CS433 Modern Architectures

Video 8

AESNI Encryption

Towards AESNI encryption - Cipher Block Chain - CBC

Encrypt

Message: $M = 0110_{(1-4)}, 1111_{(5-8)}, 0000_{(9-12)}\ldots$

$IV = 1010$

$C_{1-4} = IV \otimes M_{1-4}B = 1100$

$C_{5-8} = C_{1-4} \otimes M_{5-8}B = 0011$

$C_{9-12} = C_{5-8} \otimes M_{9-12}B = 0011$

Cipher: $C = 0001_{(1-4)}, 0111_{(5-8)}, 1000_{(9-12)}$

Decrypt

Cipher: $C = 0001_{(1-4)}, 0111_{(5-8)}, 1000_{(9-12)}$

$IV = 1010$

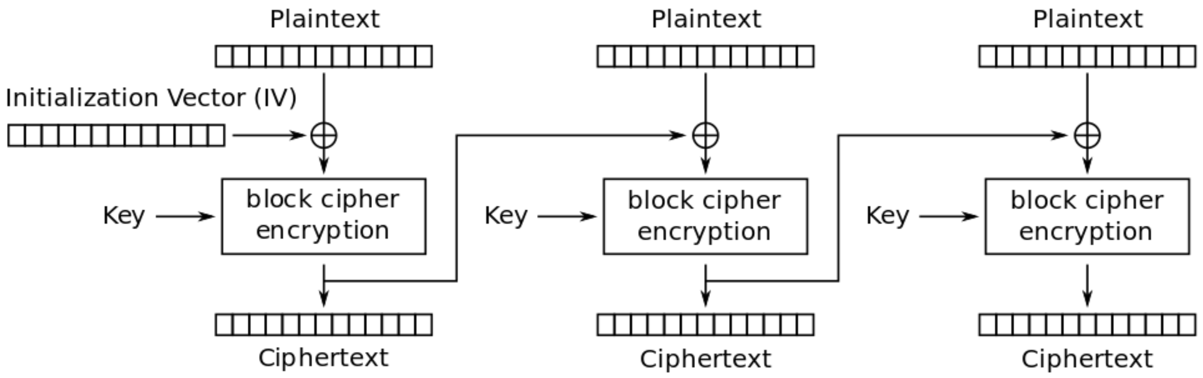
$M_{1-4} = IV \otimes C_{1-4}B^{-1} = 1011$

$M_{5-8} = C_{1-4} \otimes C_{5-8}B^{-1} = 0110$

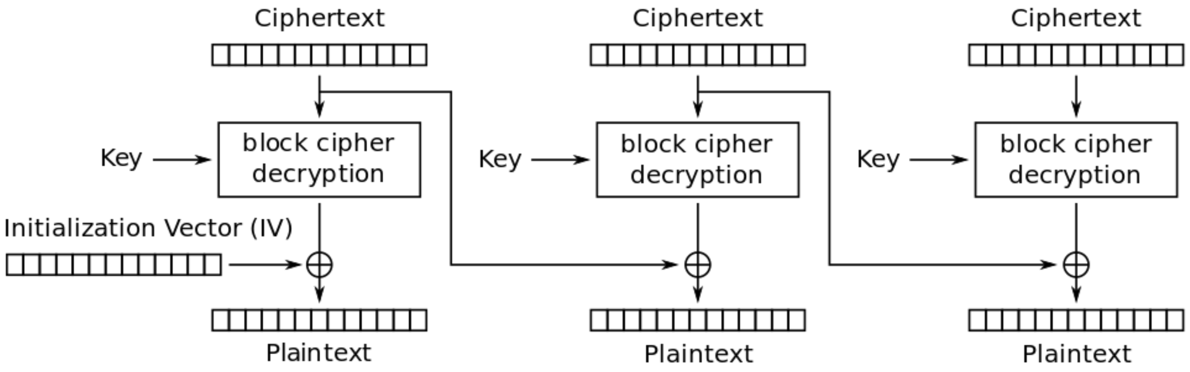
$M_{9-12} = C_{5-8} \otimes C_{9-12}B^{-1} = 1111$

Message: $M = 1011_{(1-4)}, 0110_{(5-8)}, 11\ 11_{(9-12)}$

Q1	Q2	XOR⊗
1	1	0
1	0	1
0	1	1
0	0	0



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

IV – Initialisation vector (number used once - nonce), M – Message (4 bit blocks), C- Cipher, B is a bit encryption function.

Cipher Block Chain - CBC

B = AddRoundKey(MixColums(SubBytes(ShiftRows(Message_block,Key))))

B^{-1} = ShiftRows^{-1}(SubBytes^{-1}(MixColums^{-1}(AddRoundKey((Cipher_block,Key))))

4 AES instruction provided by Intel AESNI x64

```
AESENC xmm1, xmm2/m128
Tmp := xmm1
Round Key := xmm2/m128
Tmp := ShiftRows (Tmp)
Tmp := SubBytes (Tmp)
Tmp := MixColumns (Tmp)
xmm1 := Tmp xor Round Key
```

```
AESDEC xmm1, xmm2/m128
Tmp := xmm1
Round Key := xmm2/m128
Tmp := InvShift Rows (Tmp)
Tmp := InvSubBytes (Tmp)
Tmp := InvMixColumns (Tmp)
xmm1 := Tmp xor Round Key
```

```
AESENCLAST xmm1, xmm2/m128
Tmp := xmm1
Round Key := xmm2/m128
Tmp := Shift Rows (Tmp)
Tmp := SubBytes (Tmp)
xmm1 := Tmp xor Round Key
```

```
AESDECLAST xmm1, xmm2/m128
State := xmm1
Round Key := xmm2/m128
Tmp := InvShift Rows (State)
Tmp := InvSubBytes (Tmp)
xmm1:= Tmp xor Round Key
```

Key expansion

AddRoundKey() This function uses exclusive or to combine the round key with the function. It has no inverse function as XOR is its own inverse.

Ten round keys are generated the original secret (128 bit) key. This is known as "Key expansion"

K=Key

$K_1 = B(K)$

$K_2 = B(K1)$

….

$K_{10}=B(K9)$

Encrypt

```
AESKEYGENASSIST xmm1, xmm2/m128, imm8
Tmp := xmm2/LOAD(m128)
X3[31-0] := Tmp[127-96];
X2[31-0] := Tmp[95-64];
X1[31-0] := Tmp[63-32];
X0[31-0] := Tmp[31-0];
RCON[7-0] := imm8;
RCON [31-8] := 0;
xmm1 := [RotWord (SubWord (X3)) XOR RCON,
SubWord (X3),
RotWord (SubWord (X1)) XOR RCON, SubWord
(X1)]
```

Takes previous expanded key (or key) and returns the next key

Extra step used for decrypt

```
AESIMC xmm1, xmm2/m128
RoundKey := xmm2/m128;
xmm1 := InvMixColumns (RoundKey)
```

Decryption requires keys to be pre-processed before use

Definitive guide to AESNI: Intel® Advanced Encryption Standard (AES) New Instructions Set

https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

Key expansion

`AESKEYGENASSIST xmm1, xmm2/m128, imm8`

X3 = msb 32 bits, X2, X1, X0 lsb 32 bits

xmm1 := [RotWord (SubWord (X3)) XOR RCON, SubWord (X3), RotWord (SubWord (X1)) XOR RCON , SubWord (X1)]

RotWord (X [31-0]) = [X[7-0], X [31-24], X [23-16], X [15-8]]

(or in C language notation, RotWord(X) = (X >> 8) | (X << 24))

Round constant (3 parameter)

RCON [1] = 0x01, RCON [2] = 0x02, RCON [3] = 0x04, RCON [4] = 0x08, RCON [5] = 0x10,
RCON [6] = 0x20, RCON [7] = 0x40, RCON [8] = 0x80, RCON [9] = 0x1B, RCON [10] = 0x36

SubWord (X) = [S-Box(X[31-24]), S-Box(X[23-16]), S-Box(X[15-8]), S-Box(X[7-0])]

Sub-Word (lookup table)

S-Box lookup table

Low nibble

<---------------------- y ---------------------->

High nibble

		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
^	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
x	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
V	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

X0 = 09->01
X1 = 6A ->02
X2 = D5->03
X3 = 30 ->04

X0 = E3->11
X1 = 39 ->12
X2 = 82->13
X3 = 9B ->14

X0 = 7B->21
X1 = 94 ->22
X2 = 32->23
X3 = A6 ->24

X0 = 2E->31
X1 = A1 ->32
X2 = 66->33
X3 = 28 ->34

```c
#if defined(_MSC_VER)
#define _ALIGNED(x) __declspec(align(x))
#endif
#include <stdio.h>
#include <string.h>
#include <stdint.h>

extern "C" void pass_xmm(uint32_t *, uint32_t *);

uint32_t _ALIGNED(32)xmm_in[12] =
{
  0xF3020100, 0x07060504, 0x0B0A0908, 0x0F0E0D0C,
  0x22222222, 0x11111111, 0x00000000, 0xFFFFFFFF,
  0x11112222, 0x33334444, 0x55556666, 0x77778888
};
uint32_t _ALIGNED(32)xmm_out[12] ={0};

void print_xmm(uint32_t buf[], size_t x)
{
    printf("\n       MSB 127                                LSB 0");
    for (size_t i = 0; i < x; i++)
    {
      printf("\n XMM%d: ",(int)i);
      for (size_t j = 4; j > 0; j--)
      {
          printf("0x%08lX", buf[j - 1 + (i * 4)]);
          if (j != 1)printf(", ");
      }
    }
    putchar('\n');
}

int main(void)
{
    pass_xmm(xmm_in,xmm_out);

    printf("Input");
    print_xmm(xmm_in,2);
    printf("\n          Instruction: pshufd   xmm1, xmm0, 240\n");
    printf("\nOutput");
    print_xmm(xmm_out,2);

    return 0;
}
```

Sample code to pass 3 x 128 values to and from a ASM program. Values in C contained in 12 x 32 bit numbers arrive in ASM as xmm0, xmm1, xmm2.

```asm
; Parameters passed from C function (1st, 2nd)
; 1st inputArray address in RCX.
; 2nd outputArray address in RDX

.code

public pass_xmm

pass_xmm:
movdqa     xmm0, [rcx+ 0*16]
movdqa     xmm1, [rcx+ 1*16]
movdqa     xmm2, [rcx+ 2*16]

paddq xmm0, xmm1   ; add packed quad word integers 2x64 bit

movdqa [rdx+ 0*16], xmm0;
movdqa [rdx+ 1*16], xmm1;
movdqa [rdx+ 2*16], xmm2;

ret ;

end
```

Example add packed quad word

```
Input
       MSB 127                                            LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222

          Instruction: paddq xmm0, xmm1

Output
       MSB 127                                            LSB 0
 XMM0: 0x0F0E0D0B, 0x0B0A0908, 0x18171616, 0x15242322
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222
```

AENSI -Advanced Encryption Standard New Instructions

```
Input
       MSB 127                                      LSB 0
 XMM0: 0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E

           Instruction: aeskeygenassist xmm0, xmm1, 1

Output
       MSB 127                                      LSB 0
 XMM0: 0x01040303, 0x04030201, 0x21242323, 0x24232221
```

aeskeygenassist xmm0, xmm0, 1

```
   X3         X2         X1         X0       xmm0 - Source
30D56A09   9B8239E3   A632947B   2866A12E    Input
   V          V          V          V        Lookup table
04030201   14131211   24232221   34333231
RotWord>              RotWord>               Rotate X3 and X1
01040302              21242322
 EXOR(1)              EXOR(1)                Exor X3 and X1 with 1
01040303              21242323               Gather output
   V                     V
01040303   04030201   21242323   24232221    xmm0 - destination
```

xmm1 := [RotWord (SubWord (X3)) XOR RCON, SubWord (X3), RotWord (SubWord (X1)) XOR RCON, SubWord (X1)]

AENSI -Advanced Encryption Standard New Instructions

```
Input
       MSB 127                                    LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222

           Instruction: paddq xmm0, xmm1

Output
       MSB 127                                    LSB 0
 XMM0: 0x0F0E0D0B, 0x0B0A0908, 0x18171616, 0x15242322
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222
```

xmm0 = xmm0 + xmm1

```
Input
       MSB 127                                    LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222

           Instruction: pxor     xmm0, xmm1

Output
       MSB 127                                    LSB 0
 XMM0: 0xF0F1F2F3, 0x0B0A0908, 0x16171415, 0xD1202322
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222
```

xmm0 = xmm0 exor xmm1

paddq
add as two separate quad word (2x64bits)

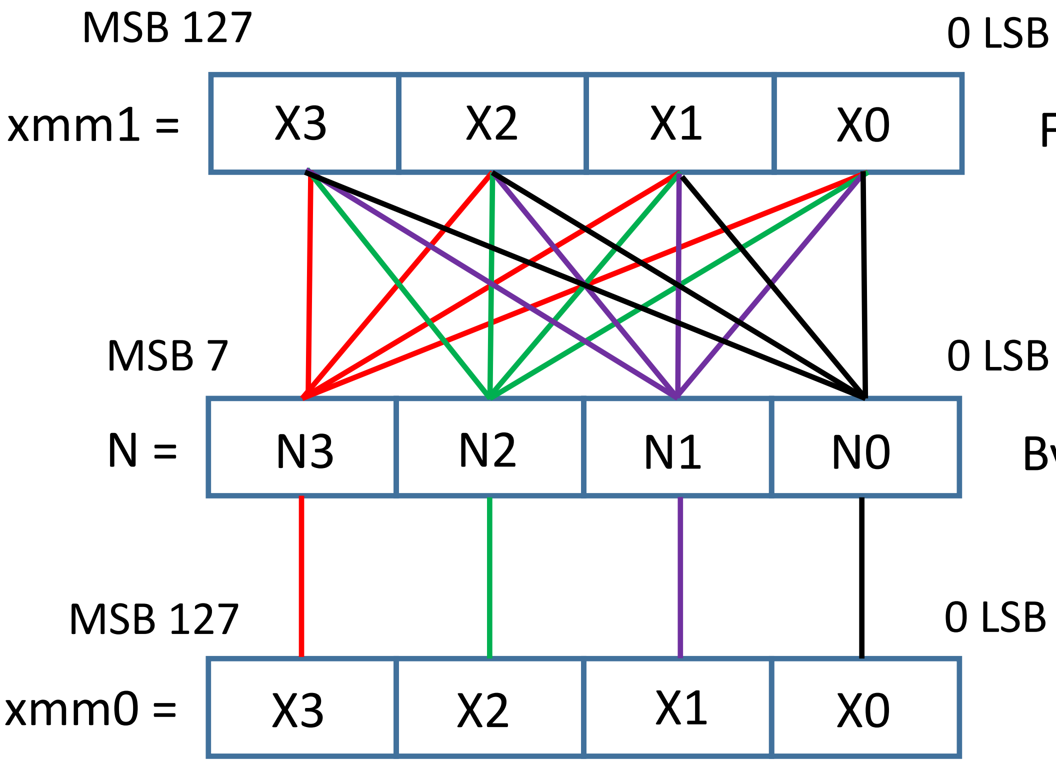
paddd
add as four separate double word (4x32bits)

paddw
add as eight separate word (8x16bits)

addb
add as 16 separate bytes (16x8bits)

pxor
Bitwise exclusive or 128 bits

```
pshufd   xmm0, xmm1, N   ; packed shuffle double word
```



Four long unsigned integers (each 32 bit)

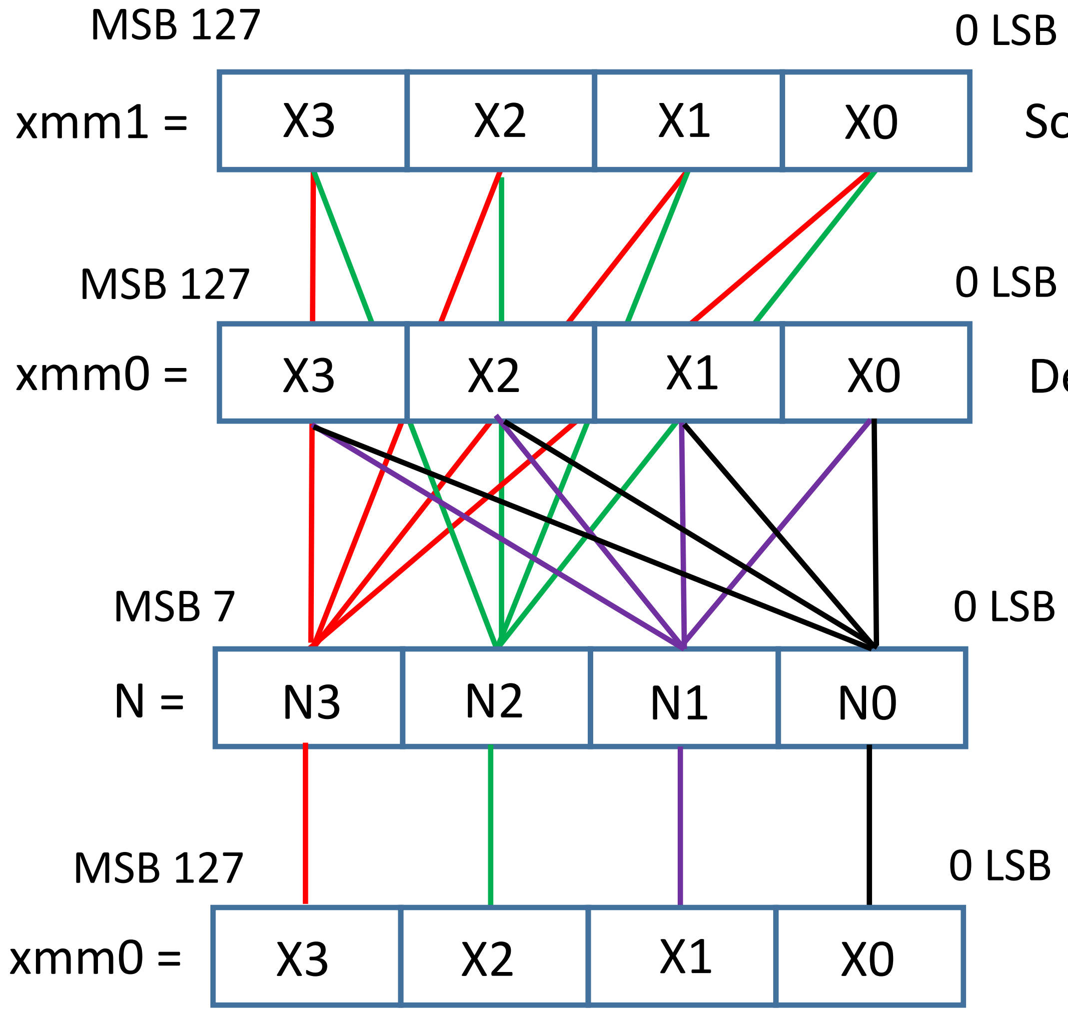
Byte, each pair of bits specified the destination in xmm1

```
Input
        MSB 127                                           LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222

            Instruction: pshufd   xmm1, xmm0, 240

                                    (240 = 11 11 00 00)
Output
        MSB 127                                           LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0x0F0E0D0C, 0x0F0E0D0C, 0xF3020100, 0xF3020100
```

```
shufps   xmm0, xmm1, N  ; Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values
```




```
Input
       MSB 127                                          LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222

           Instruction: shufps xmm0, xmm1, 27

Output
       MSB 127                                          LSB 0
 XMM0: 0x22222222, 0x11111111, 0x0B0A0908, 0x0F0E0D0C
 XMM1: 0xFFFFFFFF, 0x00000000, 0x11111111, 0x22222222
```

(27 = 00 01 10 11)

Destination – X3, X2 from xmm1, X1, X0 from xmm0

AENSI -Advanced Encryption Standard New Instructions

```
Input
       MSB 127                                  LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFF1234, 0x00000000, 0x11111111, 0x22222222
           3           2           1           0
           Instruction: shufps   xmm0, xmm1, 140

Output
       MSB 127  2        0        3        0    LSB 0
 XMM0: 0x00000000, 0x22222222, 0x0F0E0D0C, 0xF3020100
 XMM1: 0xFFFF1234, 0x00000000, 0x11111111, 0x22222222
```

shufps xmm0, xmm1, 140

140 = 10,00, 11 00 = 2, 0, 3, 0

```
Input
       MSB 127                                  LSB 0
 XMM0: 0x0F0E0D0C, 0x0B0A0908, 0x07060504, 0xF3020100
 XMM1: 0xFFFF1234, 0x00000000, 0x11111111, 0x22222222
           3           2           1           0
           Instruction: shufps   xmm0, xmm1, 16

Output
       MSB 127  0        1        0        0    LSB 0
 XMM0: 0x22222222, 0x11111111, 0xF3020100, 0xF3020100
 XMM1: 0xFFFF1234, 0x00000000, 0x11111111, 0x22222222
```

shufps xmm0, xmm1, 16

16= 00, 01, 00 00 = 0, 1, 0, 0

Aside: Key generation code can vary in implementation but functionally it is identical

```
71 total (42 highlighted) bytes)
00007FF6F2071AE0 66 0F 6F 01          movdqa      xmm0,xmmword ptr [rcx]
00007FF6F2071AE4 66 0F 6F 49 10       movdqa      xmm1,xmmword ptr [rcx+10h]
00007FF6F2071AE9 66 0F 6F 51 20       movdqa      xmm2,xmmword ptr [rcx+20h]

00007FF6F2071AEE 66 0F 3A DF C8 01    aeskeygenassist xmm1,xmm0,1
00007FF6F2071AF4 66 0F 70 C9 FF       pshufd      xmm1,xmm1,0FFh
00007FF6F2071AF9 C5 E9 73 F8 04       vpslldq     xmm2,xmm0,4
00007FF6F2071AFE 66 0F EF C2          pxor        xmm0,xmm2
00007FF6F2071B02 C5 E9 73 F8 04       vpslldq     xmm2,xmm0,4
00007FF6F2071B07 66 0F EF C2          pxor        xmm0,xmm2
00007FF6F2071B0B C5 E9 73 F8 04       vpslldq     xmm2,xmm0,4
00007FF6F2071B10 66 0F EF C2          pxor        xmm0,xmm2
00007FF6F2071B14 66 0F EF C1          pxor        xmm0,xmm1

00007FF6F2071B18 66 0F 7F 02          movdqa      xmmword ptr [rdx],xmm0
00007FF6F2071B1C 66 0F 7F 4A 10       movdqa      xmmword ptr [rdx+10h],xmm1
00007FF6F2071B21 66 0F 7F 52 20       movdqa      xmmword ptr [rdx+20h],xmm2
00007FF6F2071B26 C3                   ret
```

```
60 total (31 highlighted) bytes)
00007FF613441AE0 66 0F 6F 01          movdqa      xmm0,xmmword ptr [rcx]
00007FF613441AE4 66 0F 6F 49 10       movdqa      xmm1,xmmword ptr [rcx+10h]
00007FF613441AE9 66 0F 6F 51 20       movdqa      xmm2,xmmword ptr [rcx+20h]

00007FF613441AEE 66 0F 3A DF C8 01    aeskeygenassist xmm1,xmm0,1
00007FF613441AF4 66 0F 70 C9 FF       pshufd      xmm1,xmm1,0FFh
00007FF613441AF9 0F C6 D0 10          shufps      xmm2,xmm0,10h
00007FF613441AFD 66 0F EF C2          pxor        xmm0,xmm2
00007FF613441B01 0F C6 D0 8C          shufps      xmm2,xmm0,8Ch
00007FF613441B05 66 0F EF C2          pxor        xmm0,xmm2
00007FF613441B09 66 0F EF C1          pxor        xmm0,xmm1

00007FF613441B0D 66 0F 7F 02          movdqa      xmmword ptr [rdx],xmm0
00007FF613441B11 66 0F 7F 4A 10       movdqa      xmmword ptr [rdx+10h],xmm1
00007FF613441B16 66 0F 7F 52 20       movdqa      xmmword ptr [rdx+20h],xmm2
00007FF613441B1B C3                   ret
```

Aside: Key generation code can vary in implementation but functionally it is identical

```
                                           xmm0                                                  xmm1                                                    xmm2
New AVX                    0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x77778888, 0x55556666, 0x33334444, 0x11112222 - 0x00000000, 0x00000000, 0x00000000, 0x00000000
aeskeygenassist xmm1,xmm0,1 0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x01040303, 0x04030201, 0x21242323, 0x24232221 - 0x00000000, 0x00000000, 0x00000000, 0x00000000
pshufd    xmm1, xmm1, 255  0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x00000000, 0x00000000, 0x00000000, 0x00000000
vpslldq xmm2, xmm0, 4      0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x9B8239E3, 0xA632947B, 0x2866A12E, 0x00000000
pxor xmm0, xmm2            0xAB5753EA, 0x3DB0AD98, 0x8E543555, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x9B8239E3, 0xA632947B, 0x2866A12E, 0x00000000
vpslldq xmm2, xmm0, 4      0xAB5753EA, 0x3DB0AD98, 0x8E543555, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x3DB0AD98, 0x8E543555, 0x2866A12E, 0x00000000
pxor xmm0, xmm2            0x96E7FE72, 0xB3E498CD, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x3DB0AD98, 0x8E543555, 0x2866A12E, 0x00000000
vpslldq xmm2, xmm0, 4      0x96E7FE72, 0xB3E498CD, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0xB3E498CD, 0xA632947B, 0x2866A12E, 0x00000000
pxor xmm0, xmm2            0x250366BF, 0x15D60CB6, 0x8E543555, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0xB3E498CD, 0xA632947B, 0x2866A12E, 0x00000000
pxor xmm0, xmm1            0x240765BC, 0x14D20FB5, 0x8F503656, 0x2962A22D - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0xB3E498CD, 0xA632947B, 0x2866A12E, 0x00000000

Old SSE
aeskeygenassist xmm1,xmm0,1 0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x77778888, 0x55556666, 0x33334444, 0x11112222 - 0x00000000, 0x00000000, 0x00000000, 0x00000000
pshufd    xmm1, xmm1, 255  0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x01040303, 0x04030201, 0x21242323, 0x24232221 - 0x00000000, 0x00000000, 0x00000000, 0x00000000
shufps    xmm2, xmm0, 16   0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x2866A12E, 0xA632947B, 0x00000000, 0x00000000
pxor      xmm0, xmm2       0x18B3CB27, 0x3DB0AD98, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x2866A12E, 0xA632947B, 0x00000000, 0x00000000
shufps    xmm2, xmm0, 140  0x18B3CB27, 0x3DB0AD98, 0xA632947B, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x3DB0AD98, 0x2866A12E, 0x2866A12E, 0x00000000
pxor      xmm0, xmm2       0x250366BF, 0x15D60CB6, 0x8E543555, 0x2866A12E - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x3DB0AD98, 0x2866A12E, 0x2866A12E, 0x00000000
pxor      xmm0, xmm1       0x240765BC, 0x14D20FB5, 0x8F503656, 0x2962A22D - 0x01040303, 0x01040303, 0x01040303, 0x01040303 - 0x3DB0AD98, 0x2866A12E, 0x2866A12E, 0x00000000
```

```
Input
       MSB 127                                         LSB 0
 XMM0: 0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E
 XMM1: 0xF8F7F6F5, 0xF4F3F2F1, 0x08070605, 0x04030201

           Instruction:   Old: shufps

Output
       MSB 127                                         LSB 0
 XMM0: 0x240765BC, 0x14D20FB5, 0x8F503656, 0x2962A22D
 XMM1: 0x01040303, 0x01040303, 0x01040303, 0x01040303
```

```
Input
       MSB 127                                         LSB 0
 XMM0: 0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E
 XMM1: 0xF8F7F6F5, 0xF4F3F2F1, 0x08070605, 0x04030201

           Instruction:   New: vpslldq

Output
       MSB 127                                         LSB 0
 XMM0: 0x240765BC, 0x14D20FB5, 0x8F503656, 0x2962A22D
 XMM1: 0x01040303, 0x01040303, 0x01040303, 0x01040303
```

Encryption, C

```c
void Encrypt(uint32_t ek[][4], uint8_t iv[], uint8_t data[], uint8_t code[], size_t x)
{
    uint32_t nonce[1][4];
    bytes_to_words(iv, nonce, 1);     // 1 x 4 words
    bytes_to_words(data, data_in, x); // N x 4 words (x is the number of 128 bit words (groups of aligned 16 bytes)

    for (uint32_t j = 0; j < x; j++)
    {
        // Round key 0;
        Copy(&data_in[j][0], &data_out[j][0]);                  // First line of new  data copied to data_out
        if (j == 0)
        {
            EXOR(&nonce[0][0], &data_out[j][0]); }              // EXOR with nonce for first line only
        else
        {
            EXOR(&data_out[j - 1][0], &data_out[j][0]);         // EXOR with previous line for all other lines
        }
        AddRoundKey(&data_out[j][0], expanded_key, 0);          // EXOR with original key

        for (uint32_t i = 1; i <=10; i++)                       // Round keys 1-9
        {
            Shift_Rows(&data_out[j][0], &data_out[j][0], 1);    // Shift rows
            SubBytes(&data_out[j][0], &data_out[j][0], 1);      // SubBytes
            if (i != 10)                                        // MixColumns
            {
                Mix_Columms(&data_out[j][0], &data_out[j][0], 1);
            }
            AddRoundKey(&data_out[j][0], expanded_key, i);      // Add roundKey 1 to 10
        }
    }
}
```

Decryption, C

```
void Decrypt(uint32_t ek[][4], uint8_t iv[], uint8_t code[], uint8_t data[], size_t x)
{
    uint32_t mmx[1][4];
    uint32_t nonce[1][4];
    bytes_to_words(iv, nonce, 1);      // 1 x 4 words
    bytes_to_words(code, data_in, x); // N x 4 words

    uint32_t i_expanded_key[11][4] = { 0 }; //aesimc of keys[1-10] but not k[0]
    for (uint32_t j = 0; j < 11; j++) Copy(&expanded_key[j][0], &i_expanded_key[j][0]);
    for (uint32_t j = 1; j < 10; j++) Mix_Columms(&i_expanded_key[j][0], &i_expanded_key[j][0], -1);

    for (uint32_t j = 0; j < x; j++)
    {
        Copy(&data_in[j][0], &data_out[j][0]);                    // Round 0, First line of new  data copied to data_out
        AddRoundKey(&data_out[j][0], i_expanded_key, 10);         // EXOR with original key, key 0

        for (int32_t i = 9; i >= 0; i--)                          // Round keys 1-9
        {
            Shift_Rows(&data_out[j][0], &data_out[j][0], -1);  // Shift rows
            SubBytes(&data_out[j][0], &data_out[j][0], -1);    // SubBytes
            if (i != 0) Mix_Columms(&data_out[j][0], &data_out[j][0], -1); // MixColumns
            AddRoundKey(&data_out[j][0], i_expanded_key, i);    // Add roundKey
        }

        if   (j == 0) EXOR(&nonce[0][0], &data_out[j][0]);       // EXOR with nonce for first line only
        else          EXOR(&data_in[j-1][0], &data_out[j][0]);  // EXOR with previous line for all other lines
    }
    words_to_bytes(data_out, data, x);
}
```

Encryption, C

```
; Calling function aes_encrypt_cbc128(NB/16, text, key, iv);
; RCX=NB/16, RDX=&text/cipher[0],R8=&keys[0-10], R9=iv/nonce

aes_encrypt_cbc128:
    movdqa      xmm0, [r9]          ; Nonce
    movdqa      xmm1, [r8+ 0*16] ; Key 0
    movdqa      xmm2, [r8+ 1*16] ; Key 1
    movdqa      xmm3, [r8+ 2*16]
    movdqa      xmm4, [r8+ 3*16]
    movdqa      xmm5, [r8+ 4*16]
    movdqa      xmm6, [r8+ 5*16]
    movdqa      xmm7, [r8+ 6*16]
    movdqa      xmm8, [r8+ 7*16]
    movdqa      xmm9, [r8+ 8*16]
    movdqa      xmm10,[r8+ 9*16]
    movdqa      xmm11,[r8+10*16] ; Key 10

 xor          eax, eax            ; rax=0 (for each
encrypt_blocks:
    pxor        xmm0, [rdx+rax]   ; rdx=text/cipher,
    pxor        xmm0, xmm1        ; xmm0 exor key 0
    aesenc      xmm0, xmm2        ; aesenc key 1
    aesenc      xmm0, xmm3
    aesenc      xmm0, xmm4
    aesenc      xmm0, xmm5
    aesenc      xmm0, xmm6
    aesenc      xmm0, xmm7
    aesenc      xmm0, xmm8
    aesenc      xmm0, xmm9
    aesenc      xmm0, xmm10
    aesenclast xmm0, xmm11      ; last key
    movdqa      [rdx+rax], xmm0 ; cipher xmm0
                                ; overwrites text
    add         eax, 16         ; Move on 16 bytes,
128 bits
    loop        encrypt_blocks  ; CX=CX-1 jnz …
    ret
```

Decryption, ASM

RCX=NB/16, RDX=&text/cipher[0],R8=&keys[0-10], R9=iv/nonce

```
aes_decrypt_cbc128:
    movdqa      xmm1, [r8+ 0*16]
    movdqa      xmm2, [r8+ 1*16]
    aesimc      xmm2, xmm2
    movdqa      xmm3, [r8+ 2*16]
    aesimc      xmm3, xmm3
    movdqa      xmm4, [r8+ 3*16]
    aesimc      xmm4, xmm4
    movdqa      xmm5, [r8+ 4*16]
    aesimc      xmm5, xmm5
    movdqa      xmm6, [r8+ 5*16]
    aesimc      xmm6, xmm6
    movdqa      xmm7, [r8+ 6*16]
    aesimc      xmm7, xmm7
    movdqa      xmm8, [r8+ 7*16]
    aesimc      xmm8, xmm8
    movdqa      xmm9, [r8+ 8*16]
    aesimc      xmm9, xmm9
    movdqa      xmm10,[r8+ 9*16]
    aesimc      xmm10, xmm10
    movdqa      xmm11,[r8+10*16]
    movdqa      xmm12, [r9]
    xor         eax, eax
```

```
decrypt_blocks:
    movdqa      xmm0, [rdx+rax]
    movdqa      xmm13, xmm0
    pxor        xmm0, xmm11
    aesdec      xmm0, xmm10
    aesdec      xmm0, xmm9
    aesdec      xmm0, xmm8
    aesdec      xmm0, xmm7
    aesdec      xmm0, xmm6
    aesdec      xmm0, xmm5
    aesdec      xmm0, xmm4
    aesdec      xmm0, xmm3
    aesdec      xmm0, xmm2
    aesdeclast  xmm0, xmm1
    pxor        xmm0, xmm12
    movdqa      xmm12, xmm13
    movdqa      [rdx+rax], xmm0
    add         eax, 16
    loop        decrypt_blocks
    ret
```

Key expansion, ASM

```
Calling function aes_set_key128(key_value, key);
RCX=key, RDX=&expanded key[0]

aes_set_key128:
    movdqa   xmm0, [rcx]           ; input - 128 cipher key in xmm0
    movdqa   [rdx], xmm0           ; output
    add      rdx, 16

    aeskeygenassist xmm1, xmm0, 1 ; Create Round Key 1
    call     expand
    aeskeygenassist xmm1, xmm0, 2
    call     expand
    aeskeygenassist xmm1, xmm0, 4
    call     expand
    aeskeygenassist xmm1, xmm0, 8
    call     expand
    aeskeygenassist xmm1, xmm0, 16
    call     expand
    aeskeygenassist xmm1, xmm0, 32
    call     expand
    aeskeygenassist xmm1, xmm0, 64
    call     expand
    aeskeygenassist xmm1, xmm0, 128
    call     expand
    aeskeygenassist xmm1, xmm0, 27
    call     expand
    aeskeygenassist xmm1, xmm0, 54
    call     expand
    ret
```

```
expand:
    pshufd   xmm1, xmm1, 255

    ; old
    ;shufps   xmm4, xmm0, 16
    ;pxor     xmm0, xmm4
    ;shufps   xmm4, xmm0, 140

    vpslldq xmm2, xmm0, 4
    pxor xmm0, xmm2
    vpslldq xmm2, xmm0, 4
    pxor xmm0, xmm2
    vpslldq xmm2, xmm0, 4

    pxor     xmm0, xmm2
    pxor     xmm0, xmm1

    movdqa   [rdx], xmm0    ; Store new round key
    add      rdx, 16        ; rdx points to key last key
    ret
```

Key generation

Sample code for encryption and decryption using both C++ and ASM separately were developed. At each step outputs were compared to demonstrate equivalence. Key expansion comparison shown below. Code available on Moodle.

```
Key :
30D56A09 9B8239E3 A632947B 2866A12E

Expanded Key :
240765BC 14D20FB5 8F503656 2962A22D
F3D13B3D D7D65E81 C3045134 4C546762
8C5A6D0C 7F8B5631 A85D08B0 6B595984
CEB1D43D 42EBB931 3D60EF00 953DE7B0
038CADE4 CD3D79D9 8FD6C0E8 B2B62FE8
9AAA5F88 9926F26C 541B8BB5 DBCD4B5D
48E2C183 D2489E0B 4B6E6C67 1F75E7D2
22E34CC5 6A018D46 B849134D F3277F2A
A51FBCD6 87FCF013 EDFD7D55 55B46E18
6CAC9FDB C9B3230D 4E4FD31E A3B2AE4B
```

Keys generated by Assembly language program

```
Key
       MSB 127                                           LSB 0
 XMM0: 0x30D56A09, 0x9B8239E3, 0xA632947B, 0x2866A12E

Expanded Keys
       MSB 127                                           LSB 0
 XMM0: 0x240765BC, 0x14D20FB5, 0x8F503656, 0x2962A22D
 XMM1: 0xF3D13B3D, 0xD7D65E81, 0xC3045134, 0x4C546762
 XMM2: 0x8C5A6D0C, 0x7F8B5631, 0xA85D08B0, 0x6B595984
 XMM3: 0xCEB1D43D, 0x42EBB931, 0x3D60EF00, 0x953DE7B0
 XMM4: 0x038CADE4, 0xCD3D79D9, 0x8FD6C0E8, 0xB2B62FE8
 XMM5: 0x9AAA5F88, 0x9926F26C, 0x541B8BB5, 0xDBCD4B5D
 XMM6: 0x48E2C183, 0xD2489E0B, 0x4B6E6C67, 0x1F75E7D2
 XMM7: 0x22E34CC5, 0x6A018D46, 0xB849134D, 0xF3277F2A
 XMM8: 0xA51FBCD6, 0x87FCF013, 0xEDFD7D55, 0x55B46E18
 XMM9: 0x6CAC9FDB, 0xC9B3230D, 0x4E4FD31E, 0xA3B2AE4B
```

Keys generated by C program

Timings for AES encryption

Running AES_CPP on the computer completed the encryption and decryption of Shakespeare's works using C++ (code not optimised) in the following times

Encryption time: 329.596 ms (17.5 Mbytes/sec)

Decryption time: 554.576 ms (10.4 Mbytes/sec)

Using AES_ASM the same process was completed in

Encryption time: 7.069 ms (817.3 Mbytes/sec)

Decryption time: 1.406 ms (4109.1 Mbytes/sec)

Encrypted string assembly language program
"This is a test abcdefghijklmnopqrstuvwxyz"

```
Unencrypted :
61207473 65742061 20736920 73696854
71706F6E 6D6C6B6A 69686766 65646362
00000000 00000A7A 79787776 75747372
00000000 00000000 00000000 00000000

Encrypted :
BDA3AD96 8E87CF45 8E6021BE D19FBCC7
142CE483 7970368B 883C117F D0EA3B06
51A14209 DB20E170 0D6596D9 B17CADE5
DB7C02AB DA714ACC D689D2E4 CC6E6248
```

Assembly language is 46 times faster encrypting and decryption is 394 times faster.

Why do you think decryption is faster than encryption?