Use of the AES instruction set

(ECRYPT II AES Day - Bruges, Belgium)

Ryad BENADJILA

Agence Nationale de la Sécurité des Systèmes d'Information

18 October 2012

AES-NI

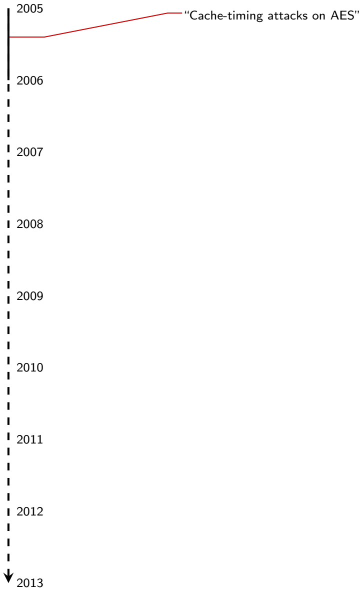
- AES-NI stands for "AES New Instructions"
- Introduced by Intel as:
  - A hardware accelerated implementation of AES subparts
  - Ways of implementing efficient versions of the algorithm with constant time operations, offering a mitigation against timing side channel attacks (especially cache based attacks)

AES-NI

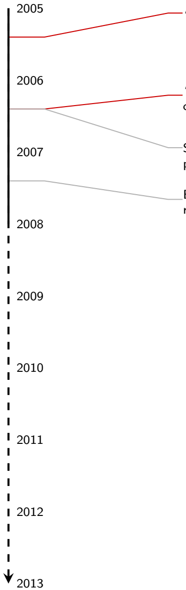
- AES-NI stands for “AES New Instructions”
- Introduced by Intel as:
  - A hardware accelerated implementation of AES subparts
  - Ways of implementing efficient versions of the algorithm with constant time operations, offering a mitigation against timing side channel attacks (especially cache based attacks)
- Access to the instructions from the userland (Ring3) level

AES-NI

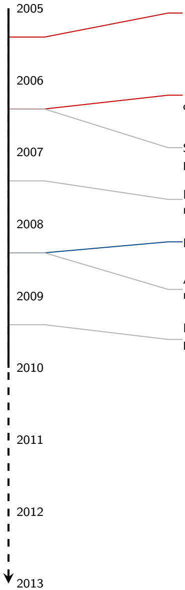
- AES-NI stands for "AES New Instructions"
- Introduced by Intel as:
  - A hardware accelerated implementation of AES subparts
  - Ways of implementing efficient versions of the algorithm with constant time operations, offering a mitigation against timing side channel attacks (especially cache based attacks)
- Access to the instructions from the userland (`Ring3`) level
- Six new instructions over the previous SSE4 set:
  - 4 for encryption and decryption: `aesenc`, `aesdec`, `aesenclast` and `aesdeclast`
  - 2 for the Key Schedule: `aeskeygenassist` and `aesimc`
- Plus a companion carry-less multiplication instruction `clmul`



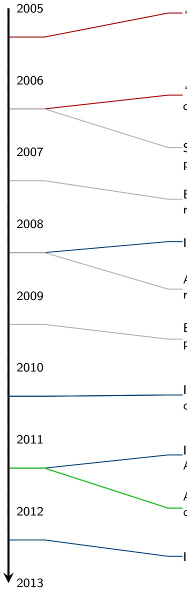
Bernstein's remote cache attacks on AES table based implementations



"Cache-timing attacks on AES"	Bernstein's remote cache attacks on AES table based implementations
"Cache Attacks and Countermeasures: The Case of AES"	Osvik, Shamir and Tromer local cache timing attacks
Software counter measures with performance impact	Table based implementations using masking
Bitslice implementations: interesting but efficiency related to block size	Mastsui and Nakajima: ∼10c/B for 2KB data blocks on Core 2 CPU



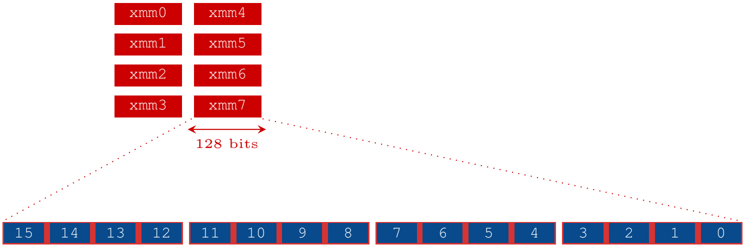
"Cache-timing attacks on AES"	Bernstein's remote cache attacks on AES table based implementations
"Cache Attacks and Countermeasures: The Case of AES"	Osvik, Shamir and Tromer local cache timing attacks
Software counter measures with performance impact	Table based implementations using masking
Bitslice implementations: interesting but efficiency related to block size	Mastsui and Nakajima: ∼10c/B for 2KB data blocks on Core 2 CPU
Intel's proposal for AES-NI instructions	White Paper by Gueron
AES table based implementation new speed records	Bernstein and Shwabe: ∼10c/B on Core 2 (using a custom high level assembler)
Bitslice implementations: very efficient SSSE3 implementation	Käsper and Shwabe: ∼7c/B on Core 2 and Nehalem Core i7 by using SSSE3 new instructions, for 128 bytes data blocks



