Andrew Scott
E155
Lab 7
November 2, 2015

Lab 7: Advanced Encryption Standard

Introduction

In this lab, I wrote a AES hardware accelerator. The accelerator reads in a 128 bit plaintext message and a 128 bit key via the SPI protocol and then performs the AES encryption, outputting the 128 bit encrypted message via the SPI protocol.

Design and Testing Methodology

There was essentially no hardware (breadboard) design to be done this week, since the only breadboard work to be done was connecting Pi pins to FPGA pins.

For the software (Verilog), I decided to use the datapath/controller design to create a circuit that would execute one round of the encryption algorithm each clock cycle. I achieved this by having a 3mux leading into a register which then fed into a module that performed the AddRoundKey part of the encryption. The 3mux chooses between the first plaintext input, the output from a module performing the MixColumns part of the encryption, and the output from the ShiftRows module part of the encryption (depending on the current round of encryption being performed). The signal controlling the 3mux comes from my controller module which has a finite state machine inside it that keeps track of the current round. The implementations of AddRoundKey and ShiftRows were simple, and the implementations for MixColumns and SubBytes were given (SubBytes just required writing a wrapper function to call the given SubByte module on each of the individual bytes in the plaintext). The AddRoundKey implementation was simple because it is just an XOR with a round key, but generating the round key itself turned out to be quite a bit of work. My key expansion module to generate this round key ended up resembling a datapath on its own, generating the 128 bit key each round (representing 4 executions of the key expansion loop in Figure 11 of the AES standard). This datapath has a 2mux control signal coming from the aes controller module. The point of this 2mux is to choose between the original input key on the first round and the key generated on the previous round for the rest of the rounds. I put a register to hold the current key which has as input the output from the 2mux described before. The first 32 bits of the new key go through a module that performs the `temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]` line from the key expansion pseudocode. I then had a series of XOR gates performing the `w[i] = w[i-Nk] xor temp` line for each of the 32 bit segments of the new key. The resulting new key was then stored in the key register and then used to generate the next key on the next clock cycle. I tested my implementation using the provided testbench, which helped me to find several bugs (one of which was the fact that I wasn't maintaining the value of the encrypted message after encrypting it, which was solved by simply making my registers write-enabled).

