key10 ^ p10;
433   assign q11 = key11 ^ p11;
434   assign q12 = key12 ^ p12;
435   assign q13 = key13 ^ p13;
436   assign q14 = key14 ^ p14;
437   assign q15 = key15 ^ p15;
438
```

```
endmodule


module subbytes8 (input wire [7:0] a, output wire [7:0] q);
//this module implements the subbytes operation of an element in GF(2^8) using
//GF (2^4) operatoins following the equations presented in:
//An ASIC Implementation of the AES SBoxes paper by J. Wokerstorfer, et al.
//I follow the Fig. 2 schematic in implementing the subbyte operation in GF(2^8)
wire aA, aB, aC, al0, al1, al2, al3, ah0, ah1, ah2, ah3;
// convert from GF(2^8) to GF(2^f): a = ahx + al.

xor (aA, a[1], a[7]); xor (aB, a[5], a[7]); xor (aC, a[4], a[6]);
xor (al0, aC, a[0], a[5]); xor (al1, a[1], a[2]); buf (al2, aA); xor (al3, a[2], a[4]);
xor (ah0, aC, a[5]); xor (ah1, aA, aC); xor (ah2, aB, a[2], a[3]); buf (ah3, aB);
//square ah and al elements go get qh and ql
wire qh0, qh1, qh2, qh3;
xor (qh0, ah0, ah2); buf (qh1, ah2); xor (qh2, ah1, ah3); buf (qh3, ah3);
wire ql0, ql1, ql2, ql3;
xor (ql0, al0, al2); buf (ql1, al2); xor (ql2, al1, al3); buf (ql3, al3);

//multiply qh with e
wire qhe0, qhe1, qhe2, qhe3;
xor (qhe0, qh1, qh2, qh3); xor (qhe1, qh0, qh1); xor (qhe2, qh0, qh1, qh2); xor (qhe3, qh0, qh1, qh2, qh3);
//xor qhe and ql to get qhel
wire qhel0, qhel1, qhel2, qhel3;
xor (qhel0, qhe0, ql0); xor (qhel1, qhe1, ql1); xor (qhel2, qhe2, ql2); xor (qhel3, qhe3, ql3);


//multiply ah with al to get qhl
wire qhl0, qhl1, qhl2, qhl3, qhA, qhB;
xor (qhA, ah0, ah3); xor (qhB, ah2, ah3);
assign qhl0 = (ah0 & al0) ^ (ah3 & al1) ^ (ah2 & al2) ^ (ah1 & al3);
assign qhl1 = (ah1 & al0) ^ (qhA & al1) ^ (qhB & al2) ^ ((ah1 ^ ah2)& al3);
assign qhl2 = (ah2 & al0) ^ (ah1 & al1) ^ (qhA & al2) ^ (qhB & al3);
assign qhl3 = (ah3 & al0) ^ (ah2 & al1) ^ (ah1 & al2) ^ (qhA & al3);

// xor qhel with qhl to get xin
wire xin0, xin1, xin2, xin3;
xor (xin0, qhel0, qhl0); xor (xin1, qhel1, qhl1); xor (xin2, qhel2, qhl2); xor (xin3, qhel3, qhl3);
//invert xin to get inv
wire  inv0, inv1, inv2, inv3, aAinv;
assign aAinv = (xin1 ^ xin2 ^ xin3) ^ (xin1 & xin2 & xin3);
assign inv0 = aAinv ^ xin0 ^ (xin0 & xin2) ^ (xin1 & xin2) ^ (xin0 & xin1 & xin2);
assign inv1 = (xin0 & xin1) ^ (xin0 & xin2) ^ (xin1 & xin2) ^ xin3 ^ (xin1 & xin3) ^ (xin0 & xin1 & xin3);
assign inv2 = (xin0 & xin1) ^ xin2 ^ (xin0 & xin2) ^ xin3 ^ (xin0 & xin3) ^ (xin0 & xin2 & xin3);
assign inv3 = aAinv ^ (xin0 & xin3) ^ (xin1 & xin3) ^ (xin2 & xin3);
//calculate ah xor al to get xhl
wire xhl0, xhl1, xhl2, xhl3;
xor (xhl0, ah0, al0); xor (xhl1, ah1, al1); xor (xhl2, ah2, al2); xor (xhl3, ah3, al3);

//calculate ah times inv to get qmh
wire qmh0, qmh1, qmh2, qmh3;
//xor (qmhA, ah0, ah3); xor (qmhB, ah2, ah3); // same as above, reuse
assign qmh0 = (ah0 & inv0) ^ (ah3 & inv1) ^ (ah2 & inv2) ^ (ah1 & inv3);
assign qmh1 = (ah1 & inv0) ^ (qhA & inv1) ^ (qhB & inv2) ^ ((ah1 ^ ah2)& inv3);
assign qmh2 = (ah2 & inv0) ^ (ah1 & inv1) ^ (qhA & inv2) ^ (qhB & inv3);
assign qmh3 = (ah3 & inv0) ^ (ah2 & inv1) ^ (ah1 & inv2) ^ (qhA & inv3);

//calculate xhl times inv to get qml
wire qml0, qml1, qml2, qml3, xhlA, xhlB;
xor (xhlA, xhl0, xhl3); xor (xhlB, xhl2, xhl3);
assign qml0 = (xhl0 & inv0) ^ (xhl3 & inv1) ^ (xhl2 & inv2) ^ (xhl1 & inv3);
assign qml1 = (xhl1 & inv0) ^ (xhlA & inv1) ^ (xhlB & inv2) ^ ((xhl1 ^ xhl2)& inv3);
assign qml2 = (xhl2 & inv0) ^ (xhl1 & inv1) ^ (xhlA & inv2) ^ (xhlB & inv3);
assign qml3 = (xhl3 & inv0) ^ (xhl2 & inv1) ^ (xhl1 & inv2) ^ (xhlA & inv3);
// convert polynomial qmhX + qml from GF(2^4) back to GF(2^8)
wire gf0, gf1, gf2, gf3, gf4, gf5, gf6, gf7, qmA, qmB;
xor (qmA, qml1, qmh3); xor (qmB, qmh0, qmh1);
```

```
subbytes8 sbox6 (m6, n6);
subbytes8 sbox7 (m7, n7);
subbytes8 sbox8 (m8, n8);
subbytes8 sbox9 (m9, n9);
subbytes8 sbox10 (m10, n10);
subbytes8 sbox11 (m11, n11);
subbytes8 sbox12 (m12, n12);
subbytes8 sbox13 (m13, n13);
subbytes8 sbox14 (m14, n14);
subbytes8 sbox15 (m15, n15);
endmodule

module shiftrows (
        input wire [7:0] n0,
        input wire [7:0] n1,
        input wire [7:0] n2,
        input wire [7:0] n3,
        input wire [7:0] n4,
        input wire [7:0] n5,
        input wire [7:0] n6,
        input wire [7:0] n7,
        input wire [7:0] n8,
        input wire [7:0] n9,
        input wire [7:0] n10,
        input wire [7:0] n11,
        input wire [7:0] n12,
        input wire [7:0] n13,
        input wire [7:0] n14,
        input wire [7:0] n15,
        output wire [7:0] nn0,
        output wire [7:0] nn1,
        output wire [7:0] nn2,
        output wire [7:0] nn3,
        output wire [7:0] nn4,
        output wire [7:0] nn5,
        output wire [7:0] nn6,
        output wire [7:0] nn7,
        output wire [7:0] nn8,
        output wire [7:0] nn9,
        output wire [7:0] nn10,
        output wire [7:0] nn11,
        output wire [7:0] nn12,
        output wire [7:0] nn13,
        output wire [7:0] nn14,
        output wire [7:0] nn15
        );
//This module pefroms the ShiftRows () transformation as described in
//Figure 8 of the AES_NIST.FIPS.197.pdf document.
assign nn0 = n0; assign nn1 = n5; assign nn2 = n10; assign nn3 = n15;
assign nn4 = n4; assign nn5 = n9; assign nn6 = n14; assign nn7 = n3;
assign nn8 = n8; assign nn9 = n13; assign nn10 = n2; assign nn11 = n7;
assign nn12 = n12; assign nn13 = n1; assign nn14 = n6; assign nn15 = n11;

endmodule

module mixcolumns1 (
        input wire [7:0] b3,
        input wire [7:0] b2,
        input wire [7:0] b1,
        input wire [7:0] b0,
        output wire [7:0] w3,
        output wire [7:0] w2,
        output wire [7:0] w1,
        output wire [7:0] w0
        );
//This module implements the MixColums () transformation as described
//in AES_NIST.FIPS.197.pdf document
//But for the implementation of this transformation I use the algorithm
//presented in Fig. 5 of the paper:
//An efficient architecture for the AES mix columsn operation by H. Li, et al.
wire [7:0] b0x2; wire [7:0] b1x2; wire [7:0] b2x2; wire [7:0] b3x2;
buf (b0x2[0], b0[7]); xor (b0x2[1], b0[0], b0[7]);
buf (b0x2[2], b0[1]); xor (b0x2[3], b0[2], b0[7]);
```

```
507   xor (gf0, qml0, qmh0); xor (gf1, qmB, qmh3); xor (gf2, qmA, qmB); xor (gf3, qmB, qml1, qmh2);
508   xor (gf4, qmA, qmB, qml3); xor (gf5, qmB, qml2);
509   xor (gf6, qmA, qml2, qml3, qmh0); xor (gf7, qmB, qml2, qmh3);
510
511   //next I do the affine transformation applied to gf byte to get sbox byte
512   wire sbox0, sbox1, sbox2, sbox3, sbox4, sbox5, sbox6, sbox7, gfA, gfB, gfC, gfD;
513   wire ngf0, ngf5, ngf1, ngf6;
514   not (ngf0, gf0); not (ngf1, gf1); not (ngf5, gf5); not (ngf6, gf6);
515   xor (gfA, gf0, gf1); xor (gfB, gf2, gf3); xor (gfC, gf4, gf5); xor (gfD, gf6, gf7);
516   xor (sbox0, ngf0, gfC, gfD);
517   //xor (sbox0, gf0, gfC, gfD);
518   xor (sbox1, ngf5, gfA, gfD);
519   //xor (sbox1, gf5, gfA, gfD);
520   xor (sbox2, gf2, gfA, gfD);
521   xor (sbox3, gf7, gfA, gfB);
522   xor (sbox4, gf4, gfA, gfB);
523   xor (sbox5, ngf1, gfB, gfC);
524   //xor (sbox5, gf1, gfB, gfC);
525   xor (sbox6, ngf6, gfB, gfC);
526   //xor (sbox6, gf6, gfB, gfC);
527   xor (sbox7, gf3, gfC, gfD);
528   //
529   assign q[0] = sbox0; assign q[1] = sbox1;
530   assign q[2] = sbox2; assign q[3] = sbox3;
531   assign q[4] = sbox4; assign q[5] = sbox5;
532   assign q[6] = sbox6; assign q[7] = sbox7;
533
534
535   endmodule
536
537   module subbytes32 (
538           input wire [7:0] m0,
539           input wire [7:0] m1,
540           input wire [7:0] m2,
541           input wire [7:0] m3,
542           input wire [7:0] m4,
543           input wire [7:0] m5,
544           input wire [7:0] m6,
545           input wire [7:0] m7,
546           input wire [7:0] m8,
547           input wire [7:0] m9,
548           input wire [7:0] m10,
549           input wire [7:0] m11,
550           input wire [7:0] m12,
551           input wire [7:0] m13,
552           input wire [7:0] m14,
553           input wire [7:0] m15,
554           output wire [7:0] n0,
555           output wire [7:0] n1,
556           output wire [7:0] n2,
557           output wire [7:0] n3,
558           output wire [7:0] n4,
559           output wire [7:0] n5,
560           output wire [7:0] n6,
561           output wire [7:0] n7,
562           output wire [7:0] n8,
563           output wire [7:0] n9,
564           output wire [7:0] n10,
565           output wire [7:0] n11,
566           output wire [7:0] n12,
567           output wire [7:0] n13,
568           output wire [7:0] n14,
569           output wire [7:0] n15
570           );
571   //this module utilizes the subbytes8 GF(2^8) operation to perform
572   //the SubBytes () transformation of each byte of the current state
573   //See Figure 6 in AES_NIST.FIPS.197.pdf document
574   subbytes8 sbox0 (m0, n0);
575   subbytes8 sbox1 (m1, n1);
576   subbytes8 sbox2 (m2, n2);
577   subbytes8 sbox3 (m3, n3);
578   subbytes8 sbox4 (m4, n4);
579   subbytes8 sbox5 (m5, n5);
```

```
653   xor (b0x2[4], b0[3], b0[7]); buf (b0x2[5], b0[4]);
654   buf (b0x2[6], b0[5]); buf (b0x2[7], b0[6]);
655
656   buf (b1x2[0], b1[7]); xor (b1x2[1], b1[0], b1[7]);
657   buf (b1x2[2], b1[1]); xor (b1x2[3], b1[2], b1[7]);
658   xor (b1x2[4], b1[3], b1[7]); buf (b1x2[5], b1[4]);
659   buf (b1x2[6], b1[5]); buf (b1x2[7], b1[6]);
660
661   buf (b2x2[0], b2[7]); xor (b2x2[1], b2[0], b2[7]);
662   buf (b2x2[2], b2[1]); xor (b2x2[3], b2[2], b2[7]);
663   xor (b2x2[4], b2[3], b2[7]); buf (b2x2[5], b2[4]);
664   buf (b2x2[6], b2[5]); buf (b2x2[7], b2[6]);
665
666   buf (b3x2[0], b3[7]); xor (b3x2[1], b3[0], b3[7]);
667   buf (b3x2[2], b3[1]); xor (b3x2[3], b3[2], b3[7]);
668   xor (b3x2[4], b3[3], b3[7]); buf (b3x2[5], b3[4]);
669   buf (b3x2[6], b3[5]); buf (b3x2[7], b3[6]);
670
671   wire [7:0] x1w0; wire [7:0] x1w1; wire [7:0] x1w2; wire [7:0] x1w3;
672
673   assign x1w0 = b0x2 ^ b0;
674   assign x1w1 = b1x2 ^ b1;
675   assign x1w2 = b2x2 ^ b2;
676   assign x1w3 = b3x2 ^ b3;
677
678   wire [7:0] x2w0l; wire [7:0] x2w1l; wire [7:0] x2w2l; wire [7:0] x2w3l;
679   wire [7:0] x2w0r; wire [7:0] x2w1r; wire [7:0] x2w2r; wire [7:0] x2w3r;
680
681   assign x2w0r = b0x2 ^ x1w3; assign x2w0l = b2 ^ b1;
682   assign x2w1r = b1x2 ^ x1w0; assign x2w1l = b3 ^ b2;
683   assign x2w2r = b2x2 ^ x1w1; assign x2w2l = b3 ^ b0;
684   assign x2w3r = b3x2 ^ x1w2; assign x2w3l = b0 ^ b1;
685
686   assign w0 = x2w0r ^ x2w0l;
687   assign w1 = x2w1r ^ x2w1l;
688   assign w2 = x2w2r ^ x2w2l;
689   assign w3 = x2w3r ^ x2w3l;
690
691   endmodule
692
693   module mixcolumns4 (
694           input wire [7:0] b0,
695           input wire [7:0] b1,
696           input wire [7:0] b2,
697           input wire [7:0] b3,
698           input wire [7:0] b4,
699           input wire [7:0] b5,
700           input wire [7:0] b6,
701           input wire [7:0] b7,
702           input wire [7:0] b8,
703           input wire [7:0] b9,
704           input wire [7:0] b10,
705           input wire [7:0] b11,
706           input wire [7:0] b12,
707           input wire [7:0] b13,
708           input wire [7:0] b14,
709           input wire [7:0] b15,
710           output wire [7:0] w0,
711           output wire [7:0] w1,
712           output wire [7:0] w2,
713           output wire [7:0] w3,
714           output wire [7:0] w4,
715           output wire [7:0] w5,
716           output wire [7:0] w6,
717           output wire [7:0] w7,
718           output wire [7:0] w8,
719           output wire [7:0] w9,
720           output wire [7:0] w10,
721           output wire [7:0] w11,
722           output wire [7:0] w12,
723           output wire [7:0] w13,
724           output wire [7:0] w14,
725           output wire [7:0] w15
```

```
726             );
727     //This module implements the MixColumn () tranformation for the state
728     mixcolumns1 col0 ( b0, b1, b2, b3,
729                        w0, w1, w2, w3);
730     mixcolumns1 col1 ( b4, b5, b6, b7,
731                        w4, w5, w6, w7);
732     mixcolumns1 col2 ( b8, b9, b10, b11,
733                        w8, w9, w10, w11);
734     mixcolumns1 col3 ( b12, b13, b14, b15,
735                        w12, w13, w14, w15);
736     endmodule
737
738     module keyschedule1 (
739             input wire [7:0] r1,
740             input wire [31:0] key00,
741             input wire [31:0] key01,
742             input wire [31:0] key02,
743             input wire [31:0] key03,
744             output wire [31:0] key10,
745             output wire [31:0] key11,
746             output wire [31:0] key12,
747             output wire [31:0] key13
748             );
749     wire [7:0] ks0; wire [7:0] ks1;
750     wire [7:0] ks2; wire [7:0] ks3;
751     wire [31:0] sw0;
752     wire [31:0] xw0;
753     wire [31:0] xw1; wire [31:0] xw2;
754     wire [31:0] xw3;
755     //wire [31:0] r1;
756     //assign r1 = 8'b00000001; // constant ri for round 1
757     subbytes8 s0 (key03[23:16], ks0);
758     subbytes8 s1 (key03[15:8], ks1);
759     subbytes8 s2 (key03[7:0], ks2);
760     subbytes8 s3 (key03[31:24], ks3);
761     assign sw0[31:24] = ks0 ^ r1;
762     assign sw0[23:16] = ks1;
763     assign sw0[15:8] = ks2;
764     assign sw0[7:0] = ks3;
765     //generate the key round output
766     assign xw0 = key00 ^ sw0;
767     assign key10 = xw0;
768     assign xw1 = key01 ^ xw0;
769     assign key11 = xw1;
770     assign xw2 = key02 ^ xw1;
771     assign key12 = xw2;
772     assign xw3 = key03 ^ xw2;
773     assign key13 = xw3;
774
775     endmodule
776
777     module Round_i (
778                     input wire [7:0] ri_s0,
779                     input wire [7:0] ri_s1,
780                     input wire [7:0] ri_s2,
781                     input wire [7:0] ri_s3,
782                     input wire [7:0] ri_s4,
783                     input wire [7:0] ri_s5,
784                     input wire [7:0] ri_s6,
785                     input wire [7:0] ri_s7,
786                     input wire [7:0] ri_s8,
787                     input wire [7:0] ri_s9,
788                     input wire [7:0] ri_s10,
789                     input wire [7:0] ri_s11,
790                     input wire [7:0] ri_s12,
791                     input wire [7:0] ri_s13,
792                     input wire [7:0] ri_s14,
793                     input wire [7:0] ri_s15,
794                     input wire [31:0] ri_k0,
795                     input wire [31:0] ri_k1,
796                     input wire [31:0] ri_k2,
797                     input wire [31:0] ri_k3,
798                     output wire [7:0] ri_w0,
```

```
                    output wire [7:0] ri_w1,
                    output wire [7:0] ri_w2,
                    output wire [7:0] ri_w3,
                    output wire [7:0] ri_w4,
                    output wire [7:0] ri_w5,
                    output wire [7:0] ri_w6,
                    output wire [7:0] ri_w7,
                    output wire [7:0] ri_w8,
                    output wire [7:0] ri_w9,
                    output wire [7:0] ri_w10,
                    output wire [7:0] ri_w11,
                    output wire [7:0] ri_w12,
                    output wire [7:0] ri_w13,
                    output wire [7:0] ri_w14,
                    output wire [7:0] ri_w15
                    );
   //This module combines the transformations for the round as described in
   //Figure 4 algorithm to implement a regular round (see AES_NIST.FIPS.197.pdf
   document)
   wire [7:0] ri_n0; wire [7:0] ri_n1; wire [7:0] ri_n2; wire [7:0] ri_n3;
   wire [7:0] ri_n4; wire [7:0] ri_n5; wire [7:0] ri_n6; wire [7:0] ri_n7;
   wire [7:0] ri_n8; wire [7:0] ri_n9; wire [7:0] ri_n10; wire [7:0] ri_n11;
   wire [7:0] ri_n12; wire [7:0] ri_n13; wire [7:0] ri_n14; wire [7:0] ri_n15;

   subbytes32 Ri_sbox (ri_s0, ri_s1, ri_s2, ri_s3, ri_s4, ri_s5, ri_s6, ri_s7,
                       ri_s8, ri_s9, ri_s10, ri_s11, ri_s12, ri_s13, ri_s14, ri_s15,
                       ri_n0, ri_n1, ri_n2, ri_n3, ri_n4, ri_n5, ri_n6, ri_n7,
                       ri_n8, ri_n9, ri_n10, ri_n11, ri_n12, ri_n13, ri_n14, ri_n15);

   wire [7:0] ri_nn0; wire [7:0] ri_nn1; wire [7:0] ri_nn2; wire [7:0] ri_nn3;
   wire [7:0] ri_nn4; wire [7:0] ri_nn5; wire [7:0] ri_nn6; wire [7:0] ri_nn7;
   wire [7:0] ri_nn8; wire [7:0] ri_nn9; wire [7:0] ri_nn10; wire [7:0] ri_nn11;
   wire [7:0] ri_nn12; wire [7:0] ri_nn13; wire [7:0] ri_nn14; wire [7:0] ri_nn15;

   shiftrows Ri_s_row(ri_n0, ri_n1, ri_n2, ri_n3, ri_n4, ri_n5, ri_n6, ri_n7,
            ri_n8, ri_n9, ri_n10, ri_n11, ri_n12, ri_n13, ri_n14, ri_n15,
            ri_nn0, ri_nn1, ri_nn2, ri_nn3, ri_nn4, ri_nn5, ri_nn6, ri_nn7,
            ri_nn8, ri_nn9, ri_nn10, ri_nn11, ri_nn12, ri_nn13, ri_nn14, ri_nn15);

   wire [7:0] ri_mx0; wire [7:0] ri_mx1; wire [7:0] ri_mx2; wire [7:0] ri_mx3;
   wire [7:0] ri_mx4; wire [7:0] ri_mx5; wire [7:0] ri_mx6; wire [7:0] ri_mx7;
   wire [7:0] ri_mx8; wire [7:0] ri_mx9; wire [7:0] ri_mx10; wire [7:0] ri_mx11;
   wire [7:0] ri_mx12; wire [7:0] ri_mx13; wire [7:0] ri_mx14; wire [7:0] ri_mx15;

   mixcolumns4 Ri_m_col (ri_nn0, ri_nn1, ri_nn2, ri_nn3, ri_nn4, ri_nn5, ri_nn6, ri_nn7,
            ri_nn8, ri_nn9, ri_nn10, ri_nn11, ri_nn12, ri_nn13, ri_nn14, ri_nn15,
            ri_mx0, ri_mx1, ri_mx2, ri_mx3, ri_mx4, ri_mx5, ri_mx6, ri_mx7,
            ri_mx8, ri_mx9, ri_mx10, ri_mx11, ri_mx12, ri_mx13, ri_mx14, ri_mx15);
   //

