Jeremy Barthélemy
ECE 746
Hw. 3

1.

a.

We take $A(z)*B(z) = (z + [x])([x+1]z) = [x+1]z^2 + [x^2 + x]z$, which we reduce to $[x+1]z^2 + z$. We then reduce by R(z) to produce:

$$\begin{array}{l} \quad\quad\underline{(x+1)} \\ x^2 + x + 1 \,|\, [x+1]z^2 + z \\ \quad\quad\underline{[x+1]z^2 + [x+1]z + [x+1]^2} \\ \quad\quad 0 + z + [x+1]z + [x+1]^2 = z + [x^2 + 2x + 1] = z + [x^2 + 1] \end{array}$$

We further reduce this to $z + [x]z$ again using R(z).

b. $A(z) = z + [x]$

Using Fermat's Little Theorem, we determine the inverse of A(z) by doing the following computation:

$A^{-1}(z) = A^{(2\text{^}m - 1)} \bmod R(z)$. Here we have m = 4, thus $A^{-1}(z)$ will be equal to $A^{14}(z)$.

$A^2(z) = (z + [x])*(z + [x]) = z^2 + [x^2] \bmod z^2 + z + [x+1] = z^2 + [x+1] \bmod z^2 + z + [x+1] = z$

$A^4(z) = z^2 \bmod z^2 + z + [x + 1] = z + [x + 1]$

$A^8(z) = (z + [x+1])*(z + [x+1]) = z^2 + [x+1]^2 \bmod z^2 + z + [x + 1] =$

$z^2 + [x^2 + 1] \bmod z^2 + z + [x+1] = z^2 + x = z + 1$

$A^{10}(z) = (z + 1)*z = z^2 + z \bmod z^2 + z + [x + 1] = [x + 1]$

$A^{14}(z) = [x + 1] * (z + [x + 1]) = [x+1]*z + [x+1]^2 = x^2 + 1 + [x+1]*z = [x+1]z + x$

Thus, $A^{-1}(z) = A^{14}(z) = [x+1]z + x$

2.

We notice that no matter what value m, for a given A(x), we find that $A^2(x)$ will have the following values with regard to A(x):

$(am\text{-}1x^{m-1} + \ldots + a1x + a0)2$

$= am\text{-}1x^{2(m-1)} + \ldots + a1x^2 + a0$

In other words, for a given input A(x), we are simply squaring each of the x values to get $A^2(x)$. For example, in $GF(2^2)$, if we set $A(x) = (x^2 + x + 1)$, then the result of A(x)*A(x) will be $(x^4 + x^2 + 1)$, which holds true to this statement.

Another example is if we set $A(x) = x^4 + x^2 + 1$. $A^2(x)$ is: $x^8 + x^4 + 1$.

As a proof, if we take A(x) * A(x) =

$(am\text{-}1x^{m-1} + \ldots + a1x + a0) * (am\text{-}1x^{m-1} + \ldots + a1x + a0) =$

$a_{m-1}{}^2 x^{2(m-1)} + \ldots + a_{m-1}a_1x^m + a_{m-1}a_0x^{m-1} + \ldots + a_{m-1}a_1x^m + \ldots + a_1{}^2x^2 + a_0a_1x + a_{m-1}a_0x^{m-1} + \ldots + a_0a_1x + a_0{}^2$

$= a_{m-1}{}^2x^{2(m-1)} + \ldots + a_1{}^2x^2 + a_0{}^2$

3.
a.
Given an input “C0 FF EE 01,” we perform the MixColumns transformation:
We multiply the column against the stored MinColumns matrix to determine the output from this transformation.

C0		2	3	1	1
FF	*	1	2	3	1
EE		1	1	2	3
1		3	1	1	2

Here is what the calculations look like to determine the output column:
Note that each “+” is actually an XOR operator.
2*C0 + 3*FF + EE + 1 →

2*C0 = 10000000 + 00011011 = 10011011
3*FF = FF + 2*FF = 11111111 + 11111110 + 00011011 = 00011010

Now that we have all the values together, we can XOR them all to get the result:

10011011
00011010
00000001
11101110
01101110 = 6E

C0 + 2*FF + 3*EE + 1 →
To find FF,
11111110 + 00011011 = 11100101

To find 3*EE,
EE + 2*EE =
11101110 + 11011100 + 00011011 = 00101100

11101110 + 11011100 + 00011011 = 00101001

Now, we XOR it all together:
11000000
11100101
00101001
00000001
00001101 = 0D

C0 + FF + 2*EE + 3 →
2*EE = 11000111

XOR it all together:
11000000
11111111
11000111
00000011
11111011 = FB

3*C0 + FF + EE + 2 →
3*C0 = 2*C0 + C0 = 10011011 + 11000000 = 01011011
XOR it all together:
01011011
11111111
11101110
00000010
01001000 = 48

And so our final output is:

6E
0D
FB
48

b.

We need to compute the following in order to verify our previous solution:

6E		0E	0B	0D	9
0D	*	9	0E	0B	0D
FB		0D	9	0E	0B
48		0B	0D	9	0E

In determining these values, I used multiplication tables in GF(2^8), which I found here:
http://en.wikipedia.org/wiki/Rijndael_mix_columns
We must perform multiplications by 9, 11, 13, and 14.

0E*6E + 0B*0D + 0D*FB + 09*48 →
0E*6E = 14 * 110 = 0x22 = 00100010
0B*0D =11 * 13 = 0x7F = 01111111
0D*FB = 13 * 251 = 0xA3 = 10100011
09*48 = 9 * 72 = 0x3E = 00111110

XOR it all together:
00100010
01111111
10100011
<u>00111110</u>
11000000 = C0

9*6E + 0E*0D + 0B*FB + 0D*48 →
9*6E = 9 * 110 = 0x33
0E*0D = 14 * 13 = 0x46
0B*FB = 11 * 251 = 0x8F
0D*48 = 13 * 72 = 0x05

XOR it all together:
00110011
01000110
10001111
<u>00000101</u>
11111111 = FF

0D*6E + 09*0D + 0E*FB + 0B*48 →
0D*6E = 13 * 110 = 0x90
09*0D = 9 * 13 = 0x65
0E*FB = 14 * 251 = 0xB5
0B*48 = 11 * 72 = 0xAE
XOR it all together:
10010000
01100101
10110101
<u>10101110</u>
11101110 = EE

0B*6E + 0D*0D + 09*FB + 0E*48 →
0B*6E = 11 * 110 = 0xEF
0D*0D = 13 * 13 = 0x51
09*FB = 9 * 251 = 0x62
0E*48 = 14 * 72 = 0xDD
XOR it all together:
11101111
01010001
01100010
<u>11011101</u>
00000001 = 01

And so our final output is:

C0
FF
EE
1

We have verified part A!

c.
Now, we calculate the same thing, but we change C0 to C1 as our input.

C1		2	3	1	1
FF	*	1	2	3	1
EE		1	1	2	3
1		3	1	1	2

We must compute the following:
2*C1 + 3*FF + EE + 1 →
2*C1 = 10000010 + 00011011 = 10011001
3*FF = FF + 2*FF = 11111111 + 11111110 + 00011011 = 00011010
XOR it all together:
10011001
00011010
11101110
<u>00000001</u>
01101100 = 6C

C1 + 2*FF + 3*EE + 1 →
2*FF = 11111110 + 00011011 = 11100101
3*EE = EE + 2*EE = 11101110 + 11011100 + 00011011 = 00101001
XOR it all together:
11000001
11100101
00101001
<u>00000001</u>
00001100 = 0C

C1 + FF + 2*EE + 3 →
2*EE = 11011100 + 00011011 = 11000111
XOR it all together:
11000001
11111111
11000111
<u>00000011</u>
11111010 = FA

3*C1 + FF + EE + 2 →
3*C1 = C1 + 2*C1 = 11000001 + 10011001 = 01011000
XOR it all together:
01011000
11111111
11101110
<u>00000010</u>
01001011 = 4B

So the output for this input is:

6C
0C
FA
4B

If we compare the output for (a) and the output for (b),
01101110000011011111101101001000
01101100000011001111101001001011,
we find that 5 bits have changed of the output of 32 bits. So in byte 1, there was 1 change, in byte 2, there was 1 change, in byte 3, there was1 change, and in byte 4, there were 2 changes.

4.

(a) First we apply the multiplicative inverse of 02 under GF(2^8) (taken from a table) which is 8D.