"Cache-timing attacks on AES"	Bernstein's remote cache attacks on AES table based implementations
"Cache Attacks and Countermeasures: The Case of AES"	Osvik, Shamir and Tromer local cache timing attacks
Software counter measures with performance impact	Table based implementations using masking
Bitslice implementations: interesting but efficiency related to block size	Mastsui and Nakajima: ∼10c/B for 2KB data blocks on Core 2 CPU
Intel's proposal for AES-NI instructions	White Paper by Gueron
AES table based implementation new speed records	Bernstein and Shwabe: ∼10c/B on Core 2 (using a custom high level assembler)
Bitslice implementations: very efficient SSSE3 implementation	Käsper and Shwabe: ∼7c/B on Core 2 and Nehalem Core i7 by using SSSE3 new instructions, for 128 bytes data blocks
Intel's first generation CPUs with AES-NI (and clmul)	Westmere microarchitecture (not all CPUs concerned)
Intel's second generation CPUs with AES-NI, with AVX support	Sandy Bridge microarchitecture (almost all CPUs)
AMD's first generation CPUs with AES-NI (with clmul and AVX support)	Bulldozer microarchitecture (all CPUs)
Intel's third generation CPUs with AES-NI	Ivy Bridge microarchitecture (almost all CPUs)

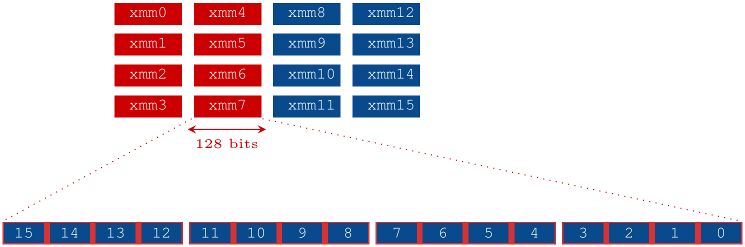
xmm and ymm registers

- xmm are 128-bit registers:
  - 8 in 32-bit mode



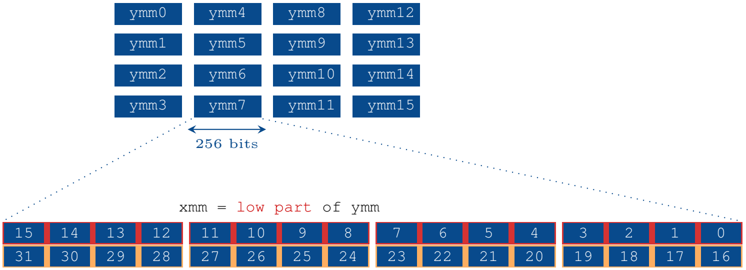
xmm and ymm registers

- xmm are 128-bit registers:
  - 16 in 64-bit mode



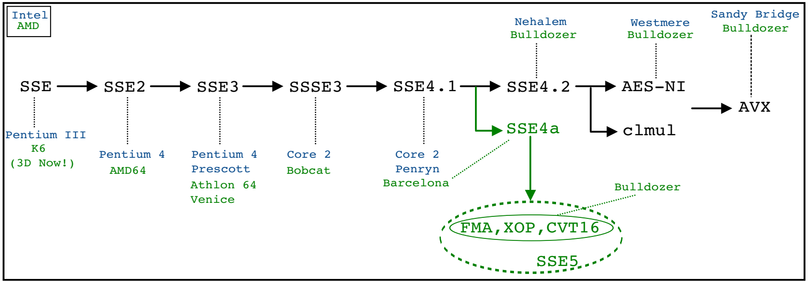
xmm and ymm registers

- ymm are 256-bit registers (only in 64-bit mode):
  - xmm extended to 256 bits with AVX new extensions



Some useful SSE instructions

- SSE = Streaming SIMD (Single Instruction Multiple Data) Extensions



Some useful SSE instructions

- SSE instructions work on bytes, 16-bit shorts, 32-bit double words, 64-bit quad words and full 128-bit xmm words

- Moving memory data to and from a xmm register:

movdqu xmm1/[mem128], [mem128]/xmm2

```
xmm1 ← [mem128]
 or
[mem128] ← xmm2
```

- “Xoring” two registers or a register and memory:

pxor xmm1, xmm2/[mem128]

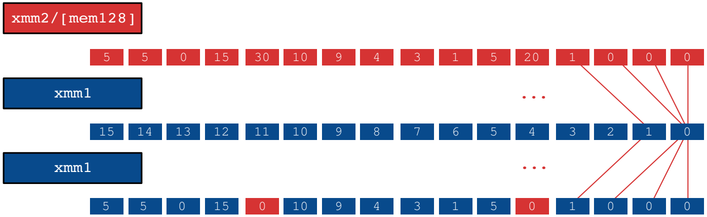
$\text{xmm1} \leftarrow \text{xmm1} \oplus (\text{xmm2}/[\text{mem128}])$

Some useful SSE instructions

- Packed Shuffle Bytes: byte-wise shuffling in xmm according to a mask in xmm (SSSE3)

pshufb xmm1, xmm2/[mem128]

```
for(i=0; i<16; i++){
  xmm1[i] ← xmm1[xmm2[i]]
}
/* With xmm1[i] = 0 for i ≥ 16 */
```

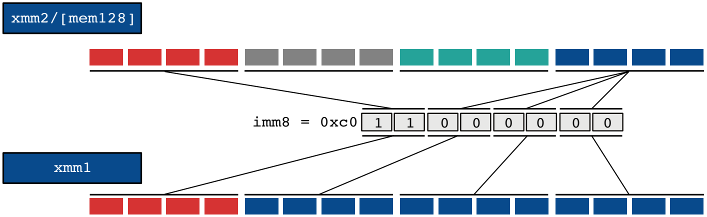


Some useful SSE instructions

- Packed Shuffle Double words: shuffling in xmm according to an immediate bitmask (SSE2)

pshufd xmm1, xmm2/[mem128], imm8

```
for(i=0; i<4; i++){
  (double word)xmm1[i] ← (double word)xmm2[(imm8>>(2*i)) & 0x3]
}
```