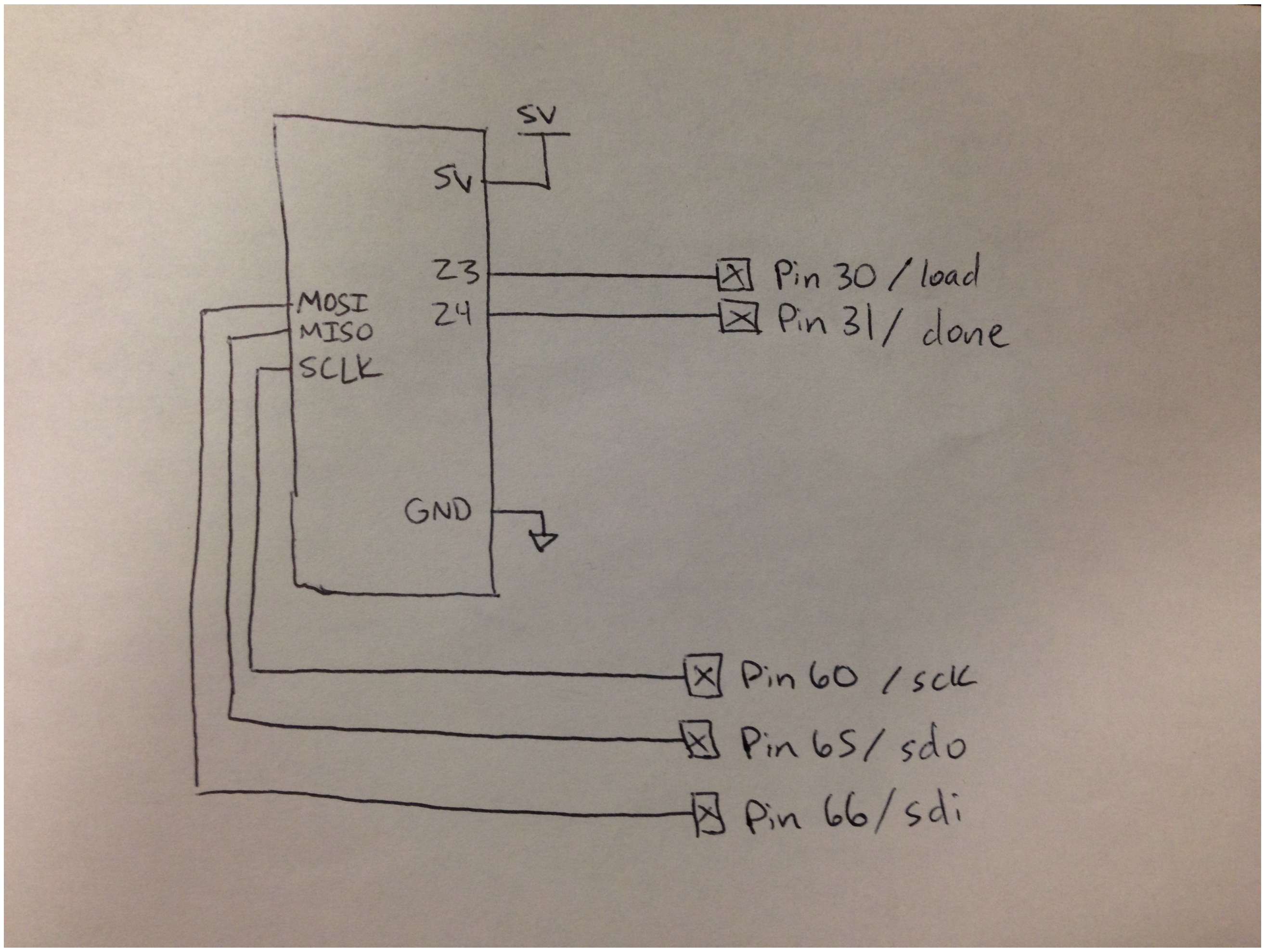
Results and Discussion

I was able to complete the entire lab, and as far as I can tell everything is working properly. Running the provided C code on the Pi sends over a plaintext message and a key and the received encrypted message is correct. This works for both of the provided test cases.

Conclusion

I unfortunately spent more than an hour trying to debug my physical hardware on the breadboard and the pin assignments from the Quartus pin planner since I was working on the lab before the problem with the EasyPIO.h file came to light, but fortunately I was eventually able to get everything working. In total I spent about 10 hours on this lab.