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**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

Given the value 8D, we then perform the affine transformation as follows to determine the output:
We must convert 8D to binary and use it as our input.
8D = $(10001101)_2$

$$\begin{vmatrix} b7 \\ b6 \\ b5 \\ b4 \\ b3 \\ b2 \\ b1 \\ b0 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{vmatrix} + \begin{vmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{vmatrix}$$

Note that "+" is being used as XOR.
b7 = 1*1 + 1*0+1*0+1*0+1*1+0*1+0*0+0*1 + 0 = 1+ 0+0+0+1+0+0+0 +0 = 0
b6 = 0*1 + 1*0+1*0+1*0+1*1+1*1+0*0+0*1 + 1 =0 + 0+0+0+1+1+0+0 + 1 = 1
b5 = 0*1 +0*0+1*0+1*0+1*1+1*1+1*0+0*1 + 1 =0 +0+0+0+1+1+0+0 + 1 =1
b4 = 0*1 + 0*0+0*0+1*0+1*1+1*1+1*0+1*1 + 0 =0 + 0+0+0+1+1+0+1 + 0 = 1
b3 = 1*1 + 0*0+0*0+0*0+1*1+1*1+1*0+1*1 + 0 =1 + 0+0+0+1+1+0+1 + 0 = 0
b2 = 1*1 + 1*0+0*0+0*0+0*1+1*1+1*0+1*1 + 0 =1 + 0+0+0+0+1+0+1 + 0 =1
b1 = 1*1 + 1*0+1*0+0*0+0*1+0*1+1*0+1*1 + 1 =1 + 0+0+0+0+0+0+1 + 1 = 1
b0 = 1*1 + 1*0+1*0+1*0+0*1+0*1+0*0+1*1 + 1 =1 + 0+0+0+0+0+0+1 + 1 = 1

$(01110111)_2$ = 0x77, which verifies that the output is 77 when our input to an Sbox is 02!

(b) We perform the inverse affine transform first:

$$\begin{vmatrix} a7 \\ a6 \\ a5 \\ a4 \\ a3 \\ a2 \\ a1 \\ a0 \end{vmatrix} = \begin{vmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{vmatrix} * \begin{vmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{vmatrix} + \begin{vmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{vmatrix}$$

a7 = 0*0 + 1*0 + 0*0 + 1*0 + 0*0 + 0*0 + 1*1 + 0*0 + 0 = 1 + 0 = 1
a6 = 0*0 + 0*0 + 1*0 + 0*0 + 1*0 + 0*0 + 0*1 + 1*0 + 0 =0 + 0 = 0
a5 = 1*0 + 0*0 + 0*0 + 1*0 + 0*0 + 1*0 + 0*1 + 0*0 + 0 = 0 + 0 = 0
a4 = 0*0 + 1*0 + 0*0 + 0*0 + 1*0 + 0*0 + 1*1 + 0*0 + 0 = 1 + 0 = 1
a3 = 0*0 + 0*0 + 1*0 + 0*0 + 0*0 + 1*0 + 0*1 + 1*0 + 0 = 0 + 0 = 0
a2 = 1*0 + 0*0 + 0*0 + 1*0 + 0*0 + 0*0 + 1*1 + 0*0 + 1 = 1 + 1 = 0

a1 = 0*0 + 1*0 + 0*0 + 0*0 + 1*0 + 0*0 + 0*1 + 1*0 + 0 = 0 + 0 = 0
a0 = 1*0 + 0*0 + 1*0 + 0*0 + 0*0 + 1*0 + 0*1 + 0*0 + 1 = 0 + 1 = 1

Thus, the output is 10010001 or 0x91 in hex. With this value, we look up in the multiplicative inverse table in GF(2^8) again but with 0x8D this time, and we find that output value is 6A, which verifies part (b) as well!

5.
With an input key of zeroes for all 128 bits, we can split this into 32 bit words to have:
w0 = 0x00000000
w1 =0x00000000
w2 =0x00000000
w3 =0x00000000
w4 = w0 + g(w3) = 0x00000000 + g(0x00000000)
To calculate g(0x00000000), we first shift bytes, but this will have no effect as all the bytes are 0. The next step is to feed in a byte into four s-boxes, and then return the output and XOR this with 1 byte from a round constant.
Looking up in each S-box, we find each byte is 63, so our current value is:
0x63636363. We then take the first byte and XOR it with our current RC, which will be 01 for the first round. Thus,
w4 = 0x62636363
w5 = 0x62636363 + 0x00000000
= 0x62636363
w6 = 0x62636363 + 0x00000000
= 0x62636363
w7 = w6 + w3 = 0x62636363 + 0x00000000
= 0x62636363

RK1 = w4 || w5 || w6 || w7 = 0x62636363626363636263636362636363

Now, to find the second round key,
w8 = w4 + g(w7) = 0x62636363 + g(62636363)
Finding g(62636363),
After we perform the permutations, we have 63636362. From after the sboxes, we have:
FBFBFBAA, which we then XOR against 02000000 such that g(62636363) =
F9FBFBAA, thus,
w8 = 0x62636363 + 0xF9FBFBAA = 0x9B9898C9
w9 = w8 + w5 = 0x9B9898C9 + 0x62636363 = 0xF9FBFBAA
w10 = w9 + w6 = 0xF9FBFBAA + 0x62636363 = 0x9B9898C9
w11 = w10 + w7 = 0x9B9898C9 + 0x62636363 = 0xF9FBFBAA
RK2 = w8 || w9 || w10 || w11 = 0x9B9898C9F9FBFBAA9B9898C9F9FBFBAA

Thus, our two round keys are:
0x62636363626363636263636362636363 and
0x9B9898C9F9FBFBAA9B9898C9F9FBFBAA

6.
Round Input:

01	01	02	01
02	01	01	01
02	01	01	01
01	01	02	01

Round Key:

00	01	FF	08
FF	02	00	04
FF	04	00	02
00	08	FF	01

We can determine the output for the round:

e0,0	e0,1	e0,2	e0,3
e1,0	e1,1	e1,2	e1,3
e2,0	e2,1	e2,2	e2,3
e3,0	e3,1	e3,2	e3,3

We determine the output in columns below.
We use the following equation to determine the outputs for each column:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j+C1}] \oplus T_2[a_{2,j+C2}] \oplus T_3[a_{3,j+C3}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Where j is used to represent a column, and T is used to refer to the respective lookup table.
First, we determine our the Sbox values for our inputs:

7C	7C	77	7C
77	7C	7C	7C
77	7C	7C	7C
7C	7C	77	7C

Next, we need to multiply these values to by the appropriate matrices in order to determine the values in the lookup tables.

We multiply our input columns by each column of these below, respectively, to determine what to XOR our key column by.

2		3		1		1
1		2		3		1
1		1		2		3
3		1		1		2

For the first column, we have the following matrices to XOR (all of which we have received from multiplying by 7C):

F8		84		7C		7C		0
7C	+	F8	+	84	+	7C	+	FF
7C		7C		F8		84		FF
84		7C		7C		F8		0

The result for the first column is:

7C
83
83
7C

We continue with calculating the second output column, where, again, we simply multiply our matrices by 7C and then XOR them with the key column matrix. The only difference now because of this is our key column matrix, which has values 1, 2, 4, and 8 from top to bottom, respectively.
For this, it's simply:

7C		1		7D
83 + FF	+	2	=	7E
83 + FF		4		78
7C		8		74

Next up, we calculate column three:
This time we are multiplying each matrix in this order: 77, 7C, 77, 7C.

EE		84		7C		77		FF
77	+	F8	+	84	+	77	+	0
77		7C		F8		99		0
99		7C		7C		EE		FF

The result is:

9E
6A
61
95

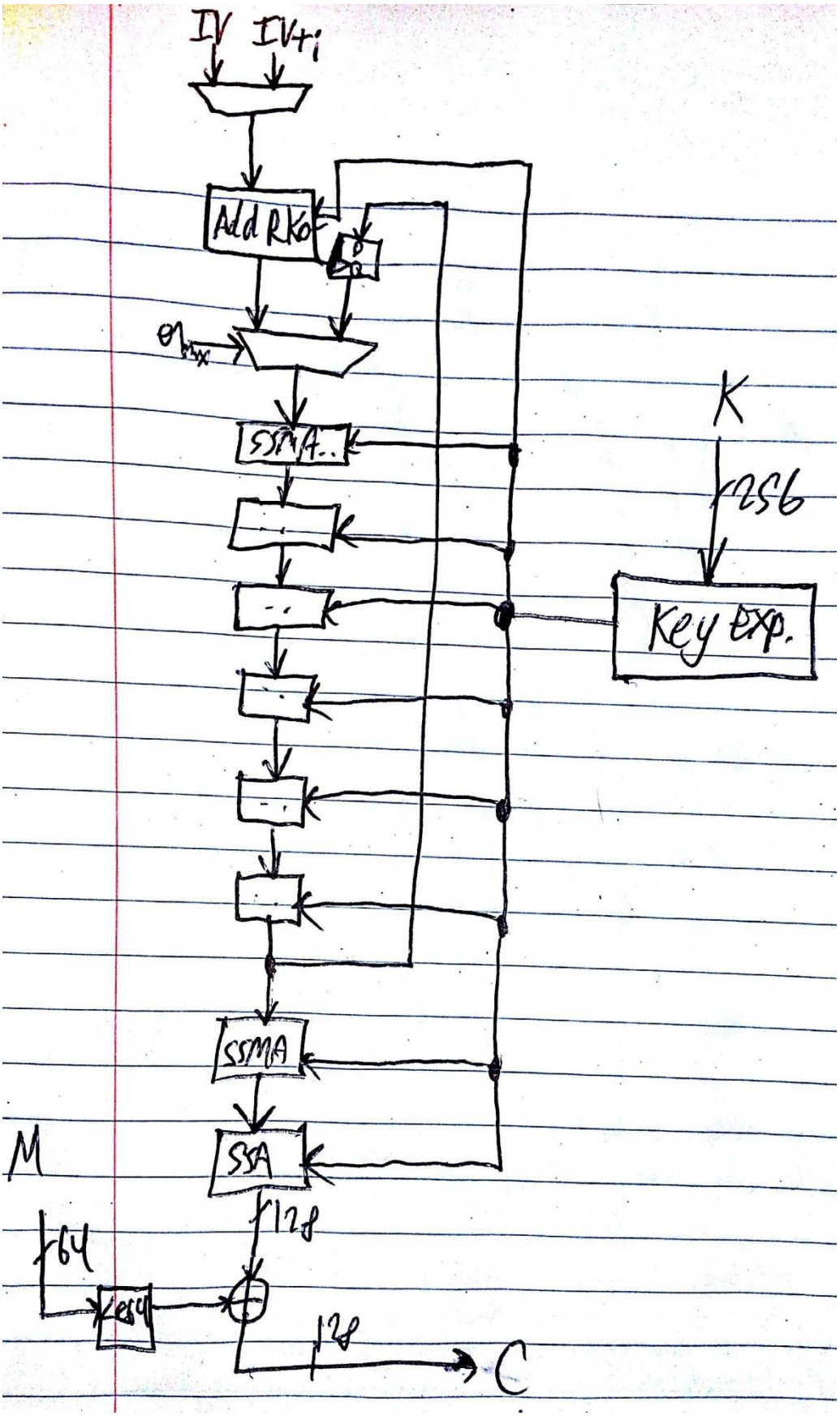
Finally, the fourth column which we obtain by multiplying by 7C, 77, 7C, 77:
The result is:

62
65
68
60

Thus, the entire result is:

7C	7D	9E	62
83	7E	6A	65
83	78	61	68
7C	74	95	60

7. Since we have a 256-bit key, we will have 14 rounds in AES. In order to reduce the number of clock cycles needed to process each block of plaintext, we will need to unroll the rounds, such that we may reduce the number of registers required to process the data. This will consume a much larger amount of area, but will also increase the throughput of our AES-CTR circuit. First, we have the data pass through the Add Round Key stage (Round 0), then we have it pass through a multiplexer to cover 6 rounds. We then have it repeat this process, go through another round, and then finally go through the final round that does not include the MixColumns operation. After this, the 128 bit output is XORed with our first two blocks of message plaintext, which has been shifted by 64 bits over the same two clock cycles. We increment from our initialization vector counter every 2 cycles. Below is the schematic of my circuit:


IV
IV+i
AddRK0
mux
SSMA..
SSMA
SSA
M
64
128
128
C
K
256
Key exp.