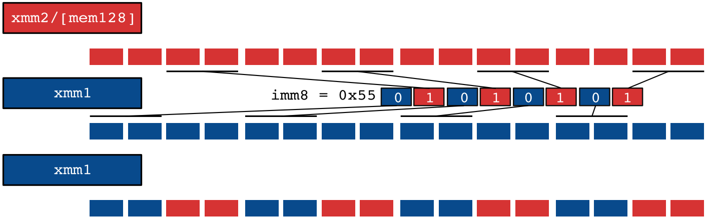


Some useful SSE instructions

- Blending two 16-bit words xmm registers according to a mask (SSE4.1):

pblendw xmm1, xmm2/[mem128], imm8

```
for(i=0; i<8; i++){
  if((imm8>>8) & 0x1 == 1){
    (short word)xmm1[i] ← (short word)xmm2[i]
  }
}
```



AVX extensions

- AVX extensions use the VEX prefix that is not compatible with 32-bit mode

AVX extensions

- AVX extensions use the VEX prefix that is not compatible with 32-bit mode
- Two main advantages over previous SSE:
  - Twice more data in ymm, which means twice more "vectorization"
  - New AVX extensions allow most compatible instructions to use 3 operands ⇒ non destructive operations

Legacy pxor

```
pxor xmm1, xmm2/[mm128]
```

$\text{xmm1} \leftarrow \text{xmm1} \oplus (\text{xmm2}/[\text{mem128}])$

AVX extended vpxor

```
vpxor xmm1, xmm2, xmm3/[mm128]
```

$\text{xmm1} \leftarrow \text{xmm2} \oplus (\text{xmm3}/[\text{mem128}])$

AVX extensions

- AVX extensions use the VEX prefix that is not compatible with 32-bit mode
- Two main advantages over previous SSE:
  - Twice more data in ymm, which means twice more "vectorization"
  - New AVX extensions allow most compatible instructions to use 3 operands ⇒ non destructive operations

Legacy pxor

```
pxor xmm1, xmm2/[mm128]
```

$xmm1 \leftarrow xmm1 \oplus (xmm2/[mem128])$

AVX extended vpxor

```
vpxor xmm1, xmm2, xmm3/[mm128]
```

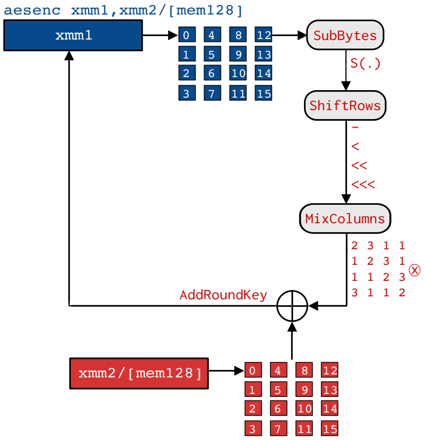
$xmm1 \leftarrow xmm2 \oplus (xmm3/[mem128])$

- However:
  - Not all legacy instructions with VEX extension benefit from ymm (e.g. vpshufd does, vpxor doesn't)
    ⇒ ymm high part is zeroed then
  - Possible latencies during AVX and legacy SSE switch

AES-NI encryption instructions

- aesenc for rounds:

```
aesenc xmm1, xmm2/[mem128]

Tmp  ← xmm1
Tmp  ← SubBytes(Tmp)
Tmp  ← ShiftRows(Tmp)
Tmp  ← MixColumns(Tmp)
xmm1 ← Tmp ⊕ xmm2/[mem128]
```

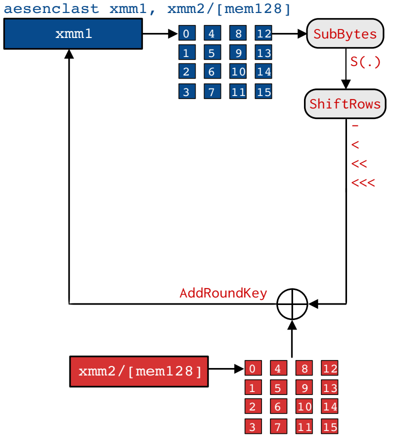


AES-NI encryption instructions

- aesenclast for the last round:

aesenclast xmm1, xmm2/[mem128]

```
Tmp  ← xmm1
Tmp  ← SubBytes(Tmp)
Tmp  ← ShiftRows(Tmp)
xmm1 ← Tmp ⊕ xmm2/[mem128]
```



Block encryption

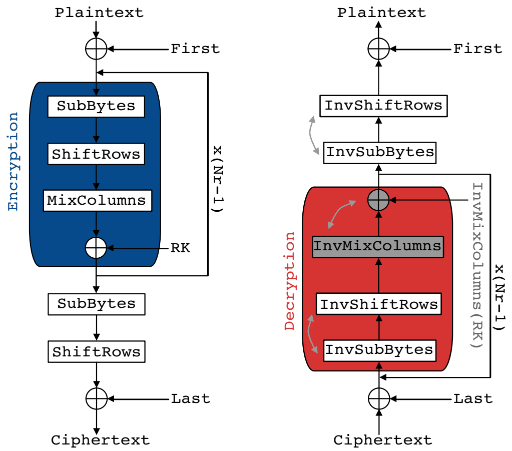
- AES128 (naive) encryption of one plaintext block:

AES128 Encryption (128-bit block)

```
xmm0         ← plaintext
xmm1—xmm11 ← scheduled keys
pxor    xmm0, xmm1       /* Round 0 (whitening) */
aesenc xmm0, xmm2       /* Round 1             */
aesenc xmm0, xmm3       /* Round 2             */
aesenc xmm0, xmm4       /* Round 3             */
aesenc xmm0, xmm5       /* Round 4             */
aesenc xmm0, xmm6       /* Round 5             */
aesenc xmm0, xmm7       /* Round 6             */
aesenc xmm0, xmm8       /* Round 7             */
aesenc xmm0, xmm9       /* Round 8             */
aesenc xmm0, xmm10      /* Round 9             */
aesenclast xmm0, xmm11 /* Round 10            */
```

AES-NI decryption instructions

- AES-NI implements the equivalent inverse cipher for decryption



AES-NI decryption instructions

- aesdec:

aesdec xmm1, xmm2/[mem128]

```
Tmp  ← xmm1
Tmp  ← SubBytes^-1(Tmp)
Tmp  ← ShiftRows^-1(Tmp)
Tmp  ← MixColumns^-1(Tmp)
xmm1 ← Tmp ⊕ xmm2/[mem128]
```

- aesdeclast:

aesdeclast xmm1, xmm2/[mem128]

```
Tmp  ← xmm1
Tmp  ← SubBytes^-1(Tmp)
Tmp  ← ShiftRows^-1(Tmp)
xmm1 ← Tmp ⊕ xmm2/[mem128]
```

- We feed aesdec with the equivalent inverse cipher keys

Block decryption

- AES128 (naive) decryption of one plaintext block:

AES128 Encryption (128-bit block)

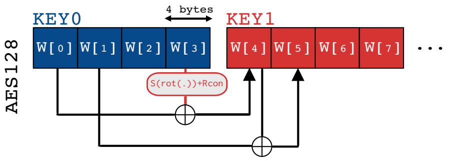
```
xmm0         ← plaintext
xmm1–xmm11 ← scheduled keys (inverse cipher)
pxor    xmm0, xmm1       /* Round 0 (whitening) */
aesdec xmm0, xmm2       /* Round 1             */
aesdec xmm0, xmm3       /* Round 2             */
aesdec xmm0, xmm4       /* Round 3             */
aesdec xmm0, xmm5       /* Round 4             */
aesdec xmm0, xmm6       /* Round 5             */
aesdec xmm0, xmm7       /* Round 6             */
aesdec xmm0, xmm8       /* Round 7             */
aesdec xmm0, xmm9       /* Round 8             */
aesdec xmm0, xmm10      /* Round 9             */
aesdeclast xmm0, xmm11 /* Round 10            */
```

Rijndael Key Schedule

- Key Schedule for AES128 and AES192

Rijndael Key Schedule (Nk≤6, i.e. AES128 and AES192)

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{ /* AES128 => (Nk=4, Nr=10, Nb=4) */
  for(i = 0; i < Nk; i++)
    W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
  for(i = Nk; i < Nb * (Nr + 1); i++)
  {
    temp = W[i - 1];
    if (i % Nk == 0)
      temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
    W[i] = W[i - Nk] ^ temp;
  }
}
```

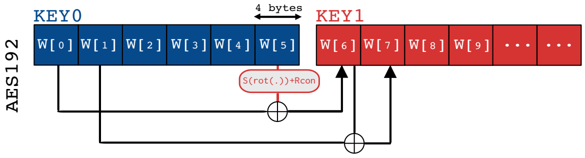


Rijndael Key Schedule

- Key Schedule for AES128 and AES192

Rijndael Key Schedule (Nk≤6, i.e. AES128 and AES192)

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{ /* AES192 => (Nk=6, Nr=12, Nb=4) */
  for(i = 0; i < Nk; i++)
    W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
  for(i = Nk; i < Nb * (Nr + 1); i++)
  {
    temp = W[i - 1];
    if (i % Nk == 0)
      temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
    W[i] = W[i - Nk] ^ temp;
  }
}
```

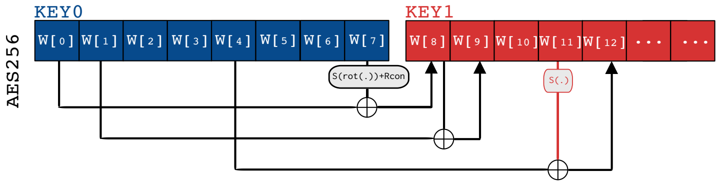


Rijndael Key Schedule

- Key Schedule for AES256

Rijndael Key Schedule (Nk>6, i.e. AES256)

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{ /* AES256 => (Nk=8, Nr=14, Nb=4) */
  for(i = 0; i < Nk; i++)
    W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
  for(i = Nk; i < Nb * (Nr + 1); i++)
  {
    temp = W[i - 1];
    if (i % Nk == 0)
      temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
    else if (i % Nk == 4)
      temp = SubByte(temp);
    W[i] = W[i - Nk] ^ temp;
  }
}
```



AES-NI Key Schedule instructions

- aeskeygenassist:

```
aeskeygenassist xmm1, xmm2/[128], imm8

xmm2  := [xmm2[3]|xmm2[2]|xmm2[1]|xmm2[0]]
/* split xmm2 in 4 bytes words */
xmm1  ←  [SubByte(RotByte(xmm2[3]))⊕imm8
          |SubByte(xmm2[3])
          |SubByte(RotByte(xmm2[1]))⊕imm8
          |SubByte(xmm2[1])]
```

- aesimc: for the equivalent inverse cipher key schedule (apply inverse MixColumns to all the keys scheduled for encryption, except first and last ones)

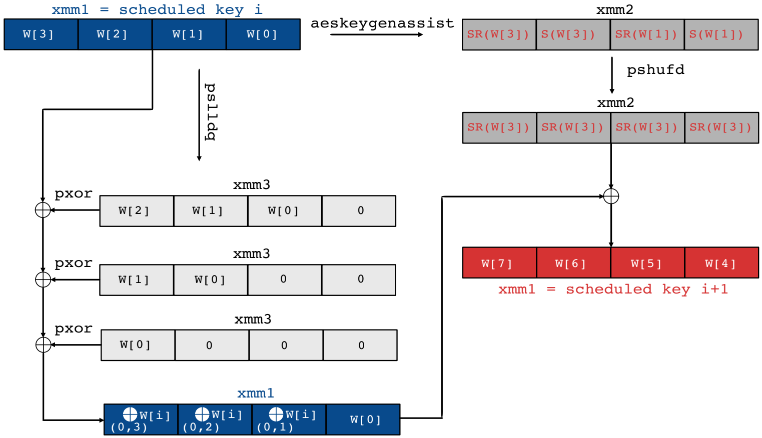
```
aesimc xmm1, xmm2/[mem128]

/* Round key scheduled for encryption in xmm2 */
Tmp  ← xmm2/[mem128]
xmm1 ← MixColumns^-1(Tmp)
```

AES-NI Key Schedule for AES128

AES-NI Key Schedule for AES128

```
/* Key in xmm1 */
xmm1  ← Key
/* Prepare value for W[4] in xmm2 */
aeskeygenassist xmm2, xmm1, Rcon /* Rcon=0x1 for the first iteration */
/* We only keep the last double word SubByte(RotByte(xmm2[3]))⊕Rcon */
pshufd xmm2, xmm2, 0xff
//
movdqa xmm3, xmm1
pslldq xmm3, 0x4 /* W[i−1] goes to W[i] place */
pxor xmm1, xmm3  /* xor all W[i] with W[i−1]  */
//
pslldq xmm3, 0x4 /* W[i−2] goes to W[i] place */
pxor xmm1, xmm3  /* xor all W[i] with W[i−2]  */
//
pslldq xmm3, 0x4 /* W[i−3] goes to W[i] place */
pxor xmm1, xmm3  /* xor all W[i] with W[i−3]  */
/* Finalize */
pxor xmm1, xmm2
//
KeySchedule[16*i]  ← xmm1
LOOP /* loop with next Rcon */
...
```

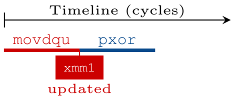
AES-NI Key Schedule for AES128



Some definitions

- Interdependent instructions: scheduled instructions that share a data dependency forcing a “stall” in the pipeline

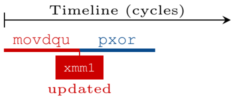
```
movdqu  xmm1, xmm2
pxor    xmm3, xmm1
```



Some definitions

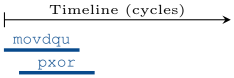
- Interdependent instructions: scheduled instructions that share a data dependency forcing a “stall” in the pipeline

```
movdqu  xmm1, xmm2
pxor    xmm3, xmm1
```



- Independent instructions: scheduled instructions that don’t share data dependency, and that can be parallelized