   AddRoundKey Ri_start (ri_k0[31:24], ri_k0[23:16], ri_k0[15:8], ri_k0[7:0],
                    ri_k1[31:24], ri_k1[23:16], ri_k1[15:8], ri_k1[7:0],
                    ri_k2[31:24], ri_k2[23:16], ri_k2[15:8], ri_k2[7:0],
                    ri_k3[31:24], ri_k3[23:16], ri_k3[15:8], ri_k3[7:0],
                    ri_mx0, ri_mx1, ri_mx2, ri_mx3, ri_mx4, ri_mx5, ri_mx6, ri_mx7,
                    ri_mx8, ri_mx9, ri_mx10, ri_mx11, ri_mx12, ri_mx13, ri_mx14, ri_mx15,
                    ri_w0, ri_w1, ri_w2, ri_w3, ri_w4, ri_w5, ri_w6, ri_w7,
                    ri_w8, ri_w9, ri_w10, ri_w11, ri_w12, ri_w13, ri_w14, ri_w15
                    );

   endmodule
```

Federal Information

Processing Standards Publication 197

November 26, 2001

Announcing the

ADVANCED ENCRYPTION STANDARD (AES)

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235).

1. Name of Standard. Advanced Encryption Standard (AES) (FIPS PUB 197).

2. Category of Standard. Computer Security Standard, Cryptography.

3. Explanation. The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

4. Approving Authority. Secretary of Commerce.

5. Maintenance Agency. Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. Applicability. This standard may be used by Federal departments and agencies when an agency determines that sensitive (unclassified) information (as defined in P. L. 100-235) requires cryptographic protection.

Other FIPS-approved cryptographic algorithms may be used in addition to, or in lieu of, this standard. Federal agencies or departments that use cryptographic devices for protecting classified information can use those devices for protecting sensitive (unclassified) information in lieu of this standard.

In addition, this standard may be adopted and used by non-Federal Government organizations. Such use is encouraged when it provides the desired security for commercial and private organizations.

7. Specifications. Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES) (affixed).

8. Implementations. The algorithm specified in this standard may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors such as the application, the environment, the technology used, etc. The algorithm shall be used in conjunction with a FIPS approved or NIST recommended mode of operation. Object Identifiers (OIDs) and any associated parameters for AES used in these modes are available at the Computer Security Objects Register (CSOR), located at http://csrc.nist.gov/csor/ [2].

Implementations of the algorithm that are tested by an accredited laboratory and validated will be considered as complying with this standard. Since cryptographic security depends on many factors besides the correct implementation of an encryption algorithm, Federal Government employees, and others, should also refer to NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, for additional information and guidance (NIST SP 800-21 is available at http://csrc.nist.gov/publications/).

9. Implementation Schedule. This standard becomes effective on May 26, 2002.

10. Patents. Implementations of the algorithm specified in this standard may be covered by U.S. and foreign patents.

11. Export Control. Certain cryptographic devices and technical data regarding them are subject to Federal export controls. Exports of cryptographic modules implementing this standard and technical data regarding them must comply with these Federal regulations and be licensed by the Bureau of Export Administration of the U.S. Department of Commerce. Applicable Federal government export controls are specified in Title 15, Code of Federal Regulations (CFR) Part 740.17; Title 15, CFR Part 742; and Title 15, CFR Part 774, Category 5, Part 2.

12. Qualifications. NIST will continue to follow developments in the analysis of the AES algorithm. As with its other cryptographic algorithm standards, NIST will formally reevaluate this standard every five years.

Both this standard and possible threats reducing the security provided through the use of this standard will undergo review by NIST as appropriate, taking into account newly available analysis and technology. In addition, the awareness of any breakthrough in technology or any mathematical weakness of the algorithm will cause NIST to reevaluate this standard and provide necessary revisions.

13. Waiver Procedure. Under certain exceptional circumstances, the heads of Federal agencies, or their delegates, may approve waivers to Federal Information Processing Standards (FIPS). The heads of such agencies may redelegate such authority only to a senior official designated pursuant to Section 3506(b) of Title 44, U.S. Code. Waivers shall be granted only when compliance with this standard would

a. adversely affect the accomplishment of the mission of an operator of Federal computer system or

b. cause a major adverse financial impact on the operator that is not offset by government-wide savings.

Agency heads may act upon a written waiver request containing the information detailed above. Agency heads may also act without a written waiver request when they determine that conditions for meeting the standard cannot be met. Agency heads may approve waivers only by a written decision that explains the basis on which the agency head made the required finding(s). A copy of each such decision, with procurement sensitive or classified portions clearly identified, shall be sent to: National Institute of Standards and Technology; ATTN: FIPS Waiver Decision, Information Technology Laboratory, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900.

In addition, notice of each waiver granted and each delegation of authority to approve waivers shall be sent promptly to the Committee on Government Operations of the House of Representatives and the Committee on Government Affairs of the Senate and shall be published promptly in the Federal Register.

When the determination on a waiver applies to the procurement of equipment and/or services, a notice of the waiver determination must be published in the Commerce Business Daily as a part of the notice of solicitation for offers of an acquisition or, if the waiver determination is made after that notice is published, by amendment to such notice.

A copy of the waiver, any supporting documents, the document approving the waiver and any supporting and accompanying documents, with such deletions as the agency is authorized and decides to make under Section 552(b) of Title 5, U.S. Code, shall be part of the procurement documentation and retained by the agency.

14. Where to obtain copies. This publication is available electronically by accessing http://csrc.nist.gov/publications/. A list of other available computer security publications, including ordering information, can be obtained from NIST Publications List 91, which is available at the same web site. Alternatively, copies of NIST computer security publications are available from: National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, VA 22161.

Federal Information

Processing Standards Publication 197

November 26, 2001

Specification for the

ADVANCED ENCRYPTION STANDARD (AES)

Table of Contents

1. INTRODUCTION 5

2. DEFINITIONS 5

2.1 GLOSSARY OF TERMS AND ACRONYMS 5

2.2 ALGORITHM PARAMETERS, SYMBOLS, AND FUNCTIONS 6

3. NOTATION AND CONVENTIONS 7

3.1 INPUTS AND OUTPUTS 7

3.2 BYTES 8

3.3 ARRAYS OF BYTES 8

3.4 THE STATE 9

3.5 THE STATE AS AN ARRAY OF COLUMNS 10

4. MATHEMATICAL PRELIMINARIES 10

4.1 ADDITION 10

4.2 MULTIPLICATION 10

4.2.1 Multiplication by x 11

4.3 POLYNOMIALS WITH COEFFICIENTS IN GF(2^8) 12

5. ALGORITHM SPECIFICATION 13

5.1 CIPHER 14

5.1.1 `SubBytes()` Transformation 15

5.1.2 `ShiftRows()` Transformation 17

5.1.3 `MixColumns()` Transformation 17

5.1.4 `AddRoundKey()` Transformation 18

5.2 KEY EXPANSION 19

5.3 INVERSE CIPHER 20

5.3.1 `InvShiftRows()` Transformation *21*
5.3.2 `InvSubBytes()` Transformation *22*
5.3.3 `InvMixColumns()` Transformation *23*
5.3.4 Inverse of the `AddRoundKey()` Transformation *23*
5.3.5 Equivalent Inverse Cipher *23*

6. IMPLEMENTATION ISSUES 25
6.1 KEY LENGTH REQUIREMENTS 25
6.2 KEYING RESTRICTIONS 26
6.3 PARAMETERIZATION OF KEY LENGTH, BLOCK SIZE, AND ROUND NUMBER 26
6.4 IMPLEMENTATION SUGGESTIONS REGARDING VARIOUS PLATFORMS 26

APPENDIX A - KEY EXPANSION EXAMPLES 27
A.1 EXPANSION OF A 128-BIT CIPHER KEY 27
A.2 EXPANSION OF A 192-BIT CIPHER KEY 28
A.3 EXPANSION OF A 256-BIT CIPHER KEY 30

APPENDIX B – CIPHER EXAMPLE 33

APPENDIX C – EXAMPLE VECTORS 35
C.1 AES-128 (N_K=4, N_R=10) 35
C.2 AES-192 (N_K=6, N_R=12) 38
C.3 AES-256 (N_K=8, N_R=14) 42

APPENDIX D - REFERENCES 47

Table of Figures

Figure 1. Hexadecimal representation of bit patterns. 8

Figure 2. Indices for Bytes and Bits. 9

Figure 3. State array input and output. 9

Figure 4. Key-Block-Round Combinations. 14

Figure 5. Pseudo Code for the Cipher. 15

Figure 6. `SubBytes()` applies the S-box to each byte of the State. 16

Figure 7. S-box: substitution values for the byte `xy` (in hexadecimal format). 16

Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State 17

Figure 9. `MixColumns()` operates on the State column-by-column. 18

Figure 10. `AddRoundKey()` XORs each column of the State with a word from the key schedule. 19

Figure 11. Pseudo Code for Key Expansion. 20

Figure 12. Pseudo Code for the Inverse Cipher. 21

Figure 13. `InvShiftRows()` cyclically shifts the last three rows in the State. 22

Figure 14. Inverse S-box: substitution values for the byte `xy` (in hexadecimal format). 22

Figure 15. Pseudo Code for the Equivalent Inverse Cipher. 25

1. Introduction

This standard specifies the **Rijndael** algorithm ([3] and [4]), a symmetric block cipher that can process **data blocks** of **128 bits**, using cipher **keys** with lengths of **128**, **192**, and **256 bits**. Rijndael was designed to handle additional block sizes and key lengths, however they are not adopted in this standard.

Throughout the remainder of this standard, the algorithm specified herein will be referred to as "the AES algorithm." The algorithm may be used with the three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256".

This specification includes the following sections:

2. Definitions of terms, acronyms, and algorithm parameters, symbols, and functions;
3. Notation and conventions used in the algorithm specification, including the ordering and numbering of bits, bytes, and words;
4. Mathematical properties that are useful in understanding the algorithm;
5. Algorithm specification, covering the key expansion, encryption, and decryption routines;
6. Implementation issues, such as key length support, keying restrictions, and additional block/key/round sizes.

The standard concludes with several appendices that include step-by-step examples for Key Expansion and the Cipher, example vectors for the Cipher and Inverse Cipher, and a list of references.

2. Definitions

2.1 Glossary of Terms and Acronyms

The following definitions are used throughout this standard:

AES	Advanced Encryption Standard
Affine Transformation	A transformation consisting of multiplication by a matrix followed by the addition of a vector.
Array	An enumerated collection of identical entities (e.g., an array of bytes).
Bit	A binary digit having a value of 0 or 1.
Block	Sequence of binary bits that comprise the input, output, State, and Round Key. The length of a sequence is the number of bits it contains. Blocks are also interpreted as arrays of bytes.
Byte	A group of eight bits that is treated either as a single entity or as an array of 8 individual bits.

Cipher	Series of transformations that converts plaintext to ciphertext using the Cipher Key.
Cipher Key	Secret, cryptographic key that is used by the Key Expansion routine to generate a set of Round Keys; can be pictured as a rectangular array of bytes, having four rows and ***Nk*** columns.
Ciphertext	Data output from the Cipher or input to the Inverse Cipher.
Inverse Cipher	Series of transformations that converts ciphertext to plaintext using the Cipher Key.
Key Expansion	Routine used to generate a series of Round Keys from the Cipher Key.
Plaintext	Data input to the Cipher or output from the Inverse Cipher.
Rijndael	Cryptographic algorithm specified in this Advanced Encryption Standard (AES).
Round Key	Round keys are values derived from the Cipher Key using the Key Expansion routine; they are applied to the State in the Cipher and Inverse Cipher.
State	Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and ***Nb*** columns.
S-box	Non-linear substitution table used in several byte substitution transformations and in the Key Expansion routine to perform a one-for-one substitution of a byte value.
Word	A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.

2.2 Algorithm Parameters, Symbols, and Functions

The following algorithm parameters, symbols, and functions are used throughout this standard:

`AddRoundKey()`	Transformation in the Cipher and Inverse Cipher in which a Round Key is added to the State using an XOR operation. The length of a Round Key equals the size of the State (i.e., for *Nb* = 4, the Round Key length equals 128 bits/16 bytes).
`InvMixColumns()`	Transformation in the Inverse Cipher that is the inverse of **`MixColumns()`**.
`InvShiftRows()`	Transformation in the Inverse Cipher that is the inverse of **`ShiftRows()`**.
`InvSubBytes()`	Transformation in the Inverse Cipher that is the inverse of **`SubBytes()`**.
K	Cipher Key.

`MixColumns()`	Transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns.
Nb	Number of columns (32-bit words) comprising the State. For this standard, ***Nb*** = 4. (Also see Sec. 6.3.)
Nk	Number of 32-bit words comprising the Cipher Key. For this standard, ***Nk*** = 4, 6, or 8. (Also see Sec. 6.3.)
Nr	Number of rounds, which is a function of ***Nk*** and ***Nb*** (which is fixed). For this standard, ***Nr*** = 10, 12, or 14. (Also see Sec. 6.3.)
Rcon[]	The round constant word array.
`RotWord()`	Function used in the Key Expansion routine that takes a four-byte word and performs a cyclic permutation.
`ShiftRows()`	Transformation in the Cipher that processes the State by cyclically shifting the last three rows of the State by different offsets.
`SubBytes()`	Transformation in the Cipher that processes the State using a non-linear byte substitution table (S-box) that operates on each of the State bytes independently.
`SubWord()`	Function used in the Key Expansion routine that takes a four-byte input word and applies an S-box to each of the four bytes to produce an output word.
XOR	Exclusive-OR operation.
$\oplus$	Exclusive-OR operation.
$\otimes$	Multiplication of two polynomials (each with degree < 4) modulo $x^4 + 1$.
$\bullet$	Finite field multiplication.

3. Notation and Conventions

3.1 Inputs and Outputs

The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits** (digits with values of 0 or 1). These sequences will sometimes be referred to as **blocks** and the number of bits they contain will be referred to as their length. The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in one of the ranges $0 \le i < 128$, $0 \le i < 192$ or $0 \le i < 256$ depending on the block length and key length (specified above).

3.2 Bytes

The basic unit for processing in the AES algorithm is a **byte,** a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences described in Sec. 3.1 are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes (see Sec. 3.3). For an input, output or Cipher Key denoted by a, the bytes in the resulting array will be referenced using one of the two forms, a_n or $a[n]$, where n will be in one of the following ranges:

Key length = 128 bits, $0 \le n < 16$; Block length = 128 bits, $0 \le n < 16$;

Key length = 192 bits, $0 \le n < 24$;

Key length = 256 bits, $0 \le n < 32$.

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i \ . \qquad (3.1)$$

For example, {`01100011`} identifies the specific finite field element $x^6 + x^5 + x + 1$.

It is also convenient to denote byte values using hexadecimal notation with each of two groups of four bits being denoted by a single character as in Fig. 1.

Bit Pattern	Character
0000	0
0001	1
0010	2
0011	3

Bit Pattern	Character
0100	4
0101	5
0110	6
0111	7

Bit Pattern	Character
1000	8
1001	9
1010	a
1011	b

Bit Pattern	Character
1100	c
1101	d
1110	e
1111	f

Figure 1. Hexadecimal representation of bit patterns.

Hence the element {`01100011`} can be represented as {`63`}, where the character denoting the four-bit group containing the higher numbered bits is again to the left.

Some finite field operations involve one additional bit (b_8) to the left of an 8-bit byte. Where this extra bit is present, it will appear as '{01}' immediately preceding the 8-bit byte; for example, a 9-bit sequence will be presented as {01}{`1b`}.

3.3 Arrays of Bytes

Arrays of bytes will be represented in the following form:

$$a_0 a_1 a_2 \ldots a_{15}$$

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence

$$input_0\ input_1\ input_2 \ldots input_{126}\ input_{127}$$

as follows:

$$a_0 = \{input_0, input_1, \ldots, input_7\};$$
$$a_1 = \{input_8, input_9, \ldots, input_{15}\};$$
$$\vdots$$
$$a_{15} = \{input_{120}, input_{121}, \ldots, input_{127}\}.$$

The pattern can be extended to longer sequences (i.e., for 192- and 256-bit keys), so that, in general,

$$a_n = \{input_{8n}, input_{8n+1}, \ldots, input_{8n+7}\}. \qquad (3.2)$$

Taking Sections 3.2 and 3.3 together, Fig. 2 shows how bits within each byte are numbered.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	…
Byte number	0								1								2								…
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	…

Figure 2. Indices for Bytes and Bits.

3.4 The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the **State**. The State consists of four rows of bytes, each containing ***Nb*** bytes, where ***Nb*** is the block length divided by 32. In the State array denoted by the symbol *s*, each individual byte has two indices, with its row number r in the range $0 \le r < 4$ and its column number c in the range $0 \le c <$ ***Nb***. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or $s[r,c]$. For this standard, ***Nb***=4, i.e., $0 \le c < 4$ (also see Sec. 6.3).

At the start of the Cipher and Inverse Cipher described in Sec. 5, the input – the array of bytes in_0, in_1, … in_{15} – is copied into the State array as illustrated in Fig. 3. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes out_0, out_1, … out_{15}.

input bytes

in_0	in_4	in_8	in_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

→

State array

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

→

output bytes

out_0	out_4	out_8	out_{12}
out_1	out_5	out_9	out_{13}
out_2	out_6	out_{10}	out_{14}
out_3	out_7	out_{11}	out_{15}

Figure 3. State array input and output.

Hence, at the beginning of the Cipher or Inverse Cipher, the input array, *in*, is copied to the State array according to the scheme:

$$s[r, c] = in[r + 4c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb, \qquad (3.3)$$

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array *out* as follows:

$$out[r + 4c] = s[r, c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb. \qquad (3.4)$$

3.5 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit **words**, where the row number r provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), $w_0 ... w_3$, where the column number c provides an index into this array. Hence, for the example in Fig. 3, the State can be considered as an array of four words, as follows:

$$w_0 = s_{0,0}\, s_{1,0}\, s_{2,0}\, s_{3,0} \qquad w_2 = s_{0,2}\, s_{1,2}\, s_{2,2}\, s_{3,2}$$

$$w_1 = s_{0,1}\, s_{1,1}\, s_{2,1}\, s_{3,1} \qquad w_3 = s_{0,3}\, s_{1,3}\, s_{2,3}\, s_{3,3}\,. \qquad (3.5)$$

4. Mathematical Preliminaries

All bytes in the AES algorithm are interpreted as finite field elements using the notation introduced in Sec. 3.2. Finite field elements can be added and multiplied, but these operations are different from those used for numbers. The following subsections introduce the basic mathematical concepts needed for Sec. 5.

4.1 Addition

The addition of two elements in a finite field is achieved by "adding" the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by $\oplus$) - i.e., modulo 2 - so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, and $0 \oplus 0 = 0$. Consequently, subtraction of polynomials is identical to addition of polynomials.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ and $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, the sum is $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, where each $c_i = a_i \oplus b_i$ (i.e., $c_7 = a_7 \oplus b_7$, $c_6 = a_6 \oplus b_6$, ...$c_0 = a_0 \oplus b_0$).

For example, the following expressions are equivalent to one another:

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation);

`{01010111} ⊕ {10000011} = {11010100}` (binary notation);

`{57} ⊕ {83} = {d4}` (hexadecimal notation).

4.2 Multiplication

In the polynomial representation, multiplication in GF(2^8) (denoted by $\bullet$) corresponds with the multiplication of polynomials modulo an **irreducible polynomial** of degree 8. A polynomial is irreducible if its only divisors are one and itself. **For the AES algorithm, this <u>irreducible</u> polynomial is**

$$m(x) = x^8 + x^4 + x^3 + x + 1, \qquad (4.1)$$

or {01}{1b} in hexadecimal notation.

For example, {57} • {83} = {c1}, because

$$
\begin{aligned}
(x^6 + x^4 + x^2 + x + 1)\ (x^7 + x + 1) \quad &= \quad x^{13} + x^{11} + x^9 + x^8 + x^7 + \\
& \quad\ \ x^7 + x^5 + x^3 + x^2 + x + \\
& \quad\ \ x^6 + x^4 + x^2 + x + 1 \\
&= \quad x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1
\end{aligned}
$$

and

$$
\begin{aligned}
x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \ \texttt{modulo}\ (x^8 + x^4 + x^3 + x + 1) \\
= \quad x^7 + x^6 + 1.
\end{aligned}
$$

The modular reduction by $m(x)$ ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

The multiplication defined above is associative, and the element {01} is the multiplicative identity. For any non-zero binary polynomial $b(x)$ of degree less than 8, the multiplicative inverse of $b(x)$, denoted $b^{-1}(x)$, can be found as follows: the extended Euclidean algorithm [7] is used to compute polynomials $a(x)$ and $c(x)$ such that

$$b(x)a(x) + m(x)c(x) = 1. \tag{4.2}$$

Hence, $a(x) \bullet b(x) \bmod m(x) = 1$, which means

$$b^{-1}(x) = a(x) \bmod m(x). \tag{4.3}$$

Moreover, for any $a(x)$, $b(x)$ and $c(x)$ in the field, it holds that

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x).$$

It follows that the set of 256 possible byte values, with XOR used as addition and the multiplication defined as above, has the structure of the finite field GF(2^8).

4.2.1 Multiplication by *x*

Multiplying the binary polynomial defined in equation (3.1) with the polynomial x results in

$$b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x. \tag{4.4}$$

The result $x \bullet b(x)$ is obtained by reducing the above result modulo $m(x)$, as defined in equation (4.1). If $b_7 = 0$, the result is already in reduced form. If $b_7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial $m(x)$. It follows that multiplication by x (i.e., {00000010} or {02}) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with {1b}. This operation on bytes is denoted by `xtime()`. Multiplication by higher powers of x can be implemented by repeated application of `xtime()`. By adding intermediate results, multiplication by any constant can be implemented.

For example, {57} • {13} = {fe} because

$$\{57\} \bullet \{02\} = \texttt{xtime}(\{57\}) = \{ae\}$$

$$\{57\} \bullet \{04\} = \texttt{xtime}(\{ae\}) = \{47\}$$

$$\{57\} \bullet \{08\} = \texttt{xtime}(\{47\}) = \{8e\}$$

$$\{57\} \bullet \{10\} = \texttt{xtime}(\{8e\}) = \{07\},$$

thus,

$$\begin{aligned} \{57\} \bullet \{13\} &= \{57\} \bullet (\{01\} \oplus \{02\} \oplus \{10\}) \\ &= \{57\} \oplus \{ae\} \oplus \{07\} \\ &= \{fe\}. \end{aligned}$$

4.3 Polynomials with Coefficients in GF(2^8)

Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \tag{4.5}$$

which will be denoted as a word in the form $[a_0, a_1, a_2, a_3]$. Note that the polynomials in this section behave somewhat differently than the polynomials used in the definition of finite field elements, even though both types of polynomials use the same indeterminate, x. The coefficients in this section are themselves finite field elements, i.e., bytes, instead of bits; also, the multiplication of four-term polynomials uses a different reduction polynomial, defined below. The distinction should always be clear from the context.

To illustrate the addition and multiplication operations, let

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \tag{4.6}$$

define a second four-term polynomial. Addition is performed by adding the finite field coefficients of like powers of x. This addition corresponds to an XOR operation between the corresponding bytes in each of the words – in other words, the XOR of the complete word values.

Thus, using the equations of (4.5) and (4.6),

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0) \tag{4.7}$$

Multiplication is achieved in two steps. In the first step, the polynomial product $c(x) = a(x) \bullet b(x)$ is algebraically expanded, and like powers are collected to give

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \tag{4.8}$$

where

$$\begin{aligned} c_0 &= a_0 \bullet b_0 & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\ c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 & c_5 &= a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\ c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 & c_6 &= a_3 \bullet b_3 \end{aligned} \tag{4.9}$$

$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3 .$$

The result, $c(x)$, does not represent a four-byte word. Therefore, the second step of the multiplication is to reduce $c(x)$ modulo a polynomial of degree 4; the result can be reduced to a polynomial of degree less than 4. **For the AES algorithm, this is accomplished with the polynomial $x^4 + 1$**, so that

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4} . \tag{4.10}$$

The modular product of $a(x)$ and $b(x)$, denoted by $a(x) \otimes b(x)$, is given by the four-term polynomial $d(x)$, defined as follows:

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0 \tag{4.11}$$

with

$$\begin{aligned} d_0 &= (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3) \\ d_1 &= (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3) \\ d_2 &= (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3) \\ d_3 &= (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3) \end{aligned} \tag{4.12}$$

When $a(x)$ is a fixed polynomial, the operation defined in equation (4.11) can be written in matrix form as:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{4.13}$$

Because $x^4 + 1$ is not an irreducible polynomial over GF(2^8), multiplication by a fixed four-term polynomial is not necessarily invertible. However, the AES algorithm specifies a fixed four-term polynomial that *does* have an inverse (see Sec. 5.1.3 and Sec. 5.3.3):

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{4.14}$$

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \tag{4.15}$$

Another polynomial used in the AES algorithm (see the **`RotWord()`** function in Sec. 5.2) has $a_0 = a_1 = a_2 = \{00\}$ and $a_3 = \{01\}$, which is the polynomial x^3. Inspection of equation (4.13) above will show that its effect is to form the output word by rotating bytes in the input word. This means that $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.

5. Algorithm Specification

For the AES algorithm**, the length of the input block, the output block and the State is 128 bits.** This is represented by ***Nb*** = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES algorithm**, the length of the Cipher Key, *K*, is 128, 192, or 256 bits.** The key length is represented by ***Nk*** = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by ***Nr***, where ***Nr*** = 10 when ***Nk*** = 4, ***Nr*** = 12 when ***Nk*** = 6, and ***Nr*** = 14 when ***Nk*** = 8.

The only Key-Block-Round combinations that conform to this standard are given in Fig. 4. For implementation issues relating to the key length, block size and number of rounds, see Sec. 6.3.

	Key Length *(Nk words)*	**Block Size** *(Nb words)*	**Number of Rounds** *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 4. Key-Block-Round Combinations.

For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations: 1) byte substitution using a substitution table (S-box), 2) shifting rows of the State array by different offsets, 3) mixing the data within each column of the State array, and 4) adding a Round Key to the State. These transformations (and their inverses) are described in Sec. 5.1.1-5.1.4 and 5.3.1-5.3.4.

The Cipher and Inverse Cipher are described in Sec. 5.1 and Sec. 5.3, respectively, while the Key Schedule is described in Sec. 5.2.

5.1 Cipher

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec. 3.4. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first $Nr - 1$ rounds. The final State is then copied to the output as described in Sec. 3.4.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion routine described in Sec. 5.2.

The Cipher is described in the pseudo code in Fig. 5. The individual transformations - **`SubBytes()`**, **`ShiftRows()`**, **`MixColumns()`**, and **`AddRoundKey()`** – process the State and are described in the following subsections. In Fig. 5, the array **`w[]`** contains the key schedule, which is described in Sec. 5.2.

As shown in Fig. 5, all ***Nr*** rounds are identical with the exception of the final round, which does not include the **`MixColumns()`** transformation.

Appendix B presents an example of the Cipher, showing values for the State array at the beginning of each round and after the application of each of the four transformations described in the following sections.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])            // See Sec. 5.1.4

   for round = 1 step 1 to Nr–1
      SubBytes(state)                        // See Sec. 5.1.1
      ShiftRows(state)                       // See Sec. 5.1.2
      MixColumns(state)                      // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Figure 5. Pseudo Code for the Cipher.[1]

5.1.1 `SubBytes()` Transformation

The **`SubBytes()`** transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box (Fig. 7), which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field GF(2^8), described in Sec. 4.2; the element {`00`} is mapped to itself.
2. Apply the following affine transformation (over GF(2)):

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \qquad (5.1)$$

   for $0 \le i < 8$, where b_i is the ith bit of the byte, and c_i is the ith bit of a byte c with the value {`63`} or {`01100011`}. Here and elsewhere, a prime on a variable (e.g., b') indicates that the variable is to be updated with the value on the right.

   In matrix form, the affine transformation element of the S-box can be expressed as:

[1] The various transformations (e.g., **`SubBytes()`**, **`ShiftRows()`**, etc.) act upon the State array that is addressed by the '`state`' pointer. **`AddRoundKey()`** uses an additional pointer to address the Round Key.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{5.2}$$

Figure 6 illustrates the effect of the **`SubBytes()`** transformation on the State.

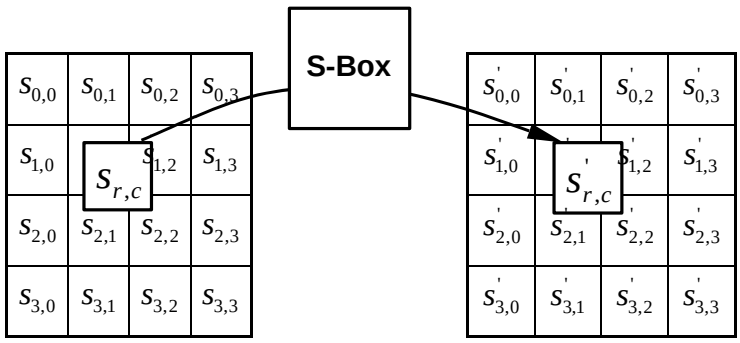


Figure 6. `SubBytes()` applies the S-box to each byte of the State.

The S-box used in the **`SubBytes()`** transformation is presented in hexadecimal form in Fig. 7.

For example, if $s_{1,1} = \{53\}$, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Fig. 7. This would result in $s'_{1,1}$ having a value of {`ed`}.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 7. S-box: substitution values for the byte xy (in hexadecimal format).

5.1.2 `ShiftRows()` Transformation

In the **`ShiftRows()`** transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted.

Specifically, the **`ShiftRows()`** transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb)) \bmod Nb} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < \boldsymbol{Nb}, \tag{5.3}$$

where the shift value $shift(r,Nb)$ depends on the row number, r, as follows (recall that $\boldsymbol{Nb} = 4$):

$$shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3. \tag{5.4}$$

This has the effect of moving bytes to "lower" positions in the row (i.e., lower values of c in a given row), while the "lowest" bytes wrap around into the "top" of the row (i.e., higher values of c in a given row).

Figure 8 illustrates the **`ShiftRows()`** transformation.

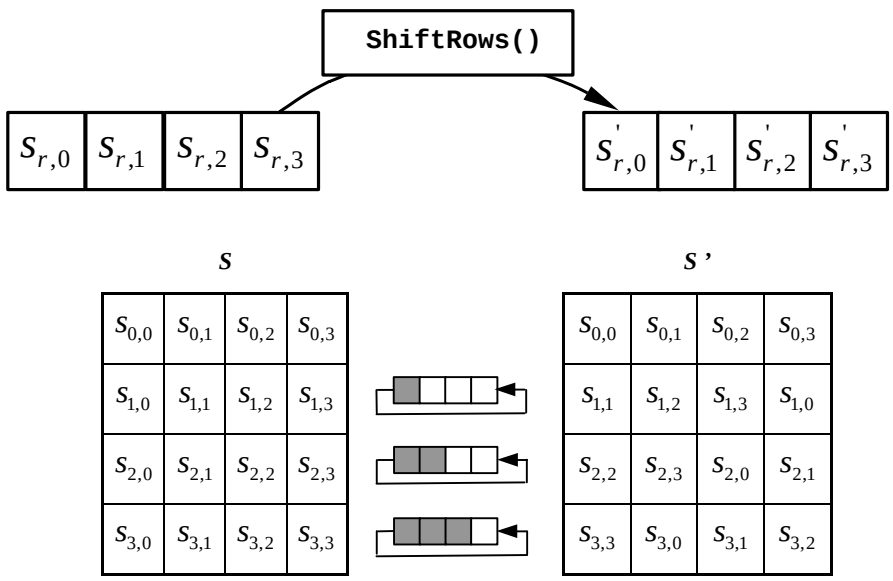


Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State.

5.1.3 `MixColumns()` Transformation

The **`MixColumns()`** transformation operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}. \tag{5.5}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$s'(x) = a(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}. \tag{5.6}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

Figure 9 illustrates the **`MixColumns()`** transformation.

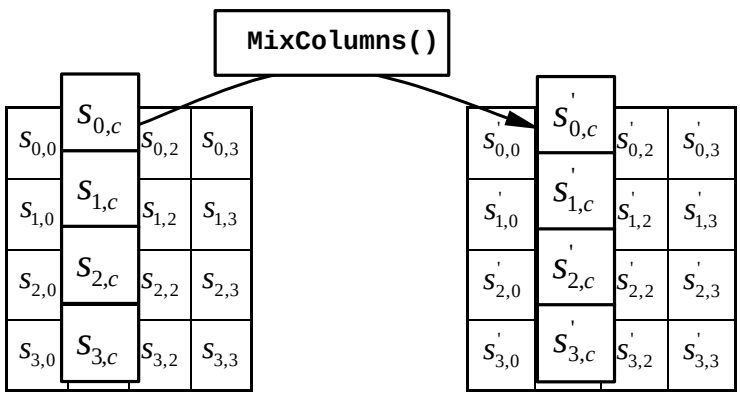


Figure 9. `MixColumns()` operates on the State column-by-column.

5.1.4 `AddRoundKey()` Transformation

In the **`AddRoundKey()`** transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of ***Nb*** words from the key schedule (described in Sec. 5.2). Those ***Nb*** words are each added into the columns of the State, such that

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad \text{for } 0 \le c < \boldsymbol{Nb}, \tag{5.7}$$

where $[w_i]$ are the key schedule words described in Sec. 5.2, and *round* is a value in the range $0 \le round \le \boldsymbol{Nr}$. In the Cipher, the initial Round Key addition occurs when *round* = 0, prior to the first application of the round function (see Fig. 5). The application of the **`AddRoundKey()`** transformation to the ***Nr*** rounds of the Cipher occurs when $1 \le round \le \boldsymbol{Nr}$.

The action of this transformation is illustrated in Fig. 10, where $l = round * \boldsymbol{Nb}$. The byte address within words of the key schedule was described in Sec. 3.1.

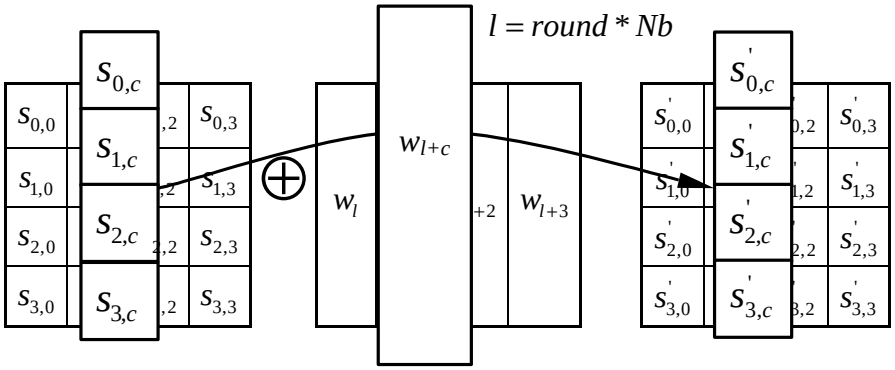


Figure 10. `AddRoundKey()` XORs each column of the State with a word from the key schedule.

5.2 Key Expansion

The AES algorithm takes the Cipher Key, ***K***, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of ***Nb*** (***Nr*** + 1) words: the algorithm requires an initial set of ***Nb*** words, and each of the ***Nr*** rounds requires ***Nb*** words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted [w_i], with i in the range $0 \le i < \boldsymbol{Nb}(\boldsymbol{Nr} + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Fig. 11.

`SubWord()` is a function that takes a four-byte input word and applies the S-box (Sec. 5.1.1, Fig. 7) to each of the four bytes to produce an output word. The function **`RotWord()`** takes a word $[a_0,a_1,a_2,a_3]$ as input, performs a cyclic permutation, and returns the word $[a_1,a_2,a_3,a_0]$. The round constant word array, **`Rcon[i]`**, contains the values given by [x^{i-1},{`00`},{`00`},{`00`}], with x^{i-1} being powers of x (x is denoted as {`02`}) in the field GF(2^8), as discussed in Sec. 4.2 (note that i starts at 1, not 0).

From Fig. 11, it can be seen that the first ***Nk*** words of the expanded key are filled with the Cipher Key. Every following word, **`w[i]`**, is equal to the XOR of the previous word, **`w[i-1]`**, and the word ***Nk*** positions earlier, **`w[i-`*Nk*`]`**. For words in positions that are a multiple of ***Nk***, a transformation is applied to **`w[i-1]`** prior to the XOR, followed by an XOR with a round constant, **`Rcon[i]`**. This transformation consists of a cyclic shift of the bytes in a word (**`RotWord()`**), followed by the application of a table lookup to all four bytes of the word (**`SubWord()`**).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys (***Nk*** = 8) is slightly different than for 128- and 192-bit Cipher Keys. If ***Nk*** = 8 and `i-4` is a multiple of ***Nk***, then **`SubWord()`** is applied to **`w[i-1]`** prior to the XOR.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end

Note that Nk=4, 6, and 8 do not all have to be implemented;
they are all included in the conditional statement above for
conciseness.   Specific implementation requirements for the
Cipher Key are presented in Sec. 6.1.
```