It will take two cycles to process 128 bits, and so, given 16MB of plaintext to encrypt, we have 16777216 bytes = 134217728 bits, which is 1048576 blocks. In other words, it will take us 1048576*2 = 2097152 cycles to perform the encryption. Thus, it will take (2*10^-8)*2097152 = 41.9ms to compute the final output ciphertext, provided we assume there is no delay in incrementing the counter from the initialization vector.

8.

a.

For msb = 0, we only need to have A + A*2 to calculate 03*A. But to do the same arithmetic in the Galois field, we need to also consider the case of the most significant bit being 1. In this case, we must also XOR the following: 00011011. Since this only occurs when $a_7 = 1$, then we can XOR certain bits of b_i with a_7 such that the necessary arithmetic operations hold true.

A*03 = B

$a_7a_6a_5a_4a_3a_2a_1a_0$ + 00011011 + $a_7a_6a_5a_4a_3a_2a_10$ for msb = 1.

Thus, for bits b_4, b_3, b_1, and b_0, we also need to XOR by a_7. This produces the following:

$b7 = a_7 + a_6$

$b6 = a_6 + a_5$

$b5 = a_5 + a_4$

$b4 = a_4 + a_3 + a_7$

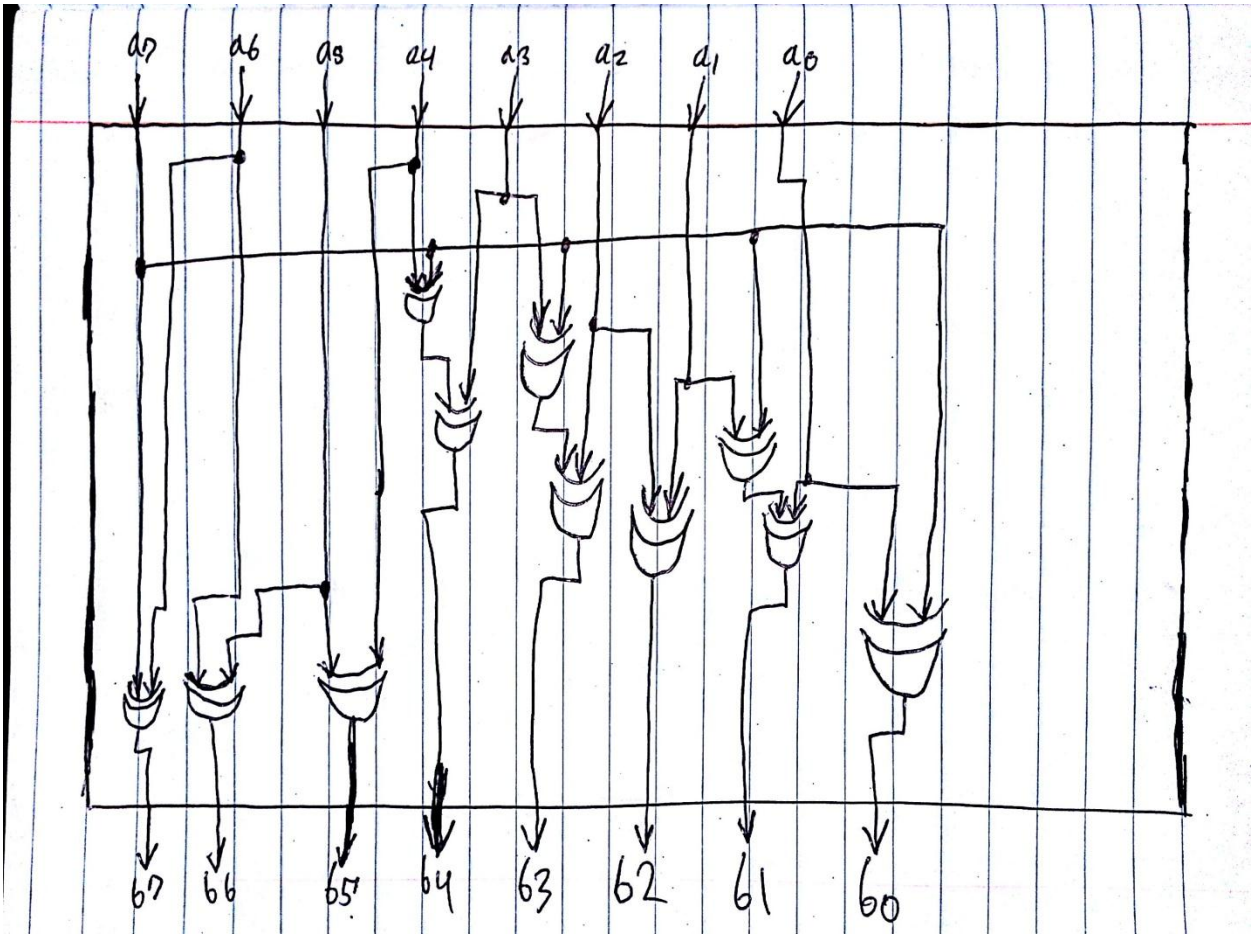
$b3 = a_3 + a_2 + a_7$

$b2 = a_2 + a_1$

$b1 = a_1 + a_0 + a_7$

$b0 = a_0 + a_7$

Below is a hardware schematic of the circuit described above:

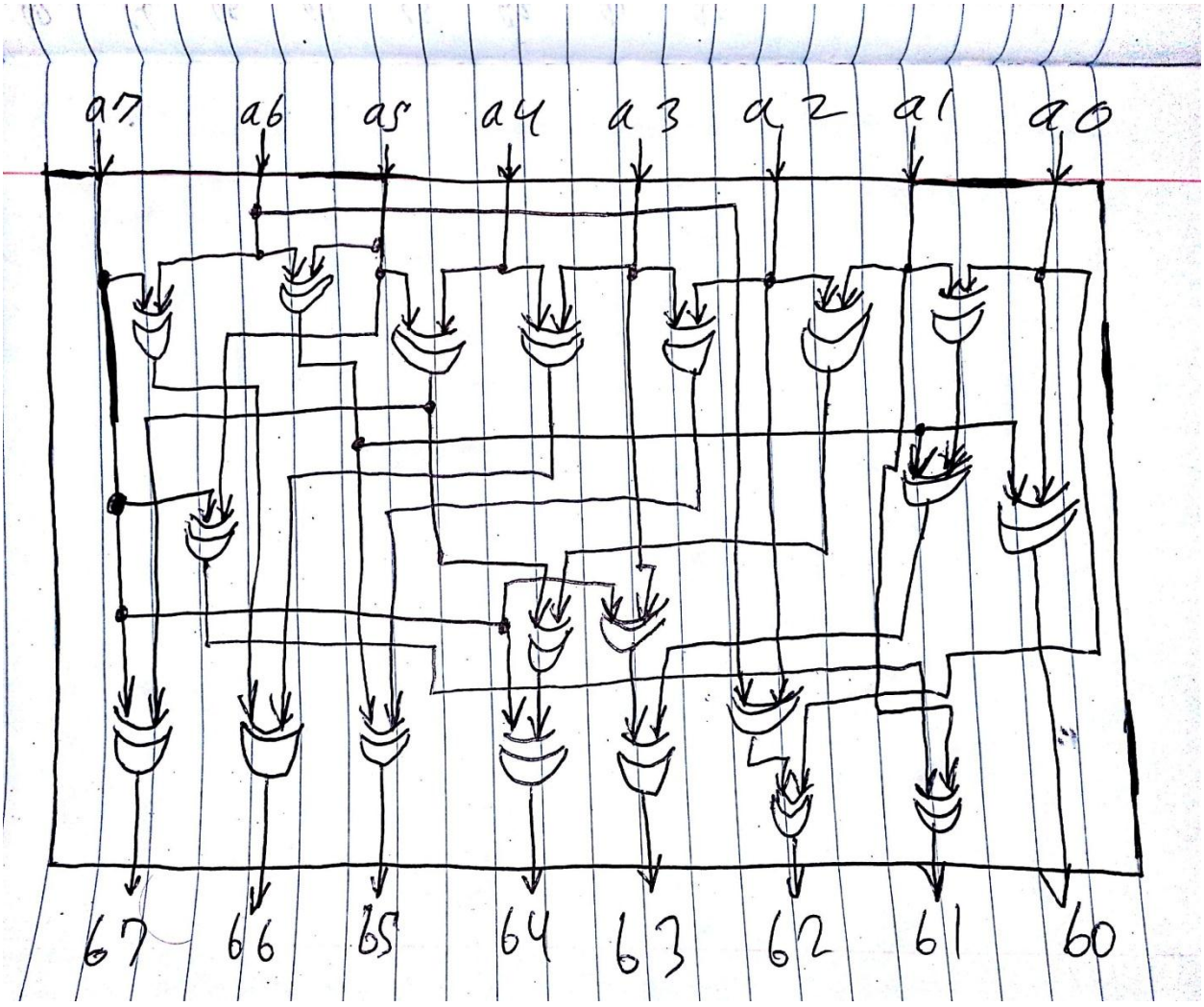


Area = 11 XOR Gates
Latency = delay of 2 XOR gates (as we cascade on four occasions, but in parallel)

b.
This time we are multiplying 0D by our input A to produce B.
To calculate 0D * A, we must perform the following:
((((A*2) + A)*2)*2) + A
Which is also:
$((((a_7a_6a_5a_4a_3a_2a_1a_0*2) + a_7a_6a_5a_4a_3a_2a_1a_0)*2)*2) + a_7a_6a_5a_4a_3a_2a_1a_0 =$
$((((a_6a_5a_4a_3a_2a_1a_00 + 000a7a70a7a7) + a_7a_6a_5a_4a_3a_2a_1a_0)*2)*2) + a_7a_6a_5a_4a_3a_2a_1a_0 =$
$((((a_6a_5a_4 (a_3 + a_7)(a_7 + a_2)(a_1)(a_0 + a_7)a_7) + a_7a_6a_5a_4a_3a_2a_1a_0)*2)*2) + a_7a_6a_5a_4a_3a_2a_1a_0 =$
$(((((a_7 + a_6)(a_6 + a_5)(a_5 + a_4)(a_4 + a_3 + a_7)(a_3 + a_7 + a_2)(a_2 + a_1)(a_1 + a_0 + a_7)a_0 + a_7))*2)*2) +$
$a_7a_6a_5a_4a_3a_2a_1a_0 =$
$(((((a_6 + a_5)(a_5 + a_4)(a_4 + a_3 + a_7)(a_3 + a_7 + a_2)(a_2 + a_1)(a_1 + a_0 + a_7)(a_0 + a_7)(0) + 000(a_7 + a_6) (a_7$