Breadboard Schematic



Verilog

aes.sv

```
/////////////////////////////////////////////
// aes.sv
// HMC E155 16 September 2015
// bchasnov@hmc.edu, David_Harris@hmc.edu
// edited 1 November 2015 ascott@hmc.edu
/////////////////////////////////////////////

/////////////////////////////////////////////
// testbench
//   Tests AES with cases from FIPS-197 appendix
/////////////////////////////////////////////

module testbench();
    logic clk, load, done, sck, sdi, sdo;
    logic [127:0] key, plaintext, cyphertext, expected;
     logic [255:0] comb;
    logic [8:0] i;

    // device under test
    aes dut(clk, sck, sdi, sdo, load, done);

    // test case
    initial begin
    // Test case from FIPS-197 Appendix A.1, B
        key       <= 128'h2B7E151628AED2A6ABF7158809CF4F3C;
        plaintext <= 128'h3243F6A8885A308D313198A2E0370734;
        expected  <= 128'h3925841D02DC09FBDC118597196A0B32;

    // Alternate test case from Appendix C.1
    // key       <= 128'h000102030405060708090A0B0C0D0E0F;
    // plaintext <= 128'h00112233445566778899AABBCCDDEEFF;
    // expected  <= 128'h69C4E0D86A7B0430D8CDB78070B4C55A;
    end

    // generate clock and load signals
    initial
        forever begin
            clk = 1'b0; #5;
            clk = 1'b1; #5;
        end

    initial begin
      i = 0;
      load = 1'b1;
    end

    assign comb = {plaintext, key};
    // shift in test vectors, wait until done, and shift out result
    always @(posedge clk) begin
      if (i == 256) load = 1'b0;
      if (i<256) begin
        #1; sdi = comb[255-i];
        #1; sck = 1; #5; sck = 0;
        i = i + 1;
      end else if (done && i < 384) begin
        #1; sck = 1;
        #1; cyphertext[383-i] = sdo;
        #4; sck = 0;
        i = i + 1;
      end else if (i == 384) begin
            if (cyphertext == expected)
                $display("Testbench ran successfully");
            else $display("Error: cyphertext = %h, expected %h",
                cyphertext, expected);
            $stop();

      end
    end

endmodule


/////////////////////////////////////////////
// aes
//   Top level module with SPI interface and SPI core
/////////////////////////////////////////////

module aes(input  logic clk,
           input  logic sck,
           input  logic sdi,
           output logic sdo,
           input  logic load,
           output logic done);

    logic [127:0] key, plaintext, cyphertext;
    aes_spi spi(sck, sdi, sdo, done, key, plaintext, cyphertext);
    aes_core core(clk, load, key, plaintext, done, cyphertext);
endmodule

/////////////////////////////////////////////
// aes_spi
//   SPI interface.  Shifts in key and plaintext
//   Captures ciphertext when done, then shifts it out
//   Tricky cases to properly change sdo on negedge clk
```

```
////////////////////////////////////////////

module aes_spi(input  logic sck,
               input  logic sdi,
               output logic sdo,
               input  logic done,
               output logic [127:0] key, plaintext,
               input  logic [127:0] cyphertext);

    logic         sdodelayed, wasdone;
    logic [127:0] cyphertextcaptured;

    // assert load
    // apply 256 sclks to shift in key and plaintext, starting with plaintext[0]
    // then deassert load, wait until done
    // then apply 128 sclks to shift out cyphertext, starting with cyphertext[0]
    always_ff @(posedge sck)
        if (!wasdone)  {cyphertextcaptured, plaintext, key} = {cyphertext,
            plaintext[126:0], key, sdi};
        else           {cyphertextcaptured, plaintext, key} = {cyphertextcaptured[126:0],
            plaintext, key, sdi};

    // sdo should change on the negative edge of sck
    always_ff @(negedge sck) begin
        wasdone = done;
        sdodelayed = cyphertextcaptured[126];
    end

    // when done is first asserted, shift out msb before clock edge
    assign sdo = (done & !wasdone) ? cyphertext[127] : sdodelayed;
endmodule

////////////////////////////////////////////
// aes_core
//   top level AES encryption module
//   when load is asserted, takes the current key and plaintext
//   generates cyphertext and asserts done when complete 11 cycles later
//
//   See FIPS-197 with Nk = 4, Nb = 4, Nr = 10
//
//   The key and message are 128-bit values packed into an array of 16 bytes as
//   shown below
//        [127:120] [95:88] [63:56] [31:24]     S0,0    S0,1    S0,2    S0,3
//        [119:112] [87:80] [55:48] [23:16]     S1,0    S1,1    S1,2    S1,3
//        [111:104] [79:72] [47:40] [15:8]      S2,0    S2,1    S2,2    S2,3
//        [103:96]  [71:64] [39:32] [7:0]       S3,0    S3,1    S3,2    S3,3
//
//   Equivalently, the values are packed into four words as given
//        [127:96]  [95:64] [63:32] [31:0]      w[0]    w[1]    w[2]    w[3]
////////////////////////////////////////////

module aes_core(input  logic         clk,
                input  logic         load,
                input  logic [127:0] key,
                input  logic [127:0] plaintext,
                output logic         done,
                output logic [127:0] cyphertext);

    logic outKeyCtrl;