Figure 11. Pseudo Code for Key Expansion.[2]

Appendix A presents examples of the Key Expansion.

5.3 Inverse Cipher

The Cipher transformations in Sec. 5.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - **`InvShiftRows()`**, **`InvSubBytes()`**,**`InvMixColumns()`**, and **`AddRoundKey()`** – process the State and are described in the following subsections.

The Inverse Cipher is described in the pseudo code in Fig. 12. In Fig. 12, the array **`w[]`** contains the key schedule, which was described previously in Sec. 5.2.

[2] The functions **`SubWord()`** and **`RotWord()`** return a result that is a transformation of the function input, whereas the transformations in the Cipher and Inverse Cipher (e.g., **`ShiftRows()`**, **`SubBytes()`**, etc.) transform the State array that is addressed by the '`state`' pointer.

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)                    // See Sec. 5.3.1
      InvSubBytes(state)                     // See Sec. 5.3.2
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)                   // See Sec. 5.3.3
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

Figure 12. Pseudo Code for the Inverse Cipher.[3]

5.3.1 `InvShiftRows()` Transformation

`InvShiftRows()` is the inverse of the **`ShiftRows()`** transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. The bottom three rows are cyclically shifted by $\boldsymbol{Nb} - shift(r, Nb)$ bytes, where the shift value *shift(r,Nb)* depends on the row number, and is given in equation (5.4) (see Sec. 5.1.2).

Specifically, the **`InvShiftRows()`** transformation proceeds as follows:

$$s'_{r,(c+shift(r,Nb)) \bmod Nb} = s_{r,c} \quad \text{for } 0 < r < 4 \quad \text{and} \quad 0 \le c < \boldsymbol{Nb} \tag{5.8}$$

Figure 13 illustrates the **`InvShiftRows()`** transformation.

[3] The various transformations (e.g., **`InvSubBytes()`**, **`InvShiftRows()`**, etc.) act upon the State array that is addressed by the '`state`' pointer. **`AddRoundKey()`** uses an additional pointer to address the Round Key.

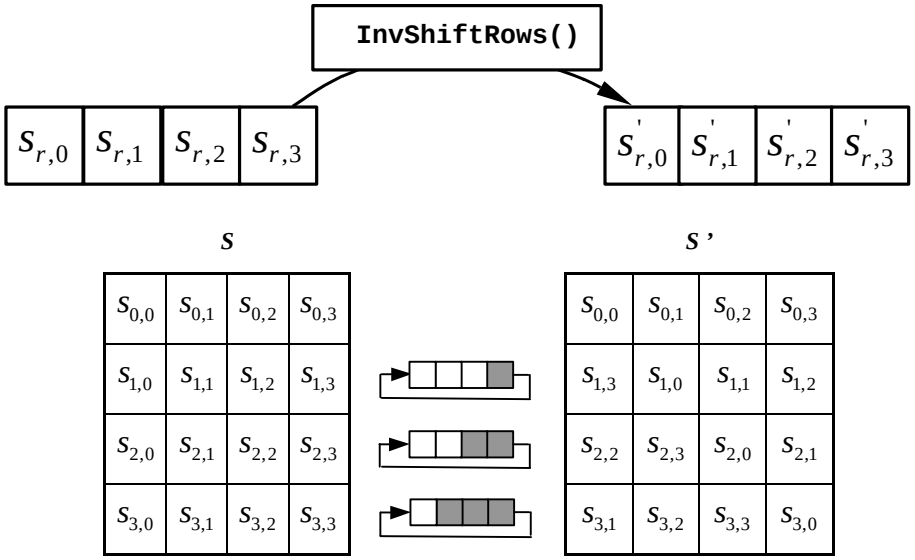


Figure 13. `InvShiftRows()` cyclically shifts the last three rows in the State.

5.3.2 `InvSubBytes()` Transformation

`InvSubBytes()` is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (5.1) followed by taking the multiplicative inverse in GF(2^8).

The inverse S-box used in the **`InvSubBytes()`** transformation is presented in Fig. 14:

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 14. Inverse S-box: substitution values for the byte xy (in hexadecimal format).

5.3.3 `InvMixColumns()` Transformation

`InvMixColumns()` is the inverse of the **`MixColumns()`** transformation. **`InvMixColumns()`** operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

$$a^{-1}(x) = \{\texttt{0b}\}x^3 + \{\texttt{0d}\}x^2 + \{\texttt{09}\}x + \{\texttt{0e}\}. \tag{5.9}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$s'(x) = a^{-1}(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}. \tag{5.10}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{\texttt{0e}\} \bullet s_{0,c}) \oplus (\{\texttt{0b}\} \bullet s_{1,c}) \oplus (\{\texttt{0d}\} \bullet s_{2,c}) \oplus (\{\texttt{09}\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{\texttt{09}\} \bullet s_{0,c}) \oplus (\{\texttt{0e}\} \bullet s_{1,c}) \oplus (\{\texttt{0b}\} \bullet s_{2,c}) \oplus (\{\texttt{0d}\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{\texttt{0d}\} \bullet s_{0,c}) \oplus (\{\texttt{09}\} \bullet s_{1,c}) \oplus (\{\texttt{0e}\} \bullet s_{2,c}) \oplus (\{\texttt{0b}\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{\texttt{0b}\} \bullet s_{0,c}) \oplus (\{\texttt{0d}\} \bullet s_{1,c}) \oplus (\{\texttt{09}\} \bullet s_{2,c}) \oplus (\{\texttt{0e}\} \bullet s_{3,c})$$

5.3.4 Inverse of the `AddRoundKey()` Transformation

`AddRoundKey()`, which was described in Sec. 5.1.4, is its own inverse, since it only involves an application of the XOR operation.

5.3.5 Equivalent Inverse Cipher

In the straightforward Inverse Cipher presented in Sec. 5.3 and Fig. 12, the sequence of the transformations differs from that of the Cipher, while the form of the key schedules for encryption and decryption remains the same. However, several properties of the AES algorithm allow for an Equivalent Inverse Cipher that has the same sequence of transformations as the Cipher (with the transformations replaced by their inverses). This is accomplished with a change in the key schedule.

The two properties that allow for this Equivalent Inverse Cipher are as follows:

1. The **`SubBytes()`** and **`ShiftRows()`** transformations commute; that is, a **`SubBytes()`** transformation immediately followed by a **`ShiftRows()`** transformation is equivalent to a **`ShiftRows()`** transformation immediately followed buy a **`SubBytes()`** transformation. The same is true for their inverses, **`InvSubBytes()`** and **`InvShiftRows.`**

2. The column mixing operations - **MixColumns()** and **InvMixColumns()** - are linear with respect to the column input, which means

```
InvMixColumns(state XOR Round Key) =
                InvMixColumns(state) XOR InvMixColumns(Round Key).
```

These properties allow the order of **InvSubBytes()** and **InvShiftRows()** transformations to be reversed. The order of the **AddRoundKey()** and **InvMixColumns()** transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the **InvMixColumns()** transformation.

The equivalent inverse cipher is defined by reversing the order of the **InvSubBytes()** and **InvShiftRows()** transformations shown in Fig. 12, and by reversing the order of the **AddRoundKey()** and **InvMixColumns()** transformations used in the "round loop" after first modifying the decryption key schedule for *round* = 1 to *Nr*-1 using the **InvMixColumns()** transformation. The first and last *Nb* words of the decryption key schedule shall *not* be modified in this manner.

Given these changes, the resulting Equivalent Inverse Cipher offers a more efficient structure than the Inverse Cipher described in Sec. 5.3 and Fig. 12. Pseudo code for the Equivalent Inverse Cipher appears in Fig. 15. (The word array **dw[]** contains the modified decryption key schedule. The modification to the Key Expansion routine is also provided in Fig. 15.)

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])

   for round = Nr-1 step -1 downto 1
      InvSubBytes(state)
      InvShiftRows(state)
      InvMixColumns(state)
      AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
   end for

   InvSubBytes(state)
   InvShiftRows(state)
   AddRoundKey(state, dw[0, Nb-1])

   out = state
end


For the Equivalent Inverse Cipher, the following pseudo code is added at
the end of the Key Expansion routine (Sec. 5.2):
   for i = 0 step 1 to (Nr+1)*Nb-1
      dw[i] = w[i]
   end for


   for round = 1 step 1 to Nr-1
      InvMixColumns(dw[round*Nb, (round+1)*Nb-1])     //   note   change   of
type
   end for

Note that, since InvMixColumns operates on a two-dimensional array of bytes
while the Round Keys are held in an array of words, the call to
InvMixColumns in this code sequence involves a change of type (i.e. the
input to InvMixColumns() is normally the State array, which is considered
to be a two-dimensional array of bytes, whereas the input here is a Round
Key computed as a one-dimensional array of words).
```

Figure 15. Pseudo Code for the Equivalent Inverse Cipher.

6. Implementation Issues

6.1 Key Length Requirements

An implementation of the AES algorithm shall support *at least one* of the three key lengths specified in Sec. 5: 128, 192, or 256 bits (i.e., ***Nk*** = 4, 6, or 8, respectively). Implementations

may optionally support two or three key lengths, which may promote the interoperability of algorithm implementations.

6.2 Keying Restrictions

No weak or semi-weak keys have been identified for the AES algorithm, and there is no restriction on key selection.

6.3 Parameterization of Key Length, Block Size, and Round Number

This standard explicitly defines the allowed values for the key length (***Nk***), block size (***Nb***), and number of rounds (***Nr***) – see Fig. 4. However, future reaffirmations of this standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind.

6.4 Implementation Suggestions Regarding Various Platforms

Implementation variations are possible that may, in many cases, offer performance or other advantages. Given the same input key and data (plaintext or ciphertext), any implementation that produces the same output (ciphertext or plaintext) as the algorithm specified in this standard is an acceptable implementation of the AES.

Reference [3] and other papers located at Ref. [1] include suggestions on how to efficiently implement the AES algorithm on a variety of platforms.

Appendix A - Key Expansion Examples

This appendix shows the development of the key schedule for various key sizes. Note that multi-byte values are presented using the notation described in Sec. 3. The intermediate values produced during the development of the key schedule (see Sec. 5.2) are given in the following table (all values are in hexadecimal format, with the exception of the index column (i)).

A.1 Expansion of a 128-bit Cipher Key

This section contains the key expansion of the following cipher key:

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

for ***Nk*** = 4, which results in

w_0 = **2b7e1516** w_1 = **28aed2a6** w_2 = **abf71588** w_3 = **09cf4f3c**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc

24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

A.2 Expansion of a 192-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =      8e 73 b0 f7 da 0e 64 52 c8 10 f3 2b
                  80 90 79 e5 62 f8 ea d2 52 2c 6b 7b
```

for ***Nk*** = 6, which results in

w_0 = **8e73b0f7** w_1 = **da0e6452** w_2 = **c810f32b** w_3 = **809079e5**

w_4 = **62f8ead2** w_5 = **522c6b7b**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e

9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd
13	4db7b4bd					2402f5a5	69b54118
14	69b54118					ec12068e	85a74796
15	85a74796					6c827f6b	e92538fd
16	e92538fd					0e7a95b9	e75fad44
17	e75fad44					5c56fec2	bb095386
18	bb095386	095386bb	01ed44ea	04000000	05ed44ea	4db7b4bd	485af057
19	485af057					69b54118	21efb14f
20	21efb14f					85a74796	a448f6d9
21	a448f6d9					e92538fd	4d6dce24
22	4d6dce24					e75fad44	aa326360
23	aa326360					bb095386	113b30e6
24	113b30e6	3b30e611	e2048e82	08000000	ea048e82	485af057	a25e7ed5
25	a25e7ed5					21efb14f	83b1cf9a
26	83b1cf9a					a448f6d9	27f93943
27	27f93943					4d6dce24	6a94f767
28	6a94f767					aa326360	c0a69407
29	c0a69407					113b30e6	d19da4e1
30	d19da4e1	9da4e1d1	5e49f83e	10000000	4e49f83e	a25e7ed5	ec1786eb
31	ec1786eb					83b1cf9a	6fa64971
32	6fa64971					27f93943	485f7032
33	485f7032					6a94f767	22cb8755
34	22cb8755					c0a69407	e26d1352
35	e26d1352					d19da4e1	33f0b7b3
36	33f0b7b3	f0b7b333	8ca96dc3	20000000	aca96dc3	ec1786eb	40beeb28
37	40beeb28					6fa64971	2f18a259
38	2f18a259					485f7032	6747d26b
39	6747d26b					22cb8755	458c553e
40	458c553e					e26d1352	a7e1466c
41	a7e1466c					33f0b7b3	9411f1df
42	9411f1df	11f1df94	82a19e22	40000000	c2a19e22	40beeb28	821f750a
43	821f750a					2f18a259	ad07d753

44	ad07d753					6747d26b	ca400538
45	ca400538					458c553e	8fcc5006
46	8fcc5006					a7e1466c	282d166a
47	282d166a					9411f1df	bc3ce7b5
48	bc3ce7b5	3ce7b5bc	eb94d565	80000000	6b94d565	821f750a	e98ba06f
49	e98ba06f					ad07d753	448c773c
50	448c773c					ca400538	8ecc7204
51	8ecc7204					8fcc5006	01002202

A.3 Expansion of a 256-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =      60 3d eb 10 15 ca 71 be 2b 73 ae f0 85 7d 77 81
                  1f 35 2c 07 3b 61 08 d7 2d 98 10 a3 09 14 df f4
```

for ***Nk*** = 8, which results in

w_0 = **603deb10** w_1 = **15ca71be** w_2 = **2b73aef0** w_3 = **857d7781**

w_4 = **1f352c07** w_5 = **3b6108d7** w_6 = **2d9810a3** w_7 = **0914dff4**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8
17	d59aecb8					8e6925af	5bf3c917
18	5bf3c917					a51a8b5f	fee94248
19	fee94248					2067fcde	de8ebe96
20	de8ebe96		1d19ae90			a8b09c1a	b5a9328a
21	b5a9328a					93d194cd	2678a647
22	2678a647					be49846e	98312229

23	98312229					b75d5b9a	2f6c79b3
24	2f6c79b3	6c79b32f	50b66d15	04000000	54b66d15	d59aecb8	812c81ad
25	812c81ad					5bf3c917	dadf48ba
26	dadf48ba					fee94248	24360af2
27	24360af2					de8ebe96	fab8b464
28	fab8b464		2d6c8d43			b5a9328a	98c5bfc9
29	98c5bfc9					2678a647	bebd198e
30	bebd198e					98312229	268c3ba7
31	268c3ba7					2f6c79b3	09e04214
32	09e04214	e0421409	e12cfa01	08000000	e92cfa01	812c81ad	68007bac
33	68007bac					dadf48ba	b2df3316
34	b2df3316					24360af2	96e939e4
35	96e939e4					fab8b464	6c518d80
36	6c518d80		50d15dcd			98c5bfc9	c814e204
37	c814e204					bebd198e	76a9fb8a
38	76a9fb8a					268c3ba7	5025c02d
39	5025c02d					09e04214	59c58239
40	59c58239	c5823959	a61312cb	10000000	b61312cb	68007bac	de136967
41	de136967					b2df3316	6ccc5a71
42	6ccc5a71					96e939e4	fa256395
43	fa256395					6c518d80	9674ee15
44	9674ee15		90922859			c814e204	5886ca5d
45	5886ca5d					76a9fb8a	2e2f31d7
46	2e2f31d7					5025c02d	7e0af1fa
47	7e0af1fa					59c58239	27cf73c3
48	27cf73c3	cf73c327	8a8f2ecc	20000000	aa8f2ecc	de136967	749c47ab
49	749c47ab					6ccc5a71	18501dda
50	18501dda					fa256395	e2757e4f
51	e2757e4f					9674ee15	7401905a
52	7401905a		927c60be			5886ca5d	cafaaae3
53	cafaaae3					2e2f31d7	e4d59b34
54	e4d59b34					7e0af1fa	9adf6ace
55	9adf6ace					27cf73c3	bd10190d
56	bd10190d	10190dbd	cad4d77a	40000000	8ad4d77a	749c47ab	fe4890d1
57	fe4890d1					18501dda	e6188d0b

58	e6188d0b					e2757e4f	046df344
59	046df344					7401905a	706c631e

Appendix B – Cipher Example

The following diagram shows the values in the State array as the Cipher progresses for a block length and a Cipher Key length of 16 bytes each (i.e., *Nb* = 4 and *Nk* = 4).

```
Input =      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
```

The Round Key values are taken from the Key Expansion example in Appendix A.

Round Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns		Round Key Value	
input	32 88 31 e0 43 5a 31 37 f6 30 98 07 a8 8d a2 34				⊕	2b 28 ab 09 7e ae f7 cf 15 d2 15 4f 16 a6 88 3c	=
1	19 a0 9a e9 3d f4 c6 f8 e3 e2 8d 48 be 2b 2a 08	d4 e0 b8 1e 27 bf b4 41 11 98 5d 52 ae f1 e5 30	d4 e0 b8 1e bf b4 41 27 5d 52 11 98 30 ae f1 e5	04 e0 48 28 66 cb f8 06 81 19 d3 26 e5 9a 7a 4c	⊕	a0 88 23 2a fa 54 a3 6c fe 2c 39 76 17 b1 39 05	=
2	a4 68 6b 02 9c 9f 5b 6a 7f 35 ea 50 f2 2b 43 49	49 45 7f 77 de db 39 02 d2 96 87 53 89 f1 1a 3b	49 45 7f 77 db 39 02 de 87 53 d2 96 3b 89 f1 1a	58 1b db 1b 4d 4b e7 6b ca 5a ca b0 f1 ac a8 e5	⊕	f2 7a 59 73 c2 96 35 59 95 b9 80 f6 f2 43 7a 7f	=
3	aa 61 82 68 8f dd d2 32 5f e3 4a 46 03 ef d2 9a	ac ef 13 45 73 c1 b5 23 cf 11 d6 5a 7b df b5 b8	ac ef 13 45 c1 b5 23 73 d6 5a cf 11 b8 7b df b5	75 20 53 bb ec 0b c0 25 09 63 cf d0 93 33 7c dc	⊕	3d 47 1e 6d 80 16 23 7a 47 fe 7e 88 7d 3e 44 3b	=
4	48 67 4d d6 6c 1d e3 5f 4e 9d b1 58 ee 0d 38 e7	52 85 e3 f6 50 a4 11 cf 2f 5e c8 6a 28 d7 07 94	52 85 e3 f6 a4 11 cf 50 c8 6a 2f 5e 94 28 d7 07	0f 60 6f 5e d6 31 c0 b3 da 38 10 13 a9 bf 6b 01	⊕	ef a8 b6 db 44 52 71 0b a5 5b 25 ad 41 7f 3b 00	=
5	e0 c8 d9 85 92 63 b1 b8 7f 63 35 be e8 c0 50 01	e1 e8 35 97 4f fb c8 6c d2 fb 96 ae 9b ba 53 7c	e1 e8 35 97 fb c8 6c 4f 96 ae d2 fb 7c 9b ba 53	25 bd b6 4c d1 11 3a 4c a9 d1 33 c0 ad 68 8e b0	⊕	d4 7c ca 11 d1 83 f2 f9 c6 9d b8 15 f8 87 bc bc	=

6	f1	c1	7c	5d	a1	78	10	4c	a1	78	10	4c	4b	2c	33	37	⊕	6d	11	db	ca	=
	00	92	c8	b5	63	4f	e8	d5	4f	e8	d5	63	86	4a	9d	d2		88	0b	f9	00	
	6f	4c	8b	d5	a8	29	3d	03	3d	03	a8	29	8d	89	f4	18		a3	3e	86	93	
	55	ef	32	0c	fc	df	23	fe	fe	fc	df	23	6d	80	e8	d8		7a	fd	41	fd	
7	26	3d	e8	fd	f7	27	9b	54	f7	27	9b	54	14	46	27	34	⊕	4e	5f	84	4e	=
	0e	41	64	d2	ab	83	43	b5	83	43	b5	ab	15	16	46	2a		54	5f	a6	a6	
	2e	b7	72	8b	31	a9	40	3d	40	3d	31	a9	b5	15	56	d8		f7	c9	4f	dc	
	17	7d	a9	25	f0	ff	d3	3f	3f	f0	ff	d3	bf	ec	d7	43		0e	f3	b2	4f	
8	5a	19	a3	7a	be	d4	0a	da	be	d4	0a	da	00	b1	54	fa	⊕	ea	b5	31	7f	=
	41	49	e0	8c	83	3b	e1	64	3b	e1	64	83	51	c8	76	1b		d2	8d	2b	8d	
	42	dc	19	04	2c	86	d4	f2	d4	f2	2c	86	2f	89	6d	99		73	ba	f5	29	
	b1	1f	65	0c	c8	c0	4d	fe	fe	c8	c0	4d	d1	ff	cd	ea		21	d2	60	2f	
9	ea	04	65	85	87	f2	4d	97	87	f2	4d	97	47	40	a3	4c	⊕	ac	19	28	57	=
	83	45	5d	96	ec	6e	4c	90	6e	4c	90	ec	37	d4	70	9f		77	fa	d1	5c	
	5c	33	98	b0	4a	c3	46	e7	46	e7	4a	c3	94	e4	3a	42		66	dc	29	00	
	f0	2d	ad	c5	8c	d8	95	a6	a6	8c	d8	95	ed	a5	a6	bc		f3	21	41	6e	
10	eb	59	8b	1b	e9	cb	3d	af	e9	cb	3d	af					⊕	d0	c9	e1	b6	=
	40	2e	a1	c3	09	31	32	2e	31	32	2e	09						14	ee	3f	63	
	f2	38	13	42	89	07	7d	2c	7d	2c	89	07						f9	25	0c	0c	
	1e	84	e7	d2	72	5f	94	b5	b5	72	5f	94						a8	89	c8	a6	

output	39	02	dc	19
	25	dc	11	6a
	84	09	85	0b
	1d	fb	97	32

Appendix C – Example Vectors

This appendix contains example vectors, including intermediate values – for all three AES key lengths (***Nk*** = 4, 6, and 8), for the Cipher, Inverse Cipher, and Equivalent Inverse Cipher that are described in Sec. 5.1, 5.3, and 5.3.5, respectively. Additional examples may be found at [1] and [5].

All vectors are in hexadecimal notation, with each pair of characters giving a byte value in which the left character of each pair provides the bit pattern for the 4 bit group containing the higher numbered bits using the notation explained in Sec. 3.2, while the right character provides the bit pattern for the lower-numbered bits. The array index for all bytes (groups of two hexadecimal digits) within these test vectors starts at zero and increases from left to right.

```
Legend for CIPHER (ENCRYPT) (round number r = 0 to 10, 12 or 14):

   input:   cipher input
   start:   state at start of round[r]
   s_box:   state after SubBytes()
   s_row:   state after ShiftRows()
   m_col:   state after MixColumns()
   k_sch:   key schedule value for round[r]
   output:  cipher output


Legend for INVERSE CIPHER (DECRYPT) (round number r = 0 to 10, 12 or 14):
   iinput:  inverse cipher input
   istart:  state at start of round[r]
   is_box:  state after InvSubBytes()
   is_row:  state after InvShiftRows()
   ik_sch:  key schedule value for round[r]
   ik_add:  state after AddRoundKey()
   ioutput: inverse cipher output


Legend for EQUIVALENT INVERSE CIPHER (DECRYPT) (round number r = 0 to 10, 12
   or 14):

   iinput:  inverse cipher input
   istart:  state at start of round[r]
   is_box:  state after InvSubBytes()
   is_row:  state after InvShiftRows()
   im_col:  state after InvMixColumns()
   ik_sch:  key schedule value for round[r]
   ioutput: inverse cipher output
```

C.1 AES-128 (*Nk*=4, *Nr*=10)

```
PLAINTEXT:         00112233445566778899aabbccddeeff
KEY:               000102030405060708090a0b0c0d0e0f

CIPHER (ENCRYPT):
```

```
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 2].start    89d810e8855ace682d1843d8cb128fe4
round[ 2].s_box    a761ca9b97be8b45d8ad1a611fc97369
round[ 2].s_row    a7be1a6997ad739bd8c9ca451f618b61
round[ 2].m_col    ff87968431d86a51645151fa773ad009
round[ 2].k_sch    b692cf0b643dbdf1be9bc5006830b3fe
round[ 3].start    4915598f55e5d7a0daca94fa1f0a63f7
round[ 3].s_box    3b59cb73fcd90ee05774222dc067fb68
round[ 3].s_row    3bd92268fc74fb735767cbe0c0590e2d
round[ 3].m_col    4c9c1e66f771f0762c3f868e534df256
round[ 3].k_sch    b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 4].start    fa636a2825b339c940668a3157244d17
round[ 4].s_box    2dfb02343f6d12dd09337ec75b36e3f0
round[ 4].s_row    2d6d7ef03f33e334093602dd5bfb12c7
round[ 4].m_col    6385b79ffc538df997be478e7547d691
round[ 4].k_sch    47f7f7bc95353e03f96c32bcfd058dfd
round[ 5].start    247240236966b3fa6ed2753288425b6c
round[ 5].s_box    36400926f9336d2d9fb59d23c42c3950
round[ 5].s_row    36339d50f9b539269f2c092dc4406d23
round[ 5].m_col    f4bcd45432e554d075f1d6c51dd03b3c
round[ 5].k_sch    3caaa3e8a99f9deb50f3af57adf622aa
round[ 6].start    c81677bc9b7ac93b25027992b0261996
round[ 6].s_box    e847f56514dadde23f77b64fe7f7d490
round[ 6].s_row    e8dab6901477d4653ff7f5e2e747dd4f
round[ 6].m_col    9816ee7400f87f556b2c049c8e5ad036
round[ 6].k_sch    5e390f7df7a69296a7553dc10aa31f6b
round[ 7].start    c62fe109f75eedc3cc79395d84f9cf5d
round[ 7].s_box    b415f8016858552e4bb6124c5f998a4c
round[ 7].s_row    b458124c68b68a014b99f82e5f15554c
round[ 7].m_col    c57e1c159a9bd286f05f4be098c63439
round[ 7].k_sch    14f9701ae35fe28c440adf4d4ea9c026
round[ 8].start    d1876c0f79c4300ab45594add66ff41f
round[ 8].s_box    3e175076b61c04678dfc2295f6a8bfc0
round[ 8].s_row    3e1c22c0b6fcbf768da85067f6170495
round[ 8].m_col    baa03de7a1f9b56ed5512cba5f414d23
round[ 8].k_sch    47438735a41c65b9e016baf4aebf7ad2
round[ 9].start    fde3bad205e5d0d73547964ef1fe37f1
round[ 9].s_box    5411f4b56bd9700e96a0902fa1bb9aa1
round[ 9].s_row    54d990a16ba09ab596bbf40ea111702f
round[ 9].m_col    e9f74eec023020f61bf2ccf2353c21c7
round[ 9].k_sch    549932d1f08557681093ed9cbe2c974e
round[10].start    bd6e7c3df2b5779e0b61216e8b10b689
round[10].s_box    7a9f102789d5f50b2beffd9f3dca4ea7
round[10].s_row    7ad5fda789ef4e272bca100b3d9ff59f
round[10].k_sch    13111d7fe3944a17f307a78b4d2b30c5
round[10].output   69c4e0d86a7b0430d8cdb78070b4c55a

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch   13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart   7ad5fda789ef4e272bca100b3d9ff59f
```