$+ a_6)0(a_7 + a_6) (a_7 + a_6))*2) + a_7a_6a_5a_4a_3a_2a_1a_0 =$
$(((((a_6 + a_5)(a_5 + a_4)(a_4 + a_3 + a_7)(a_3 + a_6 + a_2)(a_2 + a_7 + a_6 + a_1)(a_1 + a_0 + a_7)(a_0 + a_6)(a_7 + a_6))*2)$
$+ a_7a_6a_5a_4a_3a_2a_1a_0 =$

$((a_5 + a_4)(a_4 + a_3 + a_7)(a_3 + a_6 + a_2)(a_2 + a_7 + a_6 + a_1)(a_1 + a_0 + a_7)(a_0 + a_6)(a_7 + a_6)(0) + 000(a_6 + a_5)(a_6 + a_5)0(a_6 + a_5)(a_6 + a_5)) + a_7a_6a_5a_4a_3a_2a_1a_0 =$
$(a_5 + a_4)(a_4 + a_3 + a_7)(a_3 + a_6 + a_2)(a_2 + a_7 + a_5 + a_1)(a_1 + a_0 + a_6 + a_5 + a_7)(a0 + a_6)(a_7 + a_5)(a_6 + a_5) + a_7a_6a_5a_4a_3a_2a_1a_0 =$

$(a_4 + a_5 + a_7)(a_3 + a_4 + a_6 + a_7)(a_2 + a_3 + a_5 + a_6)(a_1 + a_2 + a_4 + a_5 + a_7)(a_0 + a_1 + a_3 + a_5 + a_6 + a_7)(a_0 + a_2 + a_6)(a_1 + a_5 + a_7)(a_0 + a_5 + a_6) = b_7b_6b_5b_4b_3b_2b_1b_0$



Area: 20 Gates
Latency: 3 XOR gate delays