    logic [1:0] roundKeyCtrl;
    logic [3:0] round;
    aescontroller controller(clk, load, done, roundKeyCtrl, outKeyCtrl, round);
    aesdatapath datapath(clk, load, done, outKeyCtrl, roundKeyCtrl, round, key,
        plaintext, cyphertext);
endmodule

////////////////////////////////////////////
// aescontroller
//   Controller for the AES encryption, generates the roundKeyCtrl and outKeyCtrl signals
```

```
/////////////////////////////////////////////

module aescontroller(input logic clk, reset,
                     output logic done,
                     output logic [1:0] roundKeyCtrl,
                     output logic outKeyCtrl,
                     output logic [3:0] round);
    typedef enum logic [1:0] {Round1 = 2'b00, Round2_9 = 2'b01, Round10 = 2'b10,
        Done = 2'b11} statetype;
    statetype state, next;

    // state register
    always_ff @(posedge clk, posedge reset)
        if (reset)
            state <= Round1;
        else
            state <= next;

    // next state logic
    always_comb
    case(state)
        Round1: next = Round2_9;
        Round2_9: next = (round == 4'b1010) ? Round10 : Round2_9;
        Round10: next = Done;
        Done: next = Done;
        default: next = Round1;
    endcase

    // output logic
    assign roundKeyCtrl = state;            // state encoding matches round key control signal
    assign outKeyCtrl = state != Round1;
    assign done = state == Done;

    // round count register
   always_ff @(posedge clk, posedge reset)
        if (reset)
            round <= 4'b0001;
        else
            round <= round + 1'b1;
endmodule


/////////////////////////////////////////////
// aesdatapath
//   Datapath for the AES encryption, handles the actual bit modifications
//   of the input
/////////////////////////////////////////////

module aesdatapath(input logic clk, reset, done, outKeyCtrl,
                   input logic [1:0] roundKeyCtrl,
                   input logic [3:0] round,
                   input logic [127:0] key, plaintext,
                   output logic [127:0] cyphertext);
    logic [127:0] addRoundKeyOut, subBytesOut, shiftRowsOut, mixColumnsOut,
        newCypher, addRoundKeyIn, roundKey;

    subbytes subbytes(addRoundKeyOut, subBytesOut);
    shiftrows shiftrows(subBytesOut, shiftRowsOut);
    mixcolumns mixcolumns(shiftRowsOut, mixColumnsOut);
    weflopr #(128) cypherreg(clk, reset, ~done, newCypher, addRoundKeyIn);
    mux3 #(128) newCypherMux(plaintext, mixColumnsOut, shiftRowsOut, roundKeyCtrl, newCypher);
    addroundkey addroundkey(addRoundKeyIn, roundKey, addRoundKeyOut);
    keyexpansion keyexpansion(clk, reset, done, outKeyCtrl, round, key, roundKey);

    assign cyphertext = addRoundKeyOut;
endmodule


/////////////////////////////////////////////
```

```
// subbytes
//   Apply the sbox transformation to each byte in a 128 bit block
//   Section 5.1.1, Figure 6
/////////////////////////////////////////////

module subbytes(input logic [127:0] a,
                output logic [127:0] y);
    subbyte subbyte0(a[127:96], y[127:96]);
    subbyte subbyte1(a[95:64], y[95:64]);
    subbyte subbyte2(a[63:32], y[63:32]);
    subbyte subbyte3(a[31:0], y[31:0]);
endmodule

/////////////////////////////////////////////
// subbyte
//   Apply the sbox transformation to a single byte
//   Section 5.1.1
/////////////////////////////////////////////

module subbyte(input logic [31:0] a,
               output logic [31:0] y);
    sbox sbox0(a[31:24], y[31:24]);
    sbox sbox1(a[23:16], y[23:16]);
    sbox sbox2(a[15:8], y[15:8]);
    sbox sbox3(a[7:0], y[7:0]);
endmodule

/////////////////////////////////////////////
// sbox
//   Infamous AES byte substitutions with magic numbers
//   Section 5.1.1, Figure 7
/////////////////////////////////////////////

module sbox(input  logic [7:0] a,
            output logic [7:0] y);

    // sbox implemented as a ROM
    logic [7:0] sbox[0:255];

    initial $readmemh("sbox.dat", sbox);
    assign y = sbox[a];
endmodule

/////////////////////////////////////////////
// shiftrows
//   Shifting of bytes in the last 3 rows, shift each row by the row number
//   Section 5.1.2, Figure 8
/////////////////////////////////////////////

module shiftrows(input logic [127:0] a,
                output logic [127:0] y);
    // assign the output rows to the proper parts of the input rows
    assign y[127:96] = {a[127:120], a[87:80], a[47:40], a[7:0]};
    assign y[95:64] = {a[95:88], a[55:48], a[15:8], a[103:96]};
    assign y[63:32] = {a[63:56], a[23:16], a[111:104], a[71:64]};
    assign y[31:0] = {a[31:24], a[119:112], a[79:72], a[39:32]};
endmodule

/////////////////////////////////////////////
// mixcolumns
//   Even funkier action on columns
//   Section 5.1.3, Figure 9
//   Same operation performed on each of four columns
/////////////////////////////////////////////

module mixcolumns(input  logic [127:0] a,
                  output logic [127:0] y);
    mixcolumn mc0(a[127:96], y[127:96]);
    mixcolumn mc1(a[95:64],  y[95:64]);
    mixcolumn mc2(a[63:32],  y[63:32]);
    mixcolumn mc3(a[31:0],   y[31:0]);
endmodule

/////////////////////////////////////////////
// mixcolumn
//   Perform Galois field operations on bytes in a column
//   See EQ(4) from E. Ahmed et al, Lightweight Mix Columns Implementation for AES, AIC09
//   for this hardware implementation
/////////////////////////////////////////////

module mixcolumn(input  logic [31:0] a,
                 output logic [31:0] y);
    logic [7:0] a0, a1, a2, a3, y0, y1, y2, y3, t0, t1, t2, t3, tmp;

    assign {a0, a1, a2, a3} = a;
    assign tmp = a0 ^ a1 ^ a2 ^ a3;

    galoismult gm0(a0^a1, t0);
    galoismult gm1(a1^a2, t1);
    galoismult gm2(a2^a3, t2);
    galoismult gm3(a3^a0, t3);

    assign y0 = a0 ^ tmp ^ t0;
    assign y1 = a1 ^ tmp ^ t1;
    assign y2 = a2 ^ tmp ^ t2;
    assign y3 = a3 ^ tmp ^ t3;
    assign y = {y0, y1, y2, y3};
endmodule

/////////////////////////////////////////////
// galoismult
//   Multiply by x in GF(2^8) is a left shift
//   followed by an XOR if the result overflows
//   Uses irreducible polynomial x^8+x^4+x^3+x+1 = 00011011
/////////////////////////////////////////////

module galoismult(input  logic [7:0] a,
                  output logic [7:0] y);
    logic [7:0] ashift;

    assign ashift = {a[6:0], 1'b0};
    assign y = a[7] ? (ashift ^ 8'b00011011) : ashift;
endmodule

/////////////////////////////////////////////
// addroundkey
//   Add the round key to each column
//   Section 5.1.4, Figure 10
/////////////////////////////////////////////

module addroundkey(input logic [127:0] a, roundkey,
                   output logic [127:0] y);
    assign y = a ^ roundkey;
endmodule

/////////////////////////////////////////////
// keyexpansion
//   Expand the cipher key to generate a key schedule
//   Section 5.2, Figure 11
/////////////////////////////////////////////

module keyexpansion(input logic clk, reset, done, outKeyCtrl,
                    input logic [3:0] round,
                    input logic [127:0] inKey,
                    output logic [127:0] outKey);
    logic [127:0] nextKey, possibleNewKey, newKey;
    logic [31:0] modNkTemp;
    // hold the new key generated after each round
    weflopr #(128) keyreg(clk, reset, ~done, newKey, nextKey);
    // perform a special operation for the lower 32 bits of the new key
    modnk modnk(nextKey[31:0], round, modNkTemp);
    // generate the 4 bytes of the new key
    assign possibleNewKey[127:96] = modNkTemp ^ nextKey[127:96];
    assign possibleNewKey[95:64] = possibleNewKey[127:96] ^ nextKey[95:64];
    assign possibleNewKey[63:32] = possibleNewKey[95:64] ^ nextKey[63:32];
    assign possibleNewKey[31:0] = possibleNewKey[63:32] ^ nextKey[31:0];
    // choose if we pick the input key (1st round) or newly generated key (all other rounds)
    assign newKey = outKeyCtrl ? possibleNewKey : inKey;
    assign outKey = nextKey;
endmodule

/////////////////////////////////////////////
// modnk
//   Generate the first 32 bits of the new key during key expansion
//   Section 5.2, Figure 11, line under 'if (i mod Nk = 0)'
/////////////////////////////////////////////

module modnk(input logic [31:0] inKey,
                 input logic [3:0] round,
                 output logic [31:0] outKey);
    logic [31:0] rotatedKey, subbedKey, rConi;
    assign rotatedKey = {inKey[23:0], inKey[31:24]};    // rotate the input properly
    subbyte subbyte(rotatedKey, subbedKey);             // perform the byte substitutions
    rconi rconi(round, rConi);                          // get the Rcon[i] value for this round
    assign outKey = subbedKey ^ rConi;
endmodule

/////////////////////////////////////////////
// rconi
//   Generate the value of the round constant array at i
//   Section 5.2, Figure 11, Rcon[i/Nk]
/////////////////////////////////////////////

module rconi(input logic [3:0] round,
             output logic [31:0] rConi);
    assign rConi[23:0] = 24'b0;
    always_comb
    case(round)
        4'b0010: rConi[31:24] = 8'b0000_0001;
        4'b0011: rConi[31:24] = 8'b0000_0010;
        4'b0100: rConi[31:24] = 8'b0000_0100;
        4'b0101: rConi[31:24] = 8'b0000_1000;
        4'b0110: rConi[31:24] = 8'b0001_0000;
        4'b0111: rConi[31:24] = 8'b0010_0000;
        4'b1000: rConi[31:24] = 8'b0100_0000;
        4'b1001: rConi[31:24] = 8'b1000_0000;
        4'b1010: rConi[31:24] = 8'b0001_1011;
        4'b1011: rConi[31:24] = 8'b0011_0110;
        default: rConi[31:24] = 8'b0000_0000;
    endcase
endmodule

/////////////////////////////////////////////
// weflopr
//   Parameterized width write enable flop that writes every clock cycle when
//   enable is high
/////////////////////////////////////////////

module weflopr #(parameter WIDTH = 8)
              (input  logic             clk, reset, enable,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if (reset)
            q <= 0;
        else if (enable)
            q <= d;
endmodule

/////////////////////////////////////////////
// mux3
//   Parameterized width 3 input mux
/////////////////////////////////////////////

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

    assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

C Code

lab7.c

```
// lab7.c
// bchasnov@hmc.edu, david_harris@hmc.edu 15 October 2015
//
// Send data to encryption accelerator over SPI


////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include "easypio.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////

#define LOAD_PIN 23
#define DONE_PIN 24

// Test Case from Appendix A.1, B
char key[16] = {0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6,
                0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C};

char plaintext[16] = {0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30, 0x8D,
                      0x31, 0x31, 0x98, 0xA2, 0xE0, 0x37, 0x07, 0x34};

char ct[16] = {0x39, 0x25, 0x84, 0x1D, 0x02, 0xDC, 0x09, 0xFB,
               0xDC, 0x11, 0x85, 0x97, 0x19, 0x6A, 0x0B, 0x32};

/*
// Another test case from Appendix C.1

char key[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};

char plaintext[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
                      0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};

char ct[16] = {0x69, 0xC4, 0xE0, 0xD8, 0x6A, 0x7B, 0x04, 0x30,
               0xD8, 0xCD, 0xB7, 0x80, 0x70, 0xB4, 0xC5, 0x5A};
*/

////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////

void encrypt(char*, char*, char*);
void print16(char*);
void printall(char*, char*, char*);
```

```
////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

void main(void) {
  char cyphertext[16];

  pioInit();
  spiInit(244000, 0);

  // Load and done pins
  pinMode(LOAD_PIN, OUTPUT);
  pinMode(DONE_PIN, INPUT);

  // hardware accelerated encryption
  encrypt(key, plaintext, cyphertext);
  printall(key, plaintext, cyphertext);
}

////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

void printall(char *key, char *plaintext, char *cyphertext) {
  printf("Key:        ");  print16(key);
  printf("Plaintext:  ");  print16(plaintext);  printf("\n");
  printf("Ciphertext: ");  print16(cyphertext);
  printf("Expected:   ");  print16(ct);

  if(strcmp(cyphertext, ct) == 0) {
    printf("\nSuccess!\n");
  } else {
    printf("\nBummer.  Test failed\n");
  }
}

void encrypt(char *key, char *plaintext, char *cyphertext) {
  int i;
  int ready;

  digitalWrite(LOAD_PIN, 1);

  for(i = 0; i < 16; i++) {
    spiSendReceive(plaintext[i]);
  }

  for(i = 0; i < 16; i++) {
    spiSendReceive(key[i]);
  }

  digitalWrite(LOAD_PIN, 0);

  while (!digitalRead(DONE_PIN));

  for(i = 0; i < 16; i++) {
    cyphertext[i] = spiSendReceive(0);
  }
}

void print16(char *text) {
  int i;

  for(i = 0; i < 16; i++) {
    printf("%02x ",text[i]);
  }
  printf("\n");
}
```