```
round[ 1].is_row   7a9f102789d5f50b2beffd9f3dca4ea7
round[ 1].is_box   bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].ik_sch   549932d1f08557681093ed9cbe2c974e
round[ 1].ik_add   e9f74eec023020f61bf2ccf2353c21c7
round[ 2].istart   54d990a16ba09ab596bbf40ea111702f
round[ 2].is_row   5411f4b56bd9700e96a0902fa1bb9aa1
round[ 2].is_box   fde3bad205e5d0d73547964ef1fe37f1
round[ 2].ik_sch   47438735a41c65b9e016baf4aebf7ad2
round[ 2].ik_add   baa03de7a1f9b56ed5512cba5f414d23
round[ 3].istart   3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_row   3e175076b61c04678dfc2295f6a8bfc0
round[ 3].is_box   d1876c0f79c4300ab45594add66ff41f
round[ 3].ik_sch   14f9701ae35fe28c440adf4d4ea9c026
round[ 3].ik_add   c57e1c159a9bd286f05f4be098c63439
round[ 4].istart   b458124c68b68a014b99f82e5f15554c
round[ 4].is_row   b415f8016858552e4bb6124c5f998a4c
round[ 4].is_box   c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].ik_sch   5e390f7df7a69296a7553dc10aa31f6b
round[ 4].ik_add   9816ee7400f87f556b2c049c8e5ad036
round[ 5].istart   e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_row   e847f56514dadde23f77b64fe7f7d490
round[ 5].is_box   c81677bc9b7ac93b25027992b0261996
round[ 5].ik_sch   3caaa3e8a99f9deb50f3af57adf622aa
round[ 5].ik_add   f4bcd45432e554d075f1d6c51dd03b3c
round[ 6].istart   36339d50f9b539269f2c092dc4406d23
round[ 6].is_row   36400926f9336d2d9fb59d23c42c3950
round[ 6].is_box   247240236966b3fa6ed2753288425b6c
round[ 6].ik_sch   47f7f7bc95353e03f96c32bcfd058dfd
round[ 6].ik_add   6385b79ffc538df997be478e7547d691
round[ 7].istart   2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_row   2dfb02343f6d12dd09337ec75b36e3f0
round[ 7].is_box   fa636a2825b339c940668a3157244d17
round[ 7].ik_sch   b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 7].ik_add   4c9c1e66f771f0762c3f868e534df256
round[ 8].istart   3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_row   3b59cb73fcd90ee05774222dc067fb68
round[ 8].is_box   4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].ik_sch   b692cf0b643dbdf1be9bc5006830b3fe
round[ 8].ik_add   ff87968431d86a51645151fa773ad009
round[ 9].istart   a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_row   a761ca9b97be8b45d8ad1a611fc97369
round[ 9].is_box   89d810e8855ace682d1843d8cb128fe4
round[ 9].ik_sch   d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 9].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[10].istart   6353e08c0960e104cd70b751bacad0e7
round[10].is_row   63cab7040953d051cd60e0e7ba70e18c
round[10].is_box   00102030405060708090a0b0c0d0e0f0
round[10].ik_sch   000102030405060708090a0b0c0d0e0f
round[10].ioutput  00112233445566778899aabbccddeeff

EQUIVALENT INVERSE CIPHER (DECRYPT):
round[ 0].iinput   69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch   13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart   7ad5fda789ef4e272bca100b3d9ff59f
round[ 1].is_box   bdb52189f261b63d0b107c9e8b6e776e
round[ 1].is_row   bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].im_col   4773b91ff72f354361cb018ea1e6cf2c
```

```
round[ 1].ik_sch   13aa29be9c8faff6f770f58000f7bf03
round[ 2].istart   54d990a16ba09ab596bbf40ea111702f
round[ 2].is_box   fde596f1054737d235febad7f1e3d04e
round[ 2].is_row   fde3bad205e5d0d73547964ef1fe37f1
round[ 2].im_col   2d7e86a339d9393ee6570a1101904e16
round[ 2].ik_sch   1362a4638f2586486bff5a76f7874a83
round[ 3].istart   3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_box   d1c4941f7955f40fb46f6c0ad68730ad
round[ 3].is_row   d1876c0f79c4300ab45594add66ff41f
round[ 3].im_col   39daee38f4f1a82aaf432410c36d45b9
round[ 3].ik_sch   8d82fc749c47222be4dadc3e9c7810f5
round[ 4].istart   b458124c68b68a014b99f82e5f15554c
round[ 4].is_box   c65e395df779cf09ccf9e1c3842fed5d
round[ 4].is_row   c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].im_col   9a39bf1d05b20a3a476a0bf79fe51184
round[ 4].ik_sch   72e3098d11c5de5f789dfe1578a2cccb
round[ 5].istart   e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_box   c87a79969b0219bc2526773bb016c992
round[ 5].is_row   c81677bc9b7ac93b25027992b0261996
round[ 5].im_col   18f78d779a93eef4f6742967c47f5ffd
round[ 5].ik_sch   2ec410276326d7d26958204a003f32de
round[ 6].istart   36339d50f9b539269f2c092dc4406d23
round[ 6].is_box   2466756c69d25b236e4240fa8872b332
round[ 6].is_row   247240236966b3fa6ed2753288425b6c
round[ 6].im_col   85cf8bf472d124c10348f545329c0053
round[ 6].ik_sch   a8a2f5044de2c7f50a7ef79869671294
round[ 7].istart   2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_box   fab38a1725664d2840246ac957633931
round[ 7].is_row   fa636a2825b339c940668a3157244d17
round[ 7].im_col   fc1fc1f91934c98210fbfb8da340eb21
round[ 7].ik_sch   c7c6e391e54032f1479c306d6319e50c
round[ 8].istart   3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_box   49e594f755ca638fda0a59a01f15d7fa
round[ 8].is_row   4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].im_col   076518f0b52ba2fb7a15c8d93be45e00
round[ 8].ik_sch   a0db02992286d160a2dc029c2485d561
round[ 9].istart   a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_box   895a43e485188fe82d121068cbd8ced8
round[ 9].is_row   89d810e8855ace682d1843d8cb128fe4
round[ 9].im_col   ef053f7c8b3d32fd4d2a64ad3c93071a
round[ 9].ik_sch   8c56dff0825dd3f9805ad3fc8659d7fd
round[10].istart   6353e08c0960e104cd70b751bacad0e7
round[10].is_box   0050a0f04090e03080d02070c01060b0
round[10].is_row   00102030405060708090a0b0c0d0e0f0
round[10].ik_sch   000102030405060708090a0b0c0d0e0f
round[10].ioutput  00112233445566778899aabbccddeeff
```

C.2 AES-192 (Nk=6, Nr=12)

```
PLAINTEXT:  00112233445566778899aabbccddeeff
KEY:        000102030405060708090a0b0c0d0e0f1011121314151617

CIPHER (ENCRYPT):
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
```

```
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    101112131415161758 46f2f95c43f4fe
round[ 2].start    4f63760643e0aa85aff8c9d041fa0de4
round[ 2].s_box    84fb386f1ae1ac977941dd70832dd769
round[ 2].s_row    84e1dd691a41d76f792d389783fbac70
round[ 2].m_col    9f487f794f955f662afc86abd7f1ab29
round[ 2].k_sch    544afef55847f0fa4856e2e95c43f4fe
round[ 3].start    cb02818c17d2af9c62aa64428bb25fd7
round[ 3].s_box    1f770c64f0b579deaaac432c3d37cf0e
round[ 3].s_row    1fb5430ef0accf64aa370cde3d77792c
round[ 3].m_col    b7a53ecbbf9d75a0c40efc79b674cc11
round[ 3].k_sch    40f949b31cbabd4d48f043b810b7b342
round[ 4].start    f75c7778a327c8ed8cfebfc1a6c37f53
round[ 4].s_box    684af5bc0acce85564bb0878242ed2ed
round[ 4].s_row    68cc08ed0abbd2bc642ef555244ae878
round[ 4].m_col    7a1e98bdacb6d1141a6944dd06eb2d3e
round[ 4].k_sch    58e151ab04a2a5557effb5416245080c
round[ 5].start    22ffc916a81474416496f19c64ae2532
round[ 5].s_box    9316dd47c2fa92834390a1de43e43f23
round[ 5].s_row    93faa123c2903f4743e4dd83431692de
round[ 5].m_col    aaa755b34cffe57cef6f98e1f01c13e6
round[ 5].k_sch    2ab54bb43a02f8f662e3a95d66410c08
round[ 6].start    80121e0776fd1d8a8d8c31bc965d1fee
round[ 6].s_box    cdc972c53854a47e5d64c765904cc028
round[ 6].s_row    cd54c7283864c0c55d4c727e90c9a465
round[ 6].m_col    921f748fd96e937d622d7725ba8ba50c
round[ 6].k_sch    f501857297448d7ebdf1c6ca87f33e3c
round[ 7].start    671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 7].s_box    8572a1542fe5727b9e86c8df27bc1404
round[ 7].s_row    85e5c8042f8614549ebca17b277272df
round[ 7].m_col    e913e7b18f507d4b227ef652758acbcc
round[ 7].k_sch    e510976183519b6934157c9ea351f1e0
round[ 8].start    0c0370d00c01e622166b8accd6db3a2c
round[ 8].s_box    fe7b5170fe7c8e93477f7e4bf6b98071
round[ 8].s_row    fe7c7e71fe7f807047b95193f67b8e4b
round[ 8].m_col    6cf5edf996eb0a069c4ef21cbfc25762
round[ 8].k_sch    1ea0372a995309167c439e77ff12051e
round[ 9].start    7255dad30fb80310e00d6c6b40d0527c
round[ 9].s_box    40fc5766766c7bcae1d7507f09700010
round[ 9].s_row    406c501076d70066e17057ca09fc7b7f
round[ 9].m_col    7478bcdce8a50b81d4327a9009188262
round[ 9].k_sch    dd7e0e887e2fff68608fc842f9dcc154
round[10].start    a906b254968af4e9b4bdb2d2f0c44336
round[10].s_box    d36f3720907ebf1e8d7a37b58c1c1a05
round[10].s_row    d37e3705907a1a208d1c371e8c6fbfb5
round[10].m_col    0d73cc2d8f6abe8b0cf2dd9bb83d422e
round[10].k_sch    859f5f237a8d5a3dc0c02952beefd63a
round[11].start    88ec930ef5e7e4b6cc32f4c906d29414
round[11].s_box    c4cedcabe694694e4b23bfdd6fb522fa
round[11].s_row    c494bffae62322ab4bb5dc4e6fce69dd
round[11].m_col    71d720933b6d677dc00b8f28238e0fb7
round[11].k_sch    de601e7827bcdf2ca223800fd8aeda32
round[12].start    afb73eeb1cd1b85162280f27fb20d585
round[12].s_box    79a9b2e99c3e6cd1aa3476cc0fb70397
round[12].s_row    793e76979c3403e9aab7b2d10fa96ccc
```

```
round[12].k_sch    a4970a331a78dc09c418c271e3a41d5d
round[12].output   dda97ca4864cdfe06eaf70a0ec0d7191

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   dda97ca4864cdfe06eaf70a0ec0d7191
round[ 0].ik_sch   a4970a331a78dc09c418c271e3a41d5d
round[ 1].istart   793e76979c3403e9aab7b2d10fa96ccc
round[ 1].is_row   79a9b2e99c3e6cd1aa3476cc0fb70397
round[ 1].is_box   afb73eeb1cd1b85162280f27fb20d585
round[ 1].ik_sch   de601e7827bcdf2ca223800fd8aeda32
round[ 1].ik_add   71d720933b6d677dc00b8f28238e0fb7
round[ 2].istart   c494bffae62322ab4bb5dc4e6fce69dd
round[ 2].is_row   c4cedcabe694694e4b23bfdd6fb522fa
round[ 2].is_box   88ec930ef5e7e4b6cc32f4c906d29414
round[ 2].ik_sch   859f5f237a8d5a3dc0c02952beefd63a
round[ 2].ik_add   0d73cc2d8f6abe8b0cf2dd9bb83d422e
round[ 3].istart   d37e3705907a1a208d1c371e8c6fbfb5
round[ 3].is_row   d36f3720907ebf1e8d7a37b58c1c1a05
round[ 3].is_box   a906b254968af4e9b4bdb2d2f0c44336
round[ 3].ik_sch   dd7e0e887e2fff68608fc842f9dcc154
round[ 3].ik_add   7478bcdce8a50b81d4327a9009188262
round[ 4].istart   406c501076d70066e17057ca09fc7b7f
round[ 4].is_row   40fc5766766c7bcae1d7507f09700010
round[ 4].is_box   7255dad30fb80310e00d6c6b40d0527c
round[ 4].ik_sch   1ea0372a995309167c439e77ff12051e
round[ 4].ik_add   6cf5edf996eb0a069c4ef21cbfc25762
round[ 5].istart   fe7c7e71fe7f807047b95193f67b8e4b
round[ 5].is_row   fe7b5170fe7c8e93477f7e4bf6b98071
round[ 5].is_box   0c0370d00c01e622166b8accd6db3a2c
round[ 5].ik_sch   e510976183519b6934157c9ea351f1e0
round[ 5].ik_add   e913e7b18f507d4b227ef652758acbcc
round[ 6].istart   85e5c8042f8614549ebca17b277272df
round[ 6].is_row   8572a1542fe5727b9e86c8df27bc1404
round[ 6].is_box   671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 6].ik_sch   f501857297448d7ebdf1c6ca87f33e3c
round[ 6].ik_add   921f748fd96e937d622d7725ba8ba50c
round[ 7].istart   cd54c7283864c0c55d4c727e90c9a465
round[ 7].is_row   cdc972c53854a47e5d64c765904cc028
round[ 7].is_box   80121e0776fd1d8a8d8c31bc965d1fee
round[ 7].ik_sch   2ab54bb43a02f8f662e3a95d66410c08
round[ 7].ik_add   aaa755b34cffe57cef6f98e1f01c13e6
round[ 8].istart   93faa123c2903f4743e4dd83431692de
round[ 8].is_row   9316dd47c2fa92834390a1de43e43f23
round[ 8].is_box   22ffc916a81474416496f19c64ae2532
round[ 8].ik_sch   58e151ab04a2a5557effb5416245080c
round[ 8].ik_add   7a1e98bdacb6d1141a6944dd06eb2d3e
round[ 9].istart   68cc08ed0abbd2bc642ef555244ae878
round[ 9].is_row   684af5bc0acce85564bb0878242ed2ed
round[ 9].is_box   f75c7778a327c8ed8cfebfc1a6c37f53
round[ 9].ik_sch   40f949b31cbabd4d48f043b810b7b342
round[ 9].ik_add   b7a53ecbbf9d75a0c40efc79b674cc11
round[10].istart   1fb5430ef0accf64aa370cde3d77792c
round[10].is_row   1f770c64f0b579deaaac432c3d37cf0e
round[10].is_box   cb02818c17d2af9c62aa64428bb25fd7
round[10].ik_sch   544afef55847f0fa4856e2e95c43f4fe
round[10].ik_add   9f487f794f955f662afc86abd7f1ab29
round[11].istart   84e1dd691a41d76f792d389783fbac70
```

```
round[11].is_row   84fb386f1ae1ac977941dd70832dd769
round[11].is_box   4f63760643e0aa85aff8c9d041fa0de4
round[11].ik_sch   101112131415161758 46f2f95c43f4fe
round[11].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[12].istart   6353e08c0960e104cd70b751bacad0e7
round[12].is_row   63cab7040953d051cd60e0e7ba70e18c
round[12].is_box   00102030405060708090a0b0c0d0e0f0
round[12].ik_sch   000102030405060708090a0b0c0d0e0f
round[12].ioutput  00112233445566778899aabbccddeeff

EQUIVALENT INVERSE CIPHER (DECRYPT):
round[ 0].iinput   dda97ca4864cdfe06eaf70a0ec0d7191
round[ 0].ik_sch   a4970a331a78dc09c418c271e3a41d5d
round[ 1].istart   793e76979c3403e9aab7b2d10fa96ccc
round[ 1].is_box   afd10f851c28d5eb62203e51fbb7b827
round[ 1].is_row   afb73eeb1cd1b85162280f27fb20d585
round[ 1].im_col   122a02f7242ac8e20605afce51cc7264
round[ 1].ik_sch   d6bebd0dc209ea494db073803e021bb9
round[ 2].istart   c494bffae62322ab4bb5dc4e6fce69dd
round[ 2].is_box   88e7f414f532940eccd293b606ece4c9
round[ 2].is_row   88ec930ef5e7e4b6cc32f4c906d29414
round[ 2].im_col   5cc7aecce3c872194ae5ef8309a933c7
round[ 2].ik_sch   8fb999c973b26839c7f9d89d85c68c72
round[ 3].istart   d37e3705907a1a208d1c371e8c6fbfb5
round[ 3].is_box   a98ab23696bd4354b4c4b2e9f006f4d2
round[ 3].is_row   a906b254968af4e9b4bdb2d2f0c44336
round[ 3].im_col   b7113ed134e85489b20866b51d4b2c3b
round[ 3].ik_sch   f77d6ec1423f54ef5378317f14b75744
round[ 4].istart   406c501076d70066e17057ca09fc7b7f
round[ 4].is_box   72b86c7c0f0d52d3e0d0da104055036b
round[ 4].is_row   7255dad30fb80310e00d6c6b40d0527c
round[ 4].im_col   ef3b1be1b9b0e64bdcb79f1e0a707fbb
round[ 4].ik_sch   1147659047cf663b9b0ece8dfc0bf1f0
round[ 5].istart   fe7c7e71fe7f807047b95193f67b8e4b
round[ 5].is_box   0c018a2c0c6b3ad016db7022d603e6cc
round[ 5].is_row   0c0370d00c01e622166b8accd6db3a2c
round[ 5].im_col   592460b248832b2952e0b831923048f1
round[ 5].ik_sch   dcc1a8b667053f7dcc5c194ab5423a2e
round[ 6].istart   85e5c8042f8614549ebca17b277272df
round[ 6].is_box   672ab1304edc9bfddf78f1033d1e1eef
round[ 6].is_row   671ef1fd4e2a1e03dfdcb1ef3d789b30
round[ 6].im_col   0b8a7783417ae3a1f9492dc0c641a7ce
round[ 6].ik_sch   c6deb0ab791e2364a4055fbe568803ab
round[ 7].istart   cd54c7283864c0c55d4c727e90c9a465
round[ 7].is_box   80fd31ee768c1f078d5d1e8a96121dbc
round[ 7].is_row   80121e0776fd1d8a8d8c31bc965d1fee
round[ 7].im_col   4ee1ddf9301d6352c9ad769ef8d20515
round[ 7].ik_sch   dd1b7cdaf28d5c158a49ab1dbbc497cb
round[ 8].istart   93faa123c2903f4743e4dd83431692de
round[ 8].is_box   2214f132a896251664aec94164ff749c
round[ 8].is_row   22ffc916a81474416496f19c64ae2532
round[ 8].im_col   1008ffe53b36ee6af27b42549b8a7bb7
round[ 8].ik_sch   78c4f708318d3cd69655b701bfc093cf
round[ 9].istart   68cc08ed0abbd2bc642ef555244ae878
round[ 9].is_box   f727bf53a3fe7f788cc377eda65cc8c1
round[ 9].is_row   f75c7778a327c8ed8cfebfc1a6c37f53
round[ 9].im_col   7f69ac1ed939ebaac8ece3cb12e159e3
```

```
round[ 9].ik_sch   60dcef10299524ce62dbef152f9620cf
round[10].istart   1fb5430ef0accf64aa370cde3d77792c
round[10].is_box   cbd264d717aa5f8c62b2819c8b02af42
round[10].is_row   cb02818c17d2af9c62aa64428bb25fd7
round[10].im_col   cfaf16b2570c18b52e7fef50cab267ae
round[10].ik_sch   4b4ecbdb4d4dcfda5752d7c74949cbde
round[11].istart   84e1dd691a41d76f792d389783fbac70
round[11].is_box   4fe0c9e443f80d06affa76854163aad0
round[11].is_row   4f63760643e0aa85aff8c9d041fa0de4
round[11].im_col   794cf891177bfd1d8a327086f3831b39
round[11].ik_sch   1a1f181d1e1b1c194742c7d74949cbde
round[12].istart   6353e08c0960e104cd70b751bacad0e7
round[12].is_box   0050a0f04090e03080d02070c01060b0
round[12].is_row   00102030405060708090a0b0c0d0e0f0
round[12].ik_sch   000102030405060708090a0b0c0d0e0f
round[12].ioutput  00112233445566778899aabbccddeeff
```

C.3 AES-256 (Nk=8, Nr=14)

```
PLAINTEXT:  00112233445566778899aabbccddeeff
KEY:        000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

CIPHER (ENCRYPT):
round[ 0].input    00112233445566778899aabbccddeeff
round[ 0].k_sch    000102030405060708090a0b0c0d0e0f
round[ 1].start    00102030405060708090a0b0c0d0e0f0
round[ 1].s_box    63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row    6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col    5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch    101112131415161718191a1b1c1d1e1f
round[ 2].start    4f63760643e0aa85efa7213201a4e705
round[ 2].s_box    84fb386f1ae1ac97df5cfd237c49946b
round[ 2].s_row    84e1fd6b1a5c946fdf4938977cfbac23
round[ 2].m_col    bd2a395d2b6ac438d192443e615da195
round[ 2].k_sch    a573c29fa176c498a97fce93a572c09c
round[ 3].start    1859fbc28a1c00a078ed8aadc42f6109
round[ 3].s_box    adcb0f257e9c63e0bc557e951c15ef01
round[ 3].s_row    ad9c7e017e55ef25bc150fe01ccb6395
round[ 3].m_col    810dce0cc9db8172b3678c1e88a1b5bd
round[ 3].k_sch    1651a8cd0244beda1a5da4c10640bade
round[ 4].start    975c66c1cb9f3fa8a93a28df8ee10f63
round[ 4].s_box    884a33781fdb75c2d380349e19f876fb
round[ 4].s_row    88db34fb1f807678d3f833c2194a759e
round[ 4].m_col    b2822d81abe6fb275faf103a078c0033
round[ 4].k_sch    ae87dff00ff11b68a68ed5fb03fc1567
round[ 5].start    1c05f271a417e04ff921c5c104701554
round[ 5].s_box    9c6b89a349f0e18499fda678f2515920
round[ 5].s_row    9cf0a62049fd59a399518984f26be178
round[ 5].m_col    aeb65ba974e0f822d73f567bdb64c877
round[ 5].k_sch    6de1f1486fa54f9275f8eb5373b8518d
round[ 6].start    c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 6].s_box    2e5bacf8af6ea9e73ac67a34c286ee2d
round[ 6].s_row    2e6e7a2dafc6eef83a86ace7c25ba934
round[ 6].m_col    b951c33c02e9bd29ae25cdb1efa08cc7
round[ 6].k_sch    c656827fc9a799176f294cec6cd5598b
round[ 7].start    7f074143cb4e243ec10c815d8375d54c
round[ 7].s_box    d2c5831a1f2f36b278fe0c4cec9d0329
```