```
movdqu  xmm1, xmm2
pxor    xmm3, xmm4
```

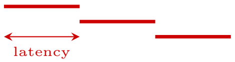


Some definitions

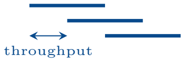
- Latency of an instruction: number of cycles taken by the instruction to complete in the worst case

Some definitions

- Latency of an instruction: number of cycles taken by the instruction to complete in the worst case

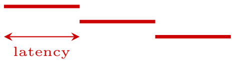


- (Reciprocal) Throughput of an instruction: number of cycles to complete in the best case

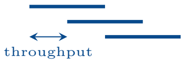


Some definitions

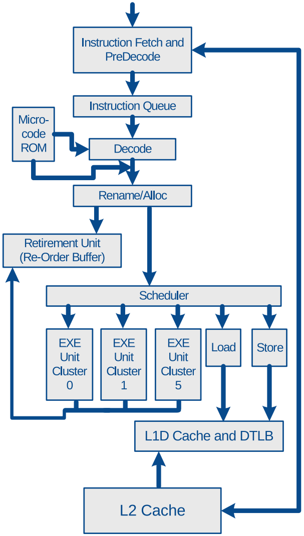
- Latency of an instruction: number of cycles taken by the instruction to complete in the worst case



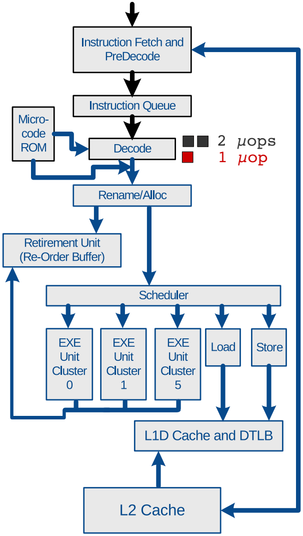
- (Reciprocal) Throughput of an instruction: number of cycles to complete in the best case



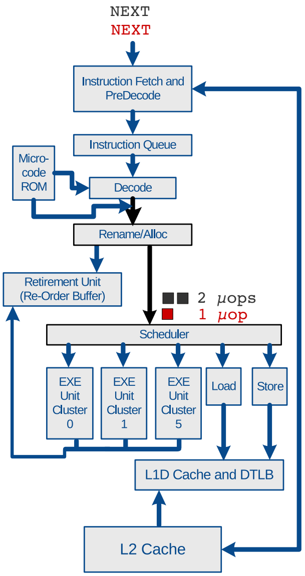
- Intel's Optimization Manual states that `aesenc`, `aesdec`, `aesenclast`, `aesdeclast` have:
  - latency of 6 cycles, throughput of 2 cycles (Westmere)
  - latency of 8 cycles, throughput of 1 cycle (Sandy and Ivy Bridge)
  - let's understand why ...

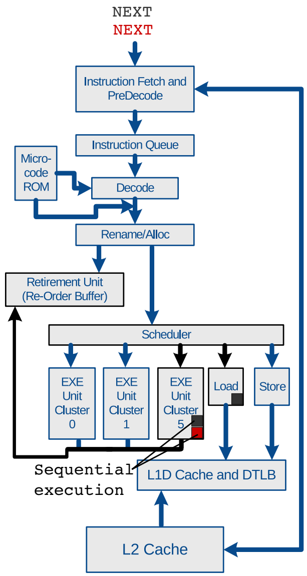

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
EXE Unit Cluster 0
EXE Unit Cluster 1
EXE Unit Cluster 5
Load
Store
L1D Cache and DTLB
L2 Cache

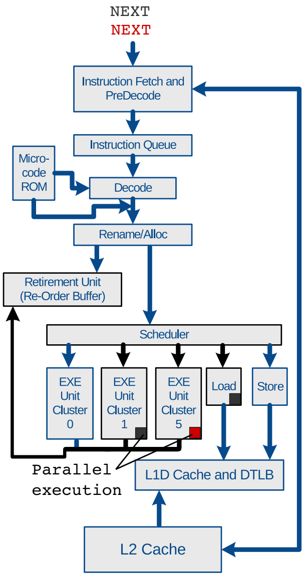
```
pshufb xmm3, [mem128]
pxor xmm1, xmm2
```



■■ 2 μops
🟥 1 μop


NEXT
NEXT
Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
■■ 2 μops
■ 1 μop
Scheduler
EXE Unit Cluster 0
EXE Unit Cluster 1
EXE Unit Cluster 5
Load
Store
L1D Cache and DTLB
L2 Cache


NEXT
NEXT
Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
EXE Unit Cluster 0
EXE Unit Cluster 1
EXE Unit Cluster 5
Load
Store
Sequential execution
L1D Cache and DTLB
L2 Cache


NEXT
NEXT
Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
EXE Unit Cluster 0
EXE Unit Cluster 1
EXE Unit Cluster 5
Load
Store
Parallel execution
L1D Cache and DTLB
L2 Cache

Throughput and μops

- Each Port (execution unit entry) has a 1 cycle latency
- The latency of an instruction represents the sum of the latencies of its sequential μops
  - `pxor xmm1, [mem128]` has to wait the resulting data from the memory load before doing the `xor` operation

Throughput and μops

- Each Port (execution unit entry) has a 1 cycle latency
- The latency of an instruction represents the sum of the latencies of its sequential μops
  - `pxor xmm1, [mem128]` has to wait the resulting data from the memory load before doing the `xor` operation
- The throughput of an instruction is directly related to its independent μops decomposition, as well as to its port binding
  - `pxor` is composed of a unique μop, and can be dispatched on Port0, 1 or 5
  - latency of the μop is 1 cycle $\Rightarrow$ throughput is $1/3 = 0.33$ cycle

AES-NI: μop analysis for Westmere

- IACA tool (Intel Architecture Code Analyzer):

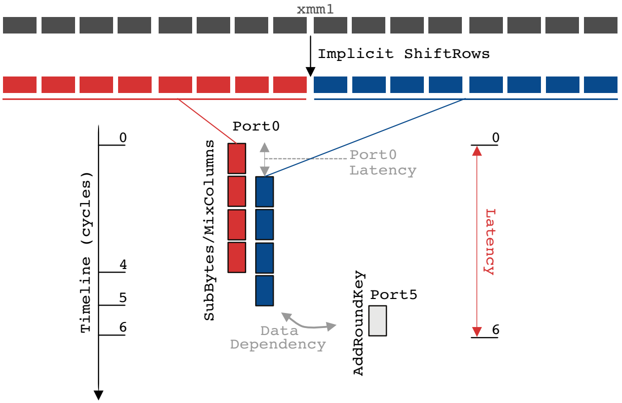
```
Throughput Analysis Report
--------------------------
Block Throughput: 6.00 Cycles       Throughput Bottleneck: InterIteration
| Num Of |              Ports pressure in cycles               |    |
|  Uops  |  0 - DV   |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
-----------------------------------------------------------------------
|   3    | 2.0       |     |           |           |     | 1.0 | CP | aesenc xmm0, xmm1
```