```
round[ 7].s_row    d22f0c291ffe031a789d83b2ecc5364c
round[ 7].m_col    ebb19e1c3ee7c9e87d7535e9ed6b9144
round[ 7].k_sch    3de23a75524775e727bf9eb45407cf39
round[ 8].start    d653a4696ca0bc0f5acaab5db96c5e7d
round[ 8].s_box    f6ed49f950e06576be74624c565058ff
round[ 8].s_row    f6e062ff507458f9be50497656ed654c
round[ 8].m_col    5174c8669da98435a8b3e62ca974a5ea
round[ 8].k_sch    0bdc905fc27b0948ad5245a4c1871c2f
round[ 9].start    5aa858395fd28d7d05e1a38868f3b9c5
round[ 9].s_box    bec26a12cfb55dff6bf80ac4450d56a6
round[ 9].s_row    beb50aa6cff856126b0d6aff45c25dc4
round[ 9].m_col    0f77ee31d2ccadc05430a83f4ef96ac3
round[ 9].k_sch    45f5a66017b2d387300d4d33640a820a
round[10].start    4a824851c57e7e47643de50c2af3e8c9
round[10].s_box    d61352d1a6f3f3a04327d9fee50d9bdd
round[10].s_row    d6f3d9dda6279bd1430d52a0e513f3fe
round[10].m_col    bd86f0ea748fc4f4630f11c1e9331233
round[10].k_sch    7ccff71cbeb4fe5413e6bbf0d261a7df
round[11].start    c14907f6ca3b3aa070e9aa313b52b5ec
round[11].s_box    783bc54274e280e0511eacc7e200d5ce
round[11].s_row    78e2acce741ed5425100c5e0e23b80c7
round[11].m_col    af8690415d6e1dd387e5fbedd5c89013
round[11].k_sch    f01afafee7a82979d7a5644ab3afe640
round[12].start    5f9c6abfbac634aa50409fa766677653
round[12].s_box    cfde0208f4b418ac5309db5c338538ed
round[12].s_row    cfb4dbedf4093808538502ac33de185c
round[12].m_col    7427fae4d8a695269ce83d315be0392b
round[12].k_sch    2541fe719bf500258813bbd55a721c0a
round[13].start    51660495435395031 4fb86e401922521
round[13].s_box    d133f22a1aed2a7bfa0f44697c4f3ffd
round[13].s_row    d1ed44fd1a0f3f2afa4ff27b7c332a69
round[13].m_col    2c21a820306f154ab712c75eee0da04f
round[13].k_sch    4e5a6699a9f24fe07e572baacdf8cdea
round[14].start    627bceb9999d5aaac945ecf423f56da5
round[14].s_box    aa218b56ee5ebeacdd6ecebf26e63c06
round[14].s_row    aa5ece06ee6e3c56dde68bac2621bebf
round[14].k_sch    24fc79ccbf0979e9371ac23c6d68de36
round[14].output   8ea2b7ca516745bfeafc49904b496089

INVERSE CIPHER (DECRYPT):
round[ 0].iinput   8ea2b7ca516745bfeafc49904b496089
round[ 0].ik_sch   24fc79ccbf0979e9371ac23c6d68de36
round[ 1].istart   aa5ece06ee6e3c56dde68bac2621bebf
round[ 1].is_row   aa218b56ee5ebeacdd6ecebf26e63c06
round[ 1].is_box   627bceb9999d5aaac945ecf423f56da5
round[ 1].ik_sch   4e5a6699a9f24fe07e572baacdf8cdea
round[ 1].ik_add   2c21a820306f154ab712c75eee0da04f
round[ 2].istart   d1ed44fd1a0f3f2afa4ff27b7c332a69
round[ 2].is_row   d133f22a1aed2a7bfa0f44697c4f3ffd
round[ 2].is_box   51660495435395031 4fb86e401922521
round[ 2].ik_sch   2541fe719bf500258813bbd55a721c0a
round[ 2].ik_add   7427fae4d8a695269ce83d315be0392b
round[ 3].istart   cfb4dbedf4093808538502ac33de185c
round[ 3].is_row   cfde0208f4b418ac5309db5c338538ed
round[ 3].is_box   5f9c6abfbac634aa50409fa766677653
round[ 3].ik_sch   f01afafee7a82979d7a5644ab3afe640
round[ 3].ik_add   af8690415d6e1dd387e5fbedd5c89013
```

```
round[ 4].istart   78e2acce741ed5425100c5e0e23b80c7
round[ 4].is_row   783bc54274e280e0511eacc7e200d5ce
round[ 4].is_box   c14907f6ca3b3aa070e9aa313b52b5ec
round[ 4].ik_sch   7ccff71cbeb4fe5413e6bbf0d261a7df
round[ 4].ik_add   bd86f0ea748fc4f4630f11c1e9331233
round[ 5].istart   d6f3d9dda6279bd1430d52a0e513f3fe
round[ 5].is_row   d61352d1a6f3f3a04327d9fee50d9bdd
round[ 5].is_box   4a824851c57e7e47643de50c2af3e8c9
round[ 5].ik_sch   45f5a66017b2d387300d4d33640a820a
round[ 5].ik_add   0f77ee31d2ccadc05430a83f4ef96ac3
round[ 6].istart   beb50aa6cff856126b0d6aff45c25dc4
round[ 6].is_row   bec26a12cfb55dff6bf80ac4450d56a6
round[ 6].is_box   5aa858395fd28d7d05e1a38868f3b9c5
round[ 6].ik_sch   0bdc905fc27b0948ad5245a4c1871c2f
round[ 6].ik_add   5174c8669da98435a8b3e62ca974a5ea
round[ 7].istart   f6e062ff507458f9be50497656ed654c
round[ 7].is_row   f6ed49f950e06576be74624c565058ff
round[ 7].is_box   d653a4696ca0bc0f5acaab5db96c5e7d
round[ 7].ik_sch   3de23a75524775e727bf9eb45407cf39
round[ 7].ik_add   ebb19e1c3ee7c9e87d7535e9ed6b9144
round[ 8].istart   d22f0c291ffe031a789d83b2ecc5364c
round[ 8].is_row   d2c5831a1f2f36b278fe0c4cec9d0329
round[ 8].is_box   7f074143cb4e243ec10c815d8375d54c
round[ 8].ik_sch   c656827fc9a799176f294cec6cd5598b
round[ 8].ik_add   b951c33c02e9bd29ae25cdb1efa08cc7
round[ 9].istart   2e6e7a2dafc6eef83a86ace7c25ba934
round[ 9].is_row   2e5bacf8af6ea9e73ac67a34c286ee2d
round[ 9].is_box   c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 9].ik_sch   6de1f1486fa54f9275f8eb5373b8518d
round[ 9].ik_add   aeb65ba974e0f822d73f567bdb64c877
round[10].istart   9cf0a62049fd59a399518984f26be178
round[10].is_row   9c6b89a349f0e18499fda678f2515920
round[10].is_box   1c05f271a417e04ff921c5c104701554
round[10].ik_sch   ae87dff00ff11b68a68ed5fb03fc1567
round[10].ik_add   b2822d81abe6fb275faf103a078c0033
round[11].istart   88db34fb1f807678d3f833c2194a759e
round[11].is_row   884a33781fdb75c2d380349e19f876fb
round[11].is_box   975c66c1cb9f3fa8a93a28df8ee10f63
round[11].ik_sch   1651a8cd0244beda1a5da4c10640bade
round[11].ik_add   810dce0cc9db8172b3678c1e88a1b5bd
round[12].istart   ad9c7e017e55ef25bc150fe01ccb6395
round[12].is_row   adcb0f257e9c63e0bc557e951c15ef01
round[12].is_box   1859fbc28a1c00a078ed8aadc42f6109
round[12].ik_sch   a573c29fa176c498a97fce93a572c09c
round[12].ik_add   bd2a395d2b6ac438d192443e615da195
round[13].istart   84e1fd6b1a5c946fdf4938977cfbac23
round[13].is_row   84fb386f1ae1ac97df5cfd237c49946b
round[13].is_box   4f63760643e0aa85efa7213201a4e705
round[13].ik_sch   101112131415161718191a1b1c1d1e1f
round[13].ik_add   5f72641557f5bc92f7be3b291db9f91a
round[14].istart   6353e08c0960e104cd70b751bacad0e7
round[14].is_row   63cab7040953d051cd60e0e7ba70e18c
round[14].is_box   00102030405060708090a0b0c0d0e0f0
round[14].ik_sch   000102030405060708090a0b0c0d0e0f
round[14].ioutput  00112233445566778899aabbccddeeff
```

EQUIVALENT INVERSE CIPHER (DECRYPT):

```
round[ 0].iinput   8ea2b7ca516745bfeafc49904b496089
round[ 0].ik_sch   24fc79ccbf0979e9371ac23c6d68de36
round[ 1].istart   aa5ece06ee6e3c56dde68bac2621bebf
round[ 1].is_box   629deca599456db9c9f5ceaa237b5af4
round[ 1].is_row   627bceb9999d5aaac945ecf423f56da5
round[ 1].im_col   e51c9502a5c1950506a61024596b2b07
round[ 1].ik_sch   34f1d1ffbfceaa2ffce9e25f2558016e
round[ 2].istart   d1ed44fd1a0f3f2afa4ff27b7c332a69
round[ 2].is_box   5153862143fb259514920403016695e4
round[ 2].is_row   516604954353950314fb86e401922521
round[ 2].im_col   91a29306cc450d0226f4b5eaef5efed8
round[ 2].ik_sch   5e1648eb384c350a7571b746dc80e684
round[ 3].istart   cfb4dbedf4093808538502ac33de185c
round[ 3].is_box   5fc69f53ba4076bf50676aaa669c34a7
round[ 3].is_row   5f9c6abfbac634aa50409fa766677653
round[ 3].im_col   b041a94eff21ae9212278d903b8a63f6
round[ 3].ik_sch   c8a305808b3f7bd043274870d9b1e331
round[ 4].istart   78e2acce741ed5425100c5e0e23b80c7
round[ 4].is_box   c13baaeccae9b5f6705207a03b493a31
round[ 4].is_row   c14907f6ca3b3aa070e9aa313b52b5ec
round[ 4].im_col   638357cec07de6300e30d0ec4ce2a23c
round[ 4].ik_sch   b5708e13665a7de14d3d824ca9f151c2
round[ 5].istart   d6f3d9dda6279bd1430d52a0e513f3fe
round[ 5].is_box   4a7ee5c9c53de85164f348472a827e0c
round[ 5].is_row   4a824851c57e7e47643de50c2af3e8c9
round[ 5].im_col   ca6f71058c642842a315595fdf54f685
round[ 5].ik_sch   74da7ba3439c7e50c81833a09a96ab41
round[ 6].istart   beb50aa6cff856126b0d6aff45c25dc4
round[ 6].is_box   5ad2a3c55fe1b93905f3587d68a88d88
round[ 6].is_row   5aa858395fd28d7d05e1a38868f3b9c5
round[ 6].im_col   ca46f5ea835eab0b9537b6dbb221b6c2
round[ 6].ik_sch   3ca69715d32af3f22b67ffade4ccd38e
round[ 7].istart   f6e062ff507458f9be50497656ed654c
round[ 7].is_box   d6a0ab7d6cca5e695a6ca40fb953bc5d
round[ 7].is_row   d653a4696ca0bc0f5acaab5db96c5e7d
round[ 7].im_col   2a70c8da28b806e9f319ce42be4baead
round[ 7].ik_sch   f85fc4f3374605f38b844df0528e98e1
round[ 8].istart   d22f0c291ffe031a789d83b2ecc5364c
round[ 8].is_box   7f4e814ccb0cd543c175413e8307245d
round[ 8].is_row   7f074143cb4e243ec10c815d8375d54c
round[ 8].im_col   f0073ab7404a8a1fc2cba0b80df08517
round[ 8].ik_sch   de69409aef8c64e7f84d0c5fcfab2c23
round[ 9].istart   2e6e7a2dafc6eef83a86ace7c25ba934
round[ 9].is_box   c345bdfa1bc799e1a2dcaab0a857b728
round[ 9].is_row   c357aae11b45b7b0a2c7bd28a8dc99fa
round[ 9].im_col   3225fe3686e498a32593c1872b613469
round[ 9].ik_sch   aed55816cf19c100bcc24803d90ad511
round[10].istart   9cf0a62049fd59a399518984f26be178
round[10].is_box   1c17c554a4211571f970f24f0405e0c1
round[10].is_row   1c05f271a417e04ff921c5c104701554
round[10].im_col   9d1d5c462e655205c4395b7a2eac55e2
round[10].ik_sch   15c668bd31e5247d17c168b837e6207c
round[11].istart   88db34fb1f807678d3f833c2194a759e
round[11].is_box   979f2863cb3a0fc1a9e166a88e5c3fdf
round[11].is_row   975c66c1cb9f3fa8a93a28df8ee10f63
round[11].im_col   d24bfb0e1f997633cfce86e37903fe87
round[11].ik_sch   7fd7850f61cc991673db890365c89d12
```

```
round[12].istart   ad9c7e017e55ef25bc150fe01ccb6395
round[12].is_box   181c8a098aed61c2782ffba0c45900ad
round[12].is_row   1859fbc28a1c00a078ed8aadc42f6109
round[12].im_col   aec9bda23e7fd8aff96d74525cdce4e7
round[12].ik_sch   2a2840c924234cc026244cc5202748c4
round[13].istart   84e1fd6b1a5c946fdf4938977cfbac23
round[13].is_box   4fe0210543a7e706efa476850163aa32
round[13].is_row   4f63760643e0aa85efa7213201a4e705
round[13].im_col   794cf891177bfd1ddf67a744acd9c4f6
round[13].ik_sch   1a1f181d1e1b1c191217101516131411
round[14].istart   6353e08c0960e104cd70b751bacad0e7
round[14].is_box   0050a0f04090e03080d02070c01060b0
round[14].is_row   00102030405060708090a0b0c0d0e0f0
round[14].ik_sch   000102030405060708090a0b0c0d0e0f
round[14].ioutput  00112233445566778899aabbccddeeff
```

Appendix D - References

[1] AES page available via http://www.nist.gov/CryptoToolkit.[4]

[2] Computer Security Objects Register (CSOR): http://csrc.nist.gov/csor/.

[3] J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999, available at [1].

[4] J. Daemen and V. Rijmen, *The block cipher Rijndael*, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp. 288-296.

[5] B. Gladman's AES related home page http://fp.gladman.plus.com/cryptography_technology/.

[6] A. Lee, NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, National Institute of Standards and Technology, November 1999.

[7] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997, p. 81-83.

[8] J. Nechvatal, et. al., *Report on the Development of the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, October 2, 2000, available at [1].

[4] A complete set of documentation from the AES development effort – including announcements, public comments, analysis papers, conference proceedings, etc. – is available from this site.

An ASIC Implementation of the AES SBoxes*

Johannes Wolkerstorfer[1], Elisabeth Oswald[1], and Mario Lamberger[2]

[1] Institute for Applied Information Processing and Communications,
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Johannes.Wolkerstorfer@iaik.at, http://www.iaik.at

[2] Department of Mathematics
Graz University of Technology, Steyrergasse 30, A-8010 Graz, Austria

Abstract. This article presents a hardware implementation of the S-Boxes from the Advanced Encryption Standard (AES). The SBoxes substitute an 8-bit input for an 8-bit output and are based on arithmetic operations in the finite field $GF(2^8)$. We show that a calculation of this function and its inverse can be done efficiently with combinational logic. This approach has advantages over a straight-forward implementation using read-only memories for table lookups. Most of the functionality is used for both encryption and decryption. The resulting circuit offers low transistor count, has low die-size, is convenient for pipelining, and can be realized easily within a semi-custom design methodology like a standard-cell design. Our standard cell implementation on a 0.6 μm CMOS process requires an area of only 0.108 mm^2 and has delay below 15 ns which equals a maximum clock frequency of 70 MHz. These results were achieved without applying any speed optimization techniques like pipelining.

Keywords: Advanced Encryption Standard (AES), finite field arithmetic, inversion, Application Specific Integrated Circuit (ASIC), standard-cell design, Very Large Scale Integration (VLSI), scalability, pipelining.

1 Introduction

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm. It will become a FIPS standard in Fall 2001 [1]. AES will replace the DES-algorithm in the coming years since it offers higher levels of security. AES supports key lengths of 128, 192, and 256 bits. It operates on 128-bit data blocks. The major building blocks of the AES algorithm are the non-linear SBoxes (SubByte-operation) and the MixColumn-operation. Both are based on finite field arithmetic and have an inverse function which is used for decryption.

* The work described originates from the European Commission funded Project *Secure Terminal IC (SETIC)* established under contract IST-2000-25167 resp. *Crypto Module with USB Interface (USB_CRYPT)* established under contract IST-2000-25169 in the Information Society Technologies (IST) Program.

B. Preneel (Ed.): CT-RSA 2002, LNCS 2271, pp. 67–78, 2002.
© Springer-Verlag Berlin Heidelberg 2002

```
AESROUND () {
    SubByte(State);
    ShiftRow(State);
    MixColumn(State);
    AddRoundKey(State, RoundKey);
}
```

Fig. 1. Round function of the AES-algorithm

The AES-algorithm's operations are performed on a two-dimensional array of bytes called the *State*. The State consists of four columns and four rows of bytes. For both encryption and decryption, the AES-algorithm uses a round function that is composed of four different transformations which modify the State (see Fig. 1). First, the SubByte-function substitutes all bytes of the State using a lookup-table called SBox. SBox-table entries are calculated by inversion in the finite field GF(2^8) followed by a short final transformation. Second, the rows of the State are shifted by different offsets (ShiftRow-function). The MixColumn-function scrambles the columns of the State by multiplying a finite field constant. An addition of the State with the Roundkey – which is derived from the input key – concludes the round function which is executed ten times when 128-bit keys are used. The RoundKey is calculated by operations which are similar to those of the round function and require the SBox functionality too.

The efficiency of an AES hardware implementation in terms of die-size, throughput, and power consumption is mainly determined by the implementation of the MixColumn-operation and the SBoxes. The remaining operations are trivial: ShiftRow is a simple cyclic shift, and AddRoundKey is a XOR-operation of the State and the RoundKey. Up to 20 instances of AES-SBoxes are used to realize hardware for the AES round function. The exact number of SBoxes depends on the architecture's degree of parallelism and is determined by throughput requirements and the desired clock frequency. In case that the AES-module should also decrypt data, it has to be taken into account that the SBoxes used for decryption have a different functionality. The number of SBoxes and their style of implementation has important influence on the size and the speed of an AES hardware. For this reason, V. Rijmen (one of the AES inventors) suggests in [4] an alternative method for the computation of the AES-SBox. It consists essentially of a replacement of the SBox lookup-table by an efficient combinational logic for the computation of the inverse elements in GF(2^8). Therefore, another representation of the finite field GF(2^8) is used. This representation leads to an efficient implementation of the finite field arithmetic and was investigated in connection with the implementation of error correcting codes in C. Paar [7], Soljanin et al. [5], and Mastrovito [6]. In contrast to V. Rijmen's original proposal which additionally suggests the optimal normal basis representation of finite field elements (for a definition see [3]) we use the polynomial representation of finite field elements. The benefit of our method is that we have a far more flexible hardware architecture (in comparison to the possible architectures with a straightforward SBox implementation) without the necessity to do complex conversions from one

representation (of finite field elements) to another. The main advantages of our architecture are:

- lower transistor count and die-size than a ROM-based approach,
- a short critical path to achieve a high operational frequency,
- easier implementation within a semi-custom design methodology since all computations can be done with standard-cells,
- flexibility for speed optimization: pipelining techniques can trade throughput for latency.
- suitability for a full-custom implementation: a few leaf-cells using an appropriate logic-style could increase speed or decrease power consumption.

The remainder of this article provides the mathematical background of the finite field arithmetic and the computation of the AES SBoxes in Sect. 2. The building blocks of an SBox and the according formulas are given in Sect. 3. Section 4 presents the implementation.

2 Mathematical Background

This article uses the same notation and conventions as the AES specification [1]. All notations and mathematical operations required for the SBox-operation are presented in a condensed form.

Bytes. The basic data unit of AES are Bytes $a = \{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\}$ each holding eight bits. A Byte can be interpreted as an element of the Galois-Field GF(2^8) in polynomial representation:

$$a(x) = \sum_{i=0}^{7} a_i x^i = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0.$$

The coefficients a_i of a polynomial $a(x)$ are bits. Bytes can be written in different notations. For example, the binary value $\{01100011\}$ is $\{63\}$ in hexadecimal notation and represents the polynomial $x^6 + x^5 + x + 1$.

Addition. The addition of two Bytes representing polynomials $a(x), b(x) \in GF(2^8)$ is achieved by adding their corresponding coefficients modulo 2 which is a XOR-operation usually denoted with $\oplus$.

$$a(x) \oplus b(x) = \sum_{i=0}^{7} a_i x^i \oplus \sum_{i=0}^{7} b_i x^i = \sum_{i=0}^{7} (a_i \oplus b_i) x^i \tag{1}$$

The additive inverse of a Byte is the Byte itself: $-b(x) = b(x)$ and therefore subtraction is identical with addition: $a(x) - b(x) = a(x) + b(x)$.

Multiplication. The multiplication of $a(x), b(x) \in GF(2^8)$ – denoted with $a(x) \otimes b(x)$ – requires an irreducible polynomial of degree 8. For the AES-algorithm it is defined as

$$m(x) = x^8 + x^4 + x^3 + x + 1 = 1\{00011011\} \text{ (bin)} = 1\{1b\} \text{ (hex)}.$$

The multiplication $q(x) = a(x) \cdot b(x)$ in GF(2^8) is done by multiplying the polynomials $a(x)b(x)$ which yields a polynomial $p(x)$ with degree less than 15. This step is followed by a modular reduction step $q(x) = p(x)\ mod\ m(x)$ to ensure that the result is an element of GF(2^8).

A convenient method to multiply in the finite field GF(2^8) is to generate eight partial products: $P_i(x) = a(x) \cdot x^i$ and to add those partial products where the according bit b_i of the multiplier $b(x)$ is 1: $q(x) = \sum_{i=0}^{7} P_i b_i$. The partial products can be calculated efficiently by iterating a multiplication by x: $P_i(x) = P_{i-1}(x) \cdot x\ mod\ m(x), P_0(x) = a(x)$. A multiplication by x is termed *xtimes* and is given by

$$q(x) = xtimes(a) = a(x)x\ mod\ m(x) \tag{2}$$

$$q_0 = a_7, \quad q_1 = a_0 \oplus a_7, \quad q_2 = a_1, \quad q_3 = a_2 \oplus a_7$$
$$q_4 = a_3 \oplus a_7, \quad q_5 = a_4, \quad q_6 = a_5, \quad q_7 = a_6.$$

Xtimes can be implemented by a shift left operation of the input Byte a and a conditional addition of the irreducible polynomial $m(x)$ if the most significant bit (a_7) of a is set. This ensures a Byte as result.

Inversion. The multiplicative inverse a^{-1} of an element $a \in GF(2^8)$ has the property that $\forall a \in GF(2^8) \setminus \{0\} : a \otimes a^{-1} = \{1\}$. Calculating the inverse of a Byte is even more costly than multiplying Bytes. A widely used algorithm for inversion is the *extended Euclidean algorithm* described in [2]. Unfortunately, this algorithm is not suitable for a hardware implementation.

2.1 GF(2^8) as an Extension of GF(2^4)

Usually, the field GF(2^8) is seen as a field extension of GF(2) and therefore its elements can be represented as Bytes. An isomorphic – but for hardware implementations far better suited – representation is to see the field GF(2^8) as a quadratic extension of the field GF(2^4). In this case, an element $a \in$ GF(2^8) is represented as a linear polynomial with coefficients in GF(2^4),

$$a \cong a_h x + a_l, \qquad a \in \text{GF}(2^8), \qquad a_h, a_l \in \text{GF}(2^4) \tag{3}$$

and will be denoted by the pair $[a_h, a_l]$. Both coefficients of such a polynomial have four bits. All mathematical operations applied to elements of GF(2^8) can also be computed in this representation which we call *two-term polynomials*. Two-term polynomials are added by addition of their corresponding coefficients

$$(a_h x + a_l) \oplus (b_h x + b_l) = (a_h \oplus b_h)x + (a_l \oplus b_l). \tag{4}$$

Multiplication and inversion of two-term polynomials require a modular reduction step to ensure that the result is a two-term polynomial too. The irreducible polynomial needed for the modular reduction is given by

$$n(x) = x^2 + \{1\}x + \{e\}. \tag{5}$$

The coefficients of $n(x)$ are elements in GF(2^4) and are written in hexadecimal notation. Their particular values are chosen to optimize the finite field arithmetic.

The multiplication of two-term polynomials involves multiplication of elements in GF(2^4) which requires an irreducible polynomial of degree 4 which is given by

$$m_4(x) = x^4 + x + 1. \tag{6}$$

Deriving formulas for multiplication in GF(2^4) is similar to Byte-multiplication. Multiplication in GF(2^4) is given by

$$q(x) = a(x) \otimes b(x) = a(x) \cdot b(x) \; mod \; m_4(x), \quad a(x), b(x), q(x) \in \mathrm{GF}(2^4) \tag{7}$$

$$\begin{aligned} a_A &= a_0 \oplus a_3, \qquad a_B = a_2 \oplus a_3 \\ q_0 &= a_0 b_0 \oplus a_3 b_1 \oplus a_2 b_2 \oplus a_1 b_3 \qquad q_1 = a_1 b_0 \oplus a_A b_1 \oplus a_B b_2 \oplus (a_1 \oplus a_2) b_3 \\ q_2 &= a_2 b_0 \oplus a_1 b_1 \oplus a_A b_2 \oplus a_B b_3 \quad q_3 = a_3 b_0 \oplus a_2 b_1 \oplus a_1 b_2 \oplus a_A b_3. \end{aligned}$$

Squaring in GF(2^4) is a special case of multiplication and is given by

$$q(x) = a(x)^2 \; mod \; m_4(x), \qquad q(x), a(x) \in \mathrm{GF}(2^4) \tag{8}$$

$$q_0 = a_0 \oplus a_2, \qquad q_1 = a_2, \qquad q_2 = a_1 \oplus a_3, \qquad q_3 = a_3.$$

The inverse a^{-1} of an element $a \in GF(2^4)$ can be derived by solving the equation $a(x) \cdot a^{-1} \; mod \; m_4(x) = 1$ as follows

$$q(x) = a(x)^{-1} \; mod \; m_4(x), \qquad q(x), a(x) \in \mathrm{GF}(2^4) \tag{9}$$

$$\begin{aligned} a_A &= a_1 \oplus a_2 \oplus a_3 \oplus a_1 a_2 a_3 \\ q_0 &= a_A \oplus a_0 \oplus a_0 a_2 \oplus a_1 a_2 \oplus a_0 a_1 a_2 \\ q_1 &= a_0 a_1 \oplus a_0 a_2 \oplus a_1 a_2 \oplus a_3 \oplus a_1 a_3 \oplus a_0 a_1 a_3 \\ q_2 &= a_0 a_1 \oplus a_2 \oplus a_0 a_2 \oplus a_3 \oplus a_0 a_3 \oplus a_0 a_2 a_3 \\ q_3 &= a_A \oplus a_0 a_3 \oplus a_1 a_3 \oplus a_2 a_3. \end{aligned}$$

The concatenation of two bits $a_i a_j$ in Equations 7 and 9 represents a binary multiplication which is an AND-operation. In contrast to inversion in GF(2^8), inversion in GF(2^4) is suitable for a hardware implementation using combinational logic since all Boolean equations depend only on four input bits.

Inversion of Two-Term Polynomials. Inversion of two-term polynomials is the equivalent operation to inversion in GF(2^8). A multiplication of a two-term polynomial with its inverse yields the 1-element of the field: $(a_h x + a_l) \otimes (a'_h x +$

$a'_l) = \{0\}x + \{1\}, \quad a_h, a_l, a'_h, a'_l \in GF(2^4)$. From this definition the formula for inversion can be derived:

$$(a_h x + a_l)^{-1} = a'_h x + a'_l = (a_h \otimes d)x + (a_h \oplus a_l) \otimes d \qquad (10)$$
$$d = ((a_h^2 \otimes \{e\}) \oplus (a_h \otimes a_l) \oplus a_l^2)^{-1}.$$

Inversion of two-term polynomials involves only operations in GF(2^4) which are suitable for a hardware implementation using combinational logic. Most of the functionality is used to calculate the term d which is used to calculate both coefficients of the inverted two-term polynomial.

Transition between Representations of GF(2^8). The finite field GF(2^8) is isomorphic to the finite field GF($(2^4)^2$) which means that for each element in GF(2^8) there exists exactly one element in GF($(2^4)^2$). The bijection from an element $a \in GF(2^8)$ to a two-term polynomial $a_h x + a_l$ where $a_h, a_l \in GF(2^4)$ is given by the function *map*:

$$a_h x + a_l = map(a), \quad a_h, a_l \in \mathrm{GF}(2^4), \quad a \in \mathrm{GF}(2^8) \qquad (11)$$

$$a_A = a_1 \oplus a_7, \quad a_B = a_5 \oplus a_7, \quad a_C = a_4 \oplus a_6$$
$$a_{l0} = a_C \oplus a_0 \oplus a_5, \quad a_{l1} = a_1 \oplus a_2, \quad a_{l2} = a_A, \quad a_{l3} = a_2 \oplus a_4$$
$$a_{h0} = a_C \oplus a_5, \quad a_{h1} = a_A \oplus a_C, \quad a_{h2} = a_B \oplus a_2 \oplus a_3, \quad a_{h3} = a_B.$$

The inverse transformation (map^{-1}) converts two-term polynomials $a_h x + a_l$, $a_h, a_l \in GF(2^4)$ back into elements $a \in GF(2^8)$. It is given by

$$a = map^{-1}(a_h x + a_l), \quad a \in \mathrm{GF}(2^8), \quad a_h, a_l \in \mathrm{GF}(2^4) \qquad (12)$$

$$a_A = a_{l1} \oplus a_{h3}, \quad a_B = a_{h0} \oplus a_{h1},$$
$$a_0 = a_{l0} \oplus a_{h0}, \quad a_1 = a_B \oplus a_{h3}$$
$$a_2 = a_A \oplus a_B, \quad a_3 = a_B \oplus a_{l1} \oplus a_{h2}$$
$$a_4 = a_A \oplus a_B \oplus a_{l3}, \quad a_5 = a_B \oplus a_{l2}$$
$$a_6 = a_A \oplus a_{l2} \oplus a_{l3} \oplus a_{h0}, \quad a_7 = a_B \oplus a_{l2} \oplus a_{h3}$$

Both transformations can be derived by following the procedure given in C. Paar's PhD thesis [7]. They differ from the transformations given in [5] because the irreducible polynomial $m(x)$ for GF(2^8) is different.

3 SBox Building Blocks

The SubByte transformation operates independently on each Byte of the State using a substitution table (SBox). An AES-SBox is composed of two transformations:

1. Calculate the multiplicative inverse in the finite field GF(2^8). The element $\{00\}$ is mapped to itself. Table 1 presents the inversion.

Table 1. Inversion of a Byte $\{xy\} \in GF(2^8)$ in hexadecimal notation.

$x \backslash y$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	00	01	8d	f6	cb	52	7b	d1	e8	4f	29	c0	b0	e1	e5	c7
1	74	b4	aa	4b	99	2b	60	5f	58	3f	fd	cc	ff	40	ee	b2
2	3a	6e	5a	f1	55	4d	a8	c9	c1	0a	98	15	30	44	a2	c2
3	2c	45	92	6c	f3	39	66	42	f2	35	20	6f	77	bb	59	19
4	1d	fe	37	67	2d	31	f5	69	a7	64	ab	13	54	25	e9	09
5	ed	5c	05	ca	4c	24	87	bf	18	3e	22	f0	51	ec	61	17
6	16	5e	af	d3	49	a6	36	43	f4	47	91	df	33	93	21	3b
7	79	b7	97	85	10	b5	ba	3c	b6	70	d0	06	a1	fa	81	82
8	83	7e	7f	80	96	73	be	56	9b	9e	95	d9	f7	02	b9	a4
9	de	6a	32	6d	d8	8a	84	72	2a	14	9f	88	f9	dc	89	9a
a	fb	7c	2e	c3	8f	b8	65	48	26	c8	12	4a	ce	e7	d2	62
b	0c	e0	1f	ef	11	75	78	71	a5	8e	76	3d	bd	bc	86	57
c	0b	28	2f	a3	da	d4	e4	0f	a9	27	53	04	1b	fc	ac	e6
d	7a	07	ae	63	c5	db	e2	ea	94	8b	c4	d5	9d	f8	90	6b
e	b1	0d	d6	eb	c6	0e	cf	ad	08	4e	d7	e3	5d	50	1e	b3
f	5b	23	38	34	68	46	03	8c	dd	9c	7d	a0	cd	1a	41	1c

2. Apply the affine transformation which is given by Equation 13.

$$q = \text{aff_trans}(a) \tag{13}$$

$$\begin{aligned} a_A &= a_0 \oplus a_1, a_B = a_2 \oplus a_3, \\ a_C &= a_4 \oplus a_5, a_D = a_6 \oplus a_7 \\ q_0 &= \overline{a_0} \oplus a_C \oplus a_D \\ q_1 &= \overline{a_5} \oplus a_A \oplus a_D \\ q_2 &= a_2 \oplus a_A \oplus a_D \\ q_3 &= a_7 \oplus a_A \oplus a_B \\ q_4 &= a_4 \oplus a_A \oplus a_B \\ q_5 &= \overline{a_1} \oplus a_B \oplus a_C \\ q_6 &= \overline{a_6} \oplus a_B \oplus a_C \\ q_7 &= a_3 \oplus a_C \oplus a_D. \end{aligned}$$

$$q = \text{aff_trans}^{-1}(a) \tag{14}$$

$$\begin{aligned} a_A &= a_0 \oplus a_5, a_B = a_1 \oplus a_4, \\ a_C &= a_2 \oplus a_7, a_D = a_3 \oplus a_6 \\ q_0 &= \overline{a_5} \oplus a_C \\ q_1 &= a_0 \oplus a_D \\ q_2 &= \overline{a_7} \oplus a_B \\ q_3 &= a_2 \oplus a_A \\ q_4 &= a_1 \oplus a_D \\ q_5 &= a_4 \oplus a_C \\ q_6 &= a_3 \oplus a_A \\ q_7 &= a_6 \oplus a_B. \end{aligned}$$

Overlined bits in Equation 13 and 14 denote inverted bits. Decryption requires the inverse function of SubByte (*InvSubByte*) which reverses the SBox-operation by applying the inverse affine transformation first (Equation 14). Then, the multiplicative inverse in the finite field GF(2^8) is calculated.

Inversion in the finite field GF(2^8) is needed to calculate the SubByte-function as well as InvSubByte. It makes sense, to merge the encryption SBox with the decryption SBox in order to reuse the finite field inversion circuit for decryption. Figure 2 depicts this approach. The control signal *enc* switches between encryption and decryption. If encryption is chosen (*enc*=1), the inverse affine transformation (*aff_trans*$^{-1}$) is bypassed and the input a is directly fed

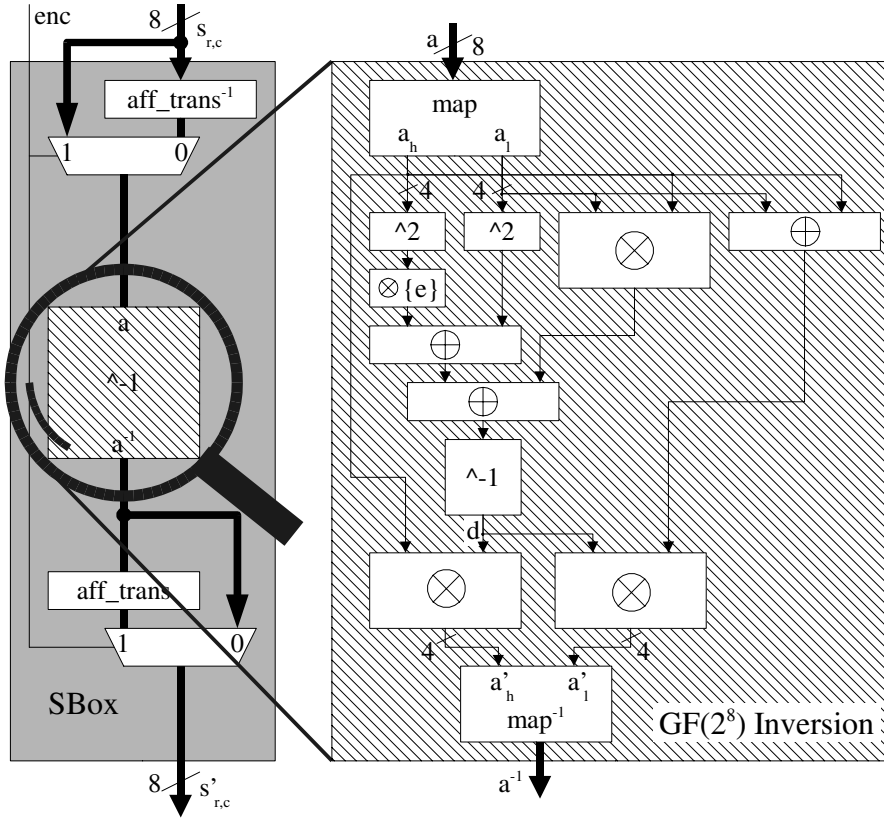


Fig. 2. Architecture of the AES-SBox

into the inversion circuit. The output of the inversion circuit is modified by the affine transformation block which calculates the result of the SubByte-function. During decryption (*enc*=0), the inverse affine transformation is active and the affine transformation is bypassed to calculate InvSubByte. The delay for encryption and decryption is essentially the same because the circuit's complexity for the affine transformation and its inverse are equal.

The circuit for inversion of elements in GF(2^8) covers most of the SBox functionality. In our approach the inversion is calculated with combinational logic and is based on Equation 10 which operates in GF(2^4). The operations occurring in this equation correspond to the function blocks shown in Fig. 2. Furthermore, this function block has to convert data from GF(2^8) to two-term polynomials and vice versa. The blocks *map* resp. map^{-1} provide this functionality based on Equation 11 and 12. Addition of elements in GF(2^4) is accomplished by a bitwise XOR-operation. Squaring relies on Equation 9. Multiplication of an element in GF(2^4) with the constant $\{e\}$ is given by

$$q = a \otimes \{e\} \tag{15}$$

$$a_A = a_0 \oplus a_1, \;\; a_B = a_2 \oplus a_3$$

$$q_0 = a_1 \oplus a_B$$

$$q_1 = a_A$$

$$q_2 = a_A \oplus a_2$$

$$q_3 = a_A \oplus a_B.$$

Multiplication and inversion in GF(2^4) are the most complex function blocks and rely on Equations 7 and 9.

4 Implementation

Our implementation is based on the architecture described in Sect. 3. It has a maskable affine transformation block, a maskable inverse affine transformation block and a block which calculates the inverse in the finite field GF(2^8). Most of the functionality can be implemented with XOR-gates. Additionally, inverters, AND-gates, and 2-to-1 multiplexers are required, but they can be neglected for performance analysis purposes.

Table 2. Complexity of the SBox

block	d_{XOR}	XORs	instances	sum XORs
aff_trans	3	16	1	16
aff_trans^{-1}	2	12	1	12
map	2	11	1	11
map^{-1}	3	15	1	15
$\oplus$	1	4	3	12
$\wedge^2$	1	2	2	4
$\otimes\{e\}$	2	5	1	5
$\otimes$	2	12	3	36
$\wedge^{-1}$	3	12	1	12
	Max 15			Sum 123

Table 3. Pipelining of the SBox

stages	flipflops	frequency	area
0	0	100%	100%
1	12	178%	111%
3	28	205%	151%

Table 2 lists the resources of all blocks – measured in number of XORs. The overall amount of gates are 123 XOR-gates with two inputs, 16 2-to-1 multiplexers and a dozen of inverters and AND-gates. If XOR-gates with three inputs are available, the number of gates can be reduced. The delay of the blocks is measured in numbers of XOR-gates in series (d_{XOR}). The critical path for encryption is composed of 15 XOR-gates in series. For decryption it is 14 because the inverse of the affine transformation has lower complexity. XOR-gates with three inputs will shorten the critical path and improve performance.

A feature that can be exploited to gain higher throughput is pipelining. Pipelining is a technique which subdivides the critical path by insertion of storing elements (flipflops). Subdividing the SBox functionality into a number of stages

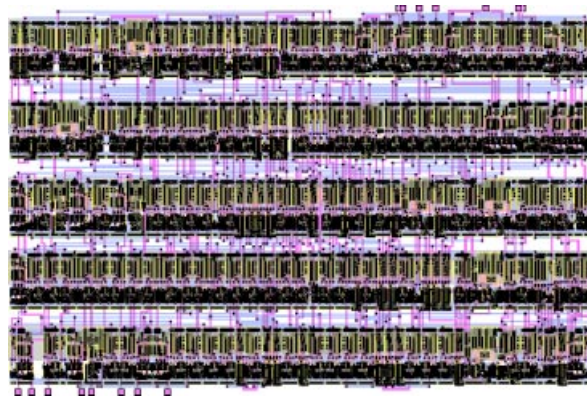


Fig. 3. Layout of the AES-SBox

is easy to accomplish since flipflops can be inserted nearly anywhere when SBoxes are implemented with combinational logic. Pipelining introduces latency but the additional clock cycles are made up by an increased clock frequency as shown in Table 3. This technique will offer best results if the number of SBox instances used for an AES implementation is kept low (e.g. 4), otherwise latency will consume more time than it is saved by shortening the critical path.

Our implementation of the AES-SBox which combines the SubByte-function and InvSubByte-function from the AES algorithm is a standard-cell circuit on a 0.6 μm CMOS process from AMS using two metal layers. It has an area of 0.108 mm^2 and contains 1624 transistors (406 NAND equivalents) when no pipelining is used. A layout is shown in Fig. 3. Layout simulation with typical mean parameters considering parasitics yields a delay below 14.2 ns which equals a maximum clock frequency of 70 MHz for both encryption and decryption. A pipelined version with one stage is slightly bigger (0.120 mm^2), contains nearly 2000 transistors (500 NAND equivalents) and yields a delay below 8 ns (125 MHz). Further increasing the number of pipeline stages will not improve throughput that strong and has a worse ratio of performance benefit to area penalty. It should be considered when the maximum clock frequency is of utmost interest.

For comparison, a lookup-table based approach requires a 4k-bit ROM to store the 256 8-bit entries of SubByte and InvSubByte. An implementation on the same process technology uses about 0.200 mm^2 and has an estimated maximum frequency of 100 MHz [9].

For an SBox implementation using a full-custom design methodology we suggest to use a differential logic style [8]. The logic functions of an SBox based on combinational logic are dominated by XOR-gates and differential XOR-gates offer good performance and have a moderate transistor count. We assume that a full-custom design could halve the required chip area because an SBbox is a small module and does not require the excessive driving capability offered by

standard cells. Output transistors of gates can be dimensioned smaller and this will in turn make it possible to scale all transistors down without deteriorating performance. At least three leaf cells are needed: a XOR-gate, an AND-gate and an inverter. To develop these cells will be an rewarding task if the resulting performance gain and area saving is pictured.

5 Related Work

Several AES-implementations have been presented recently. For comparison with our work, only ASIC circuits [11] or circuits exploiting a more efficient finite field arithmetic are of interest [10]. The approach followed in IBM implementation [10] is also based on a conversion of elements in GF(2^8) into two-term polynomials. In contrast to our approach, they calculate the whole round function in this representation. Therefore, they choose the conversion function $map()$ in a way that minimizes the overall gate count. Our primary focus on choosing $map()$ was to minimize the critical path of the complete SBox and secondarily to keep the gate count low. Our $map()$-function has a shorter critical path compared to the IBM implementation, the critical path of $map^{-1}()$ is identical.

6 Conclusion

This article presented an ASIC implementation of the SBoxes from the Advanced Encryption Standard (AES). It is based on finite field arithmetic rather than using lookup-tables. This approach offers higher flexibility. Area requirements can be traded for the maximum clock frequency. The architecture can be easily implemented within a standard-cell design methodology because it completely relies on combinational logic. It is also well suited for a full-custom implementation since it uses only a few leaf cells. We implemented the AES-SBox on 0.6 μm CMOS process with standard-cells. The most promising configuration of design parameters we found is a single stage pipeline architecture. It has a silicon area of 0.12 mm^2 and a maximum clock frequency of 125 MHz. This configuration has a latency of one clock cycle like a ROM based approach. In comparison to ROMs, it offers better performance on smaller area and can even be improved by exploiting better suited logic styles like differential logic.

References

1. NIST, *Advanced Encryption Standard (AES)*, FIPS PUBS 197, National Institute of Standards and Technology, November 2001.
2. A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997.
3. R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, Cambridge, 1986.
4. V. Rijmen, *Efficient Implementation of the Rijndael SBox*, `http://www.esat.ku-leuven.ac.be/~rijmen/rijndael/` .

5. E. Soljanin, R. Urbanke, *An Efficient Architecture for Implementation of a Multiplier and Inverter in GF(2^8)*, Lucent Technologies.
6. E. D. Mastrovito, *VLSI Architectures for Computations in Galois Fields*, PhD thesis, Linköping University, Linköping, Sweden, 1991.
7. C. Paar, *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*, PhD thesis, Universität Essen, 1994.
8. J. B. Kuo, J. H. Lou, *Low-Voltage VLSI Circuits*, John Wiley, New York, Jan. 1999.
9. AMS, *Memory Compiler for Diffusion Programmable ROM in 0.6 μm CMOS*, `http://www.amsint.com/databooks/`.
10. A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, P. Rohatgi, *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic*, Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, France, 2001, to be published in Springer LNCS.
11. I. Verbauwhede, H. Kuo, *Architectural Optimization for a 1.82 Gbits/sec VLSI Implementation of the AES Rijndael Algorithm*, Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, France, 2001, to be published in Springer LNCS.

An Efficient Architecture for the AES Mix Columns Operation

Hua Li Zac Friggstad
huali@cs.uleth.ca zac.friggstad@uleth.ca
Department of Mathematics and Computer Science
University of Lethbridge
Canada T1K 3M4

Abstract ***In this paper, a compact architecture for the AES mix columns operation and its inverse is presented. The hardware implementation is compared with previous work done in this area. We show that our design has a lower gate count than other designs that implement both the forward and the inverse mix columns operation.***

Keywords: AES, cryptography, Galois field, mix columns

I. Introduction

Since the debut of the Advanced Encryption Standard (AES) [1], it has been thoroughly studied by hardware designers with the goal of reducing the area and delay of the hardware implementation of this cryptosystem. This paper presents an implementation of the AES mix column operation. This design proves to be low in gate count as well as outputs the desired result of the operation (forward or inverse) on the same wire array, depending on an input signal that selects between the two. This will eliminate the need for a multiplexor outside of the module to select between the results of the forward and the inverse operation.

This paper is organized in the following way. Section 2 presents the mathematical background of $GF(2^8)$ and the application to this mix columns design. Section 3 presents the hardware implementation. Section 4 compares this design with previous architectures and suggests appropriate uses for the presented design. Section 5 then concludes this paper.

II. Mathematical Background

The results of the mix columns operation are calculated using $GF(2^8)$ operations. Each element of $GF(2^8)$ is a polynomial of degree 7 with coefficients in $GF(2)$ (or equivalently $\mathbb{Z}_2$). Thus, the coefficients of each term of the polynomial can take the value 0 or 1. Given that there are 8 terms in an element of $GF(2^8)$, an element can be represented by bitstring of length 8, where each bit represents a coefficient. We will use the least significant bit to represent the constant of the polynomial, and going from right to left, represent the coefficient of x^i by the bit b_i where b_i is i bits to the left of the least significant bit. For example, the bitstring 10101011 represents $x^7+x^5+x^3+x+1$. For our convenience, a term x^i is found in the expression if the corresponding coefficient is 1. The term is omitted from the expression if the coefficient is 0. Addition of two elements in $GF(2^8)$ is simply accomplished using eight XOR gates to add corresponding bits.

Multiplication of two elements in $GF(2^8)$ requires a bit more work. The multiplication of two elements of $\mathbb{Z}_2$ is simulated with an AND gate. Multiplication in $GF(2^8)$ can then be accomplished by first multiplying each term of the second polynomial with all of the terms of the first polynomial. Each of these products should be added together. If the degree of the new polynomial is greater than 7, then it must be reduced modulo some irreducible polynomial. In the case of AES, the irreducible polynomial is $x^8+x^4+x^3+x+1$. For example:

$$\begin{array}{rr} & x^4+1 \\ \times & x^5+x^4 \\ \hline & x^8+x^4 \\ + & x^9+x^5 \\ \hline x^9 & +x^8+x^5+x^4 \end{array} \tag{1}$$

Since the degree of the result is greater than 7, then we must reduce it modulo $x^8+x^4+x^3+x+1$. This can be accomplished by multiplying the polynomial by x^{i-8}, where i is the degree of the polynomial that is to be reduced. Then, adding the multiplied irreducible polynomial to polynomial to be reduced. Taking this result, we continue this process until the degree of the polynomial we are reducing is not greater than 7. In our example, we perform:

$$\begin{array}{rr} & x^9+x^8+x^5+x^4 \\ + & x^9+x^5+x^4+x^2+x \\ \hline & x^8+x^2+x \\ + & x^8+x^4+x^3+x+1 \\ \hline & x^4+x^3+x^2+1 \end{array} \tag{2}$$

The mix column operation involves multiplying a 4×4 matrix of $GF(2^8)$ values by a column vector of $GF(2^8)$ values. Given a 32-bit input word $w = w_3w_2w_1w_0$ where each w_i is 8-bits, the mix column operation is given as:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} w_3 \\ w_2 \\ w_1 \\ w_0 \end{bmatrix} = \begin{bmatrix} w'_3 \\ w'_2 \\ w'_1 \\ w'_0 \end{bmatrix} \tag{3}$$

Authorized licensed use limited to: University of Guelph. Downloaded on May 12,2022 at 15:08:30 UTC from IEEE Xplore. Restrictions apply.

The inverse of this operation is:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \cdot \begin{bmatrix} w'_3 \\ w'_2 \\ w'_1 \\ w'_0 \end{bmatrix} = \begin{bmatrix} w_3 \\ w_2 \\ w_1 \\ w_0 \end{bmatrix} \quad (4)$$

Where each element of the 4×4 matrices is a hexadecimal representations of the coefficients of an element in $GF(2^8)$. For example, $0E$ represents $x^3 + x^2 + x$.

Since multiplication is distributive over addition in $GF(2^8)$, we can rewrite the multiplication of two elements of $GF(2^8)$ as a linear combination of products of the first element and a single-termed polynomial in $GF(2^8)$. The sum of all of the single-termed polynomials must equal the second element in the original product. For example, we can express $x \cdot 0E$ as $(x \cdot 08) + (x \cdot 04) + (x \cdot 02)$, for any $x \in GF(2^8)$, because $08 + 04 + 02 = 0E$.

Because of the distributivity of $GF(2^8)$ and of matrices, we can rewrite the inverse of the mix columns operation as:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \cdot \begin{bmatrix} w'_3 \\ w'_2 \\ w'_1 \\ w'_0 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} w'_3 \\ w'_2 \\ w'_1 \\ w'_0 \end{bmatrix}$$

$$+ \begin{bmatrix} 0C & 08 & 0C & 08 \\ 08 & 0C & 08 & 0C \\ 0C & 08 & 0C & 08 \\ 08 & 0C & 08 & 0C \end{bmatrix} \cdot \begin{bmatrix} w'_3 \\ w'_2 \\ w'_1 \\ w'_0 \end{bmatrix} \quad (5)$$

Given these facts we note two things. First, for each of the four input bytes, we only need to calculate the products of the byte with 01, 02, 04 and 08. All elements of the matrices can be formed by summing these products. Also, since the inverse matrix can be expressed as the sum of the original matrix and some $\{0C, 08\}$ element matrix, where $0C = 08 + 04$, we see that 04 and 08 are never used in the forward operation, only the inverse.

For some $a \in GF(2^8)$ that is expressed by the bitstring $a_7a_6a_5a_4a_3a_2a_1a_0$, we can efficiently compute $a \times 2$ (or $a \times x$). The resulting value will be $a_6a_5a_4(a_3 + a_7)(a_2 + a_7)a_1(a_0 + a_7)a_7$, where $+$ is addition in $\mathbb{Z}_2$. This can be explained by understanding that multiplication by 02 is essentially multiplication by the single-termed polynomial x. This will result in all degrees of a being incremented by 1. If the resulting polynomial has degree 8, then the irreducible polynomial must be added. If the resulting polynomial is not 8, then nothing is done. We see that each bit in the above expression that is a sum of some a_i and a_7, if a is of degree 7, then the resulting degree is 8, and the irreducible polynomial is added. This will result in $a_i + a_7 = a_i + 1$. If a is not of degree 7, then the resulting degree will not be 8. This will result in $a_i + a_7 = a_i + 0 = a_i$ because 0 is the identity of addition in $\mathbb{Z}_2$.

III. A MIXCOLUMNS ARCHITECTURE

The implementation of the architecture based on the mathematics in the previous section can be presented in four simple modules. Diagrams of the first three modules can be found in Figures 1, 2 and 3.

The multiplication by the term x is simply a one-bit shift to the left, and an addition by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ if the coefficient of the term x^7 of the input is 1. This simple module is shown in Figure 1.

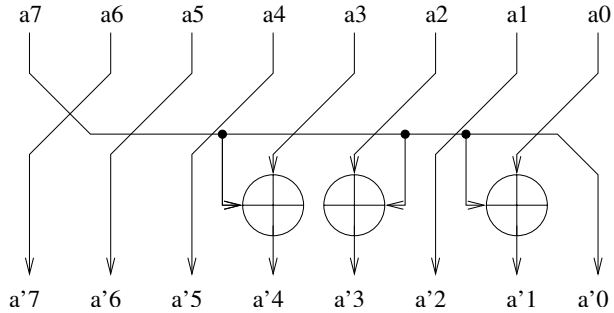


Fig. 1. Multiplication by constant polynomial x (module of $a \times 2$)

To generate the multiplications of $a \times 2^n$, i.e., $a \cdot x^n$, $(0 \leq n \leq 3)$ in $GF(2^8)$, we simply need to cascade various $a \times 02$ (i.e, $a \cdot x$) modules found in Figure 1. Figure 2 shows how these connections are made. The switch gates module in the figure denotes an eight bit array of switch gates, each of which takes the inv line as the base and one of wires in the wire array as the source. This is to propagate the signal of the wire array to the next $x2$ module if the inv line is set. If the line is not set, then logic 0 will be sent to the next $x2$ module from the sink of the switch gates. The interpretation of the function of this gate is that $a \times 2^2$ (i.e, $a \cdot x^2$) and $a \times 2^3$ (i.e, $a \cdot x^3$) for module input a will be calculated if the inverse of the mix columns operation is to be performed. If the operation is simply the forward operation, then multiplication by powers 2 and 3 are not used, and the input to the last two $a \times 2$ modules should be 0. Thus, the output of these two modules is then the identity element 0, and does not affect the XOR gates that follow in the next module.

The module $a \times 2^n$ (Figure 2) can be used to calculate two element multiplication in equations (3) and (4) as illustrated in Figure 3. Each $\oplus$ denotes an eight bit array of XOR gates that XOR the corresponding elements of the two input arrays. The hexadecimal numbers N_L/N_R in Figure 3 that mark the outputs of some of the $\oplus$ gates denote that the value $w_i \times N$ is output from the gate array. N_L is the number N used for encoding, while N_R is the number N used for decoding. We see that $N_R - N_L = t$ for some $t \in \{08, 0C\}$. It should be clearer now that if dec is set, then $N = N_R$, otherwise $N = N_L$.

Finally we can derive the logic circuit to compute the mix column operation as show in Figure 4. Four of the modules found in Figure 3 are used, one for each byte of the input word. We label the outputs from these modules as

Authorized licensed use limited to: University of Guelph. Downloaded on May 12,2022 at 15:08:30 UTC from IEEE Xplore. Restrictions apply.

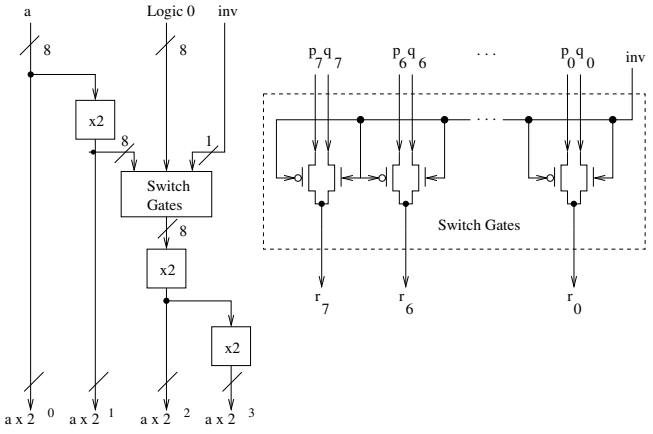


Fig. 2. Multiplications of $a \cdot x^n$ ($0 \leq n \leq 3$) in $GF(2^8)$ (module of $a \times 2^n$)

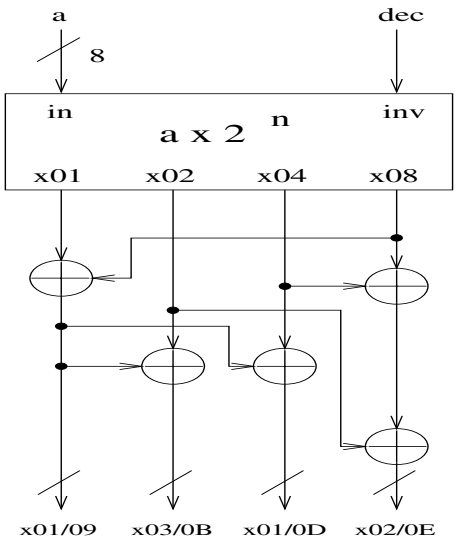


Fig. 3. Matrix product generator (module of product generator).

0109, $030B$, $010D$, and $020E$ where each output represents the product of the input w_i and some number n where $0 \leq i \leq 3$. The number n is the first number in the output label associated with that line if the dec line is not set, otherwise n is the second number. We calculate w'_i by adding, with XOR arrays, the appropriate outputs from the four modules that generate $w_i \times n$ for some $n \in GF(2^8)$. This is given by the following equations:

$$\begin{aligned} w'_3 &= w_3 \times E + w_2 \times B + w_1 \times D + w_0 \times 9 \\ w'_2 &= w_3 \times 9 + w_2 \times E + w_1 \times B + w_0 \times D \\ w'_1 &= w_3 \times D + w_2 \times 9 + w_1 \times E + w_0 \times B \\ w'_0 &= w_3 \times B + w_2 \times D + w_1 \times 9 + w_0 \times E \end{aligned} \quad (6)$$

Though the above equation is for the decode operation, the encode operation requires the exact same wiring of the exact same output from the four modules that generate the $w_i \times n$ outputs. Since these modules have already determined if the forward or the inverse values are output on the wire arrays, the connections described in the above equations will work

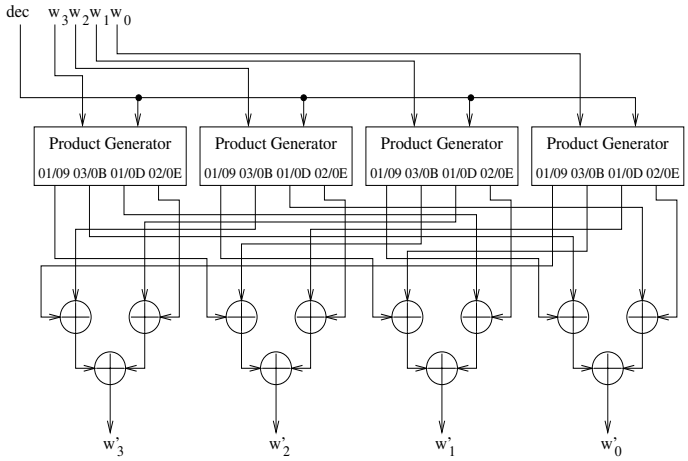


Fig. 4. Top-level view of the mix columns design.

for the forward operation as well. Therefore, the results of the forward operation will be output if the dec line is not set whereas the results of the inverse operation will be output in the same spot if the dec line is set.

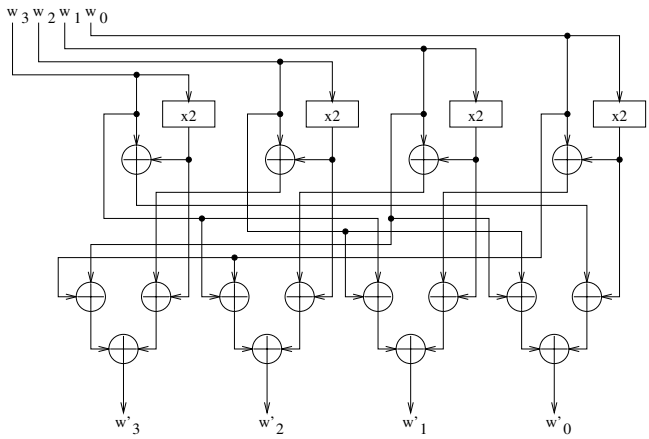


Fig. 5. A mix columns architecture that computes only the forward operation.

We also give a brief description of an architecture for the mix columns design that only computes the results of the forward operation. Some implementations of AES desire only the forward operation, such as the one found in [2]. We make special note of this design because it actually has less than half the amount of gates as in the design in Figure 4. Figure 5 shows the simple architecture of this encode-only module. We have removed the decode line, eight of the multipliers and many XOR arrays that were used to add in the $w_i \times 04$ and $w_i \times 08$ results, as they are not used in the forward operation.

IV. Comparison

The total number of gates in the presented design is $292_{XOR} + 32_{AND}$ and it results in a critical path of $6 \cdot T_{XOR} + T_{AND}$.

In [3], 16 multipliers are used for Mix Columns Operation, with 212 gates each. Replacing these multipliers with four

Authorized licensed use limited to: University of Guelph. Downloaded on May 12,2022 at 15:08:30 UTC from IEEE Xplore. Restrictions apply.

instances of our design (one for each column) can reduce the overall gate count by $2,096$ gates, which is a significant improvement considering that the paper claims $15,493$ gates in its high-performance design (14% of total gates are reduced). We also note that the design in [3] uses composite field arithmetic in the implementation of the s-box. Since the s-box and the mix columns (or its inverse) operations are performed in different clock cycles in the design in [3], and the s-box design has a critical path of at least $15 \cdot T_{XOR}$ (the AND gates were not considered in the analysis in the paper), we see that our design would not be the critical path in the whole AES design.

TABLE I

THE COMPARISONS OF SPACE AND TIME COMPLEXITY OF DIFFERENT MIX-COLUMN ARCHITECTURES FOR ENCRYPTION ONLY

Design	Gates	Critical Path	Results
Proposed Figure 5	140	4 GE	Encode Only
[2]	176	5 GE	Encode Only
[4]	176^+	6^+ GE	Encode Only

Another design that is quite similar to our design is found in [2]. In this paper, only the forward operation is presented, not the inverse. Also, though they obtain the results with the same concept presented in our design, many unnecessary gates are used in the calculation of $w \times 2$ (i.e., $w \cdots x$) because of the multiplexing between the left-shifted result and the addition of the irreducible polynomial into the left-shifted result by using the left-most bit as the selection line. Our design avoids this by using that selection line as a direct XOR into corresponding x^i terms in the irreducible polynomial. This demands a maximum of $3XOR$ gates to multiply by x as opposed to the many that were used in their design when using the selector.

Table 1 illustrates the comparisons made between our design and another designs that implements only forward Mix Column operation, where GE refers to gate equivalents.

V. CONCLUSION

A fast and low complexity architecture for the AES Mix Column and Inverse Mix Column operations is proposed in this paper. The short critical path, small gate count and versatility (between encode and decode) is desirable for most architectural descriptions of AES. The performance comparisons indicate that the proposed mix-column architectures have less complexity than previous relevant work. In particular, the total number of gates in [4] can be reduced 14% off by applying our proposed mix-column architecture.

ACKNOWLEDGMENT

This research work is supported by the funding of Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] N. I. of Standards and Technology, *Federal Information Processing Standard 197, The Advanced Encryption Standard (AES)*, http://csrc.nist.gov/publications/fips/fips197/fips197.pdf, 2001.

[2] H. Kuo, I. Verbauwhede, and P. Schaumont, "A 2.29 gbits/sec, 56 mw non-pipelined rijndael aes encryption ic in a 1.8v, 0.18 um cmos technology." [Online]. Available: citeseer.nj.nec.com/kuo02gbitssec.html

[3] S. Mangard, M. Aigner, and S. Moninikus, "A highly regular and scalable aes hardware architecture," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 483–491, April 2003.

[4] A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, and P. Rohatgi, "Efficient rijndael encryption implementation with composite field arithmetic," in *Proc. Workshop Cryptographic Hardware and Embedded Systems*, ser. CHES, 2001, pp. 171–184.

Authorized licensed use limited to: University of Guelph. Downloaded on May 12,2022 at 15:08:30 UTC from IEEE Xplore. Restrictions apply.