```
Throughput Analysis Report
--------------------------
Block Throughput: 6.00 Cycles       Throughput Bottleneck: InterIteration
| Num Of |              Ports pressure in cycles               |    |
|  Uops  |  0 - DV   |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
-----------------------------------------------------------------------
|   3    | 2.0       |     |           |           |     | 1.0 | CP | aesdec xmm0, xmm1
```

- 3 μops: two on Port0 and one on Port5

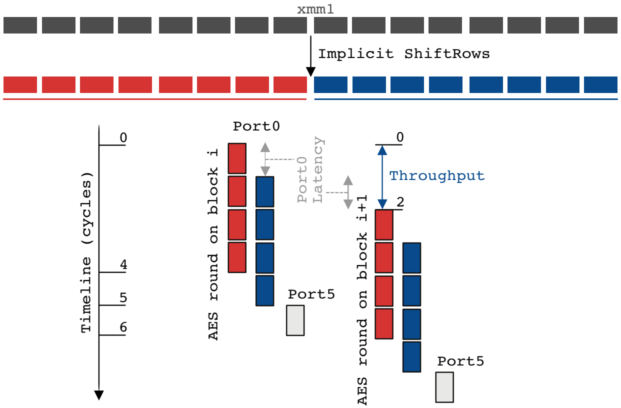
AES-NI: μop analysis for Westmere

- Decomposition and latency of `aesenc`:



AES-NI: μop analysis for Westmere

- Throughput of `aesenc`:



AES-NI: μop analysis for Sandy Bridge

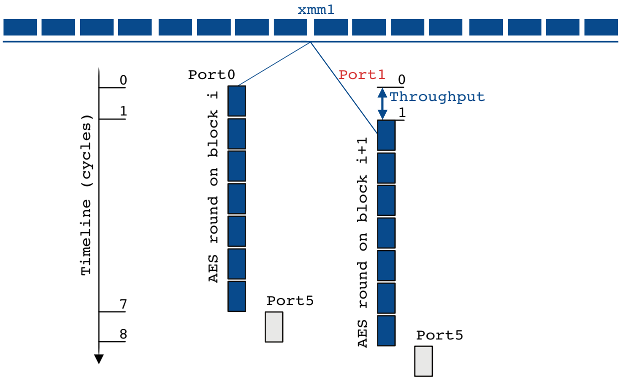
- IACA tool:

```
Throughput Analysis Report
__________________________

Block Throughput: 7.00 Cycles       Throughput Bottleneck: InterIteration
| Num Of |              Ports pressure in cycles              |    |
|  Uops  |  0 - DV   |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
______________________________________________________________________

|   2    | 0.5       | 0.5 |           |           |     | 1.0 | CP | aesenc xmm0, xmm1
```

- Latency is actually 8 cycles (Intel's Optimization Manual)
- An AES execution subunit has been added behind Port1
- The two μops operating on half states seem to have fused in one 7 cycles latency μop that can be dispatched on Port0 or Port1: pressure on Port0 is decreased

AES-NI: μop analysis for Sandy Bridge

- The throughput is reduced to 1 cycle



Latencies and throughputs summary

Westmere

Instruction	Latency	Throughput
aesenc	6	2
aesdec	6	2
aesenclast	6	2
aesdeclast	6	2
aeskeygenassist	6	2
aesimc	6	2
pxor	1	0.33

Instructions latencies and reciprocal throughputs (in cycles)

Sandy and Ivy Bridge

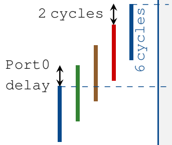
Instruction	Latency	Throughput
aesenc	8	1
aesdec	8	1
aesenclast	8	1
aesdeclast	8	1
aeskeygenassist	8[1]	8[1]
aesimc	2	2
pxor	1	0.33

Instructions latencies and reciprocal throughputs (in cycles)
[1] Only reported by Agner Fog's experimental results

- Reported by Intel documentation and confirmed experimentally (Agner Fog)

Exploiting instruction-level parallelism

- AES optimal parallel encryption for Westmere:



AES128 Parallel Encryption (4 blocks in parallel)

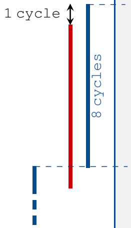
```
xmm4—xmm15                   ← scheduled keys
LOOP:
xmm0, xmm1, xmm2, xmm3       ← 4 plaintext blocks
pxor    xmm0, xmm4           /* Block0 — whitening */
pxor    xmm1, xmm4           /* Block1 — whitening */
pxor    xmm2, xmm4           /* Block2 — whitening */
pxor    xmm3, xmm4           /* Block3 — whitening */
aesenc  xmm0, xmm5           /* Block0 — Round 1   */
aesenc  xmm1, xmm5           /* Block1 — Round 1   */
aesenc  xmm2, xmm5           /* Block2 — Round 1   */
aesenc  xmm3, xmm5           /* Block3 — Round 1   */
aesenc  xmm0, xmm6           /* Block0 — Round 2   */
aesenc  xmm1, xmm6           /* Block1 — Round 2   */
aesenc  xmm2, xmm6           /* Block2 — Round 2   */
aesenc  xmm3, xmm6           /* Block3 — Round 2   */
...
aesenclast xmm0, xmm15       /* Block0 — Round 10  */
aesenclast xmm1, xmm15       /* Block1 — Round 10  */
aesenclast xmm2, xmm15       /* Block2 — Round 10  */
aesenclast xmm3, xmm15       /* Block3 — Round 10  */
CHAIN
jmp LOOP
```

Exploiting instruction-level parallelism

- Parallel encryption for Sandy and Ivy Bridge:

AES128 Parallel Encryption (8 blocks in parallel)

```
xmm8–xmm15, [mem]–[mem+3*16]     ← scheduled keys
LOOP:
xmm0, xmm1, xmm2, xmm3, xmm4,\
xmm5, xmm6, xmm7  ← 8 plaintext blocks
pxor   xmm0, xmm8         /* Block0 — whitening */
pxor   xmm1, xmm8         /* Block1 — whitening */
pxor   xmm2, xmm8         /* Block2 — whitening */
pxor   xmm3, xmm8         /* Block3 — whitening */
pxor   xmm4, xmm8         /* Block4 — whitening */
pxor   xmm5, xmm8         /* Block5 — whitening */
pxor   xmm6, xmm8         /* Block6 — whitening */
pxor   xmm7, xmm8         /* Block7 — whitening */
aesenc xmm0, xmm9         /* Block0 — Round 1   */
aesenc xmm1, xmm9         /* Block1 — Round 1   */
aesenc xmm2, xmm9         /* Block2 — Round 1   */
aesenc xmm3, xmm9         /* Block3 — Round 1   */
aesenc xmm4, xmm9         /* Block4 — Round 1   */
aesenc xmm5, xmm9         /* Block5 — Round 1   */
aesenc xmm6, xmm9         /* Block6 — Round 1   */
aesenc xmm7, xmm9         /* Block7 — Round 1   */
aesenc xmm0, xmm10        /* Block0 — Round 2   */
...
```



Theoretical performance (Westmere)

Mode	Formula	AES128	AES192[2]	AES256[2]
ECB Enc/Dec	$(N_{rounds} \times 2 + 0.33)/16 =$	1.27 c/B	1.52 c/B	1.77 c/B
CBC Encrypt[1]	$(N_{rounds} \times 6 + 0.33)/16 =$	3.77 c/B	4.52 c/B	5.27 c/B
CBC Decrypt[1]	$(N_{rounds} \times 2 + 0.33)/16 =$	1.27 c/B	1.52 c/B	1.77 c/B
CTR Enc/Dec[1]	$(N_{rounds} \times 2 + 0.33)/16 =$	1.27 c/B	1.52 c/B	1.77 c/B

Theoretical performance for 4-parallel blocks encryption and decryption (in cycles per byte) for ECB and CBC modes (Westmere)

[1] Plus a small overhead for chaining operations

[2] Plus a small overhead because of register starvation

- PCBC and CFB modes are like CBC (non parallel encryption and possible parallel decryption)
- OFB mode can't be parallelized (but precomputed for a given key)

Theoretical performance (Sandy/Ivy Bridge)

Mode	Formula	AES128[2]	AES192[2]	AES256[2]
ECB Enc/Dec	$(N_{rounds} \times 1 + 0.33)/16 =$	0.64 c/B	0.77 c/B	0.89 c/B
CBC Encrypt[1]	$(N_{rounds} \times 8 + 0.33)/16 =$	5.02 c/B	6.02 c/B	7.02 c/B
CBC Decrypt[1]	$(N_{rounds} \times 1 + 0.33)/16 =$	0.64 c/B	0.77 c/B	0.89 c/B
CTR Enc/Dec[1]	$(N_{rounds} \times 1 + 0.33)/16 =$	0.64 c/B	0.77 c/B	0.89 c/B

Theoretical performance for 8-parallel blocks encryption and decryption (in cycles per byte) for ECB and CBC modes (Sandy and Ivy Bridge)

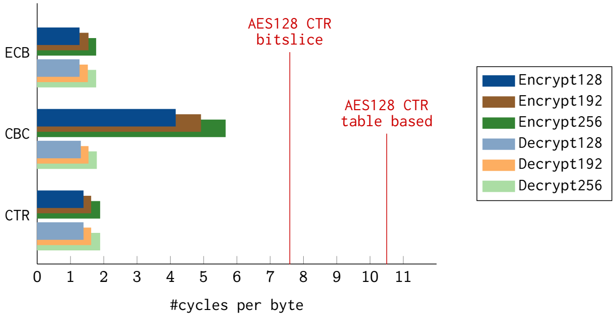
[1] Plus a small overhead for chaining operations

[2] Plus a small overhead because of register starvation

Practical results

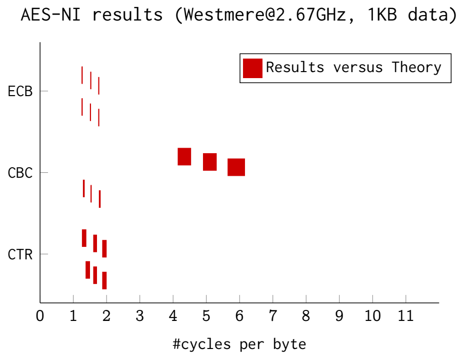
- Official results in Intel's White Paper on Westmere, 4 parallel blocks:

AES-NI results (Westmere@2.67GHz, 1KB data)



Practical results

- Differences with theory:



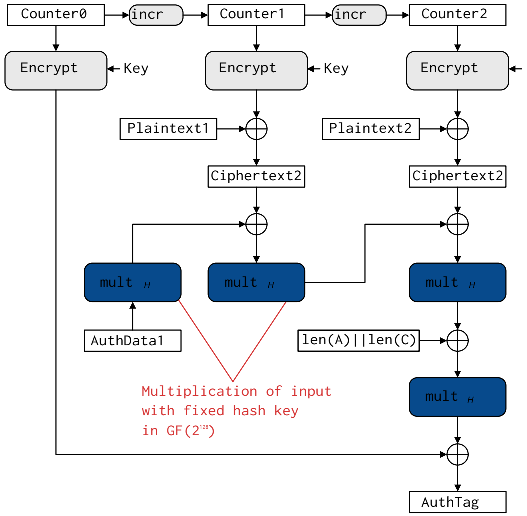
What about the Key Schedule?

- Intel has developped AES-NI with encryption and decryption using the same key in mind
- Key Schedule becomes negligible when encrypting multiple blocks with the same key
- This explains why `aeskeygenassist` performs quite poorly on Ivy/Sandy bridge
- AES-NI provides however better performance - with constant time implementation - than table based Key Schedule: $\sim$100 cycles against $\sim$160 cycles

VEX encoded AES (AVX extensions)

- There are VEX extensions of AES-NI instructions: `vaesenc, vaesdec ...`
- However, the instructions only work on the low part `xmm` of `ymm` registers
- The advantage of using three operands versions of the instructions remains:
  - the Key Schedule can benefit from the extended instructions ...
  - ... at the cost of using VEX only instructions (to avoid VEX/SSE switch latencies)

GCM mode


Counter0
incr
Counter1
incr
Counter2
Encrypt
Key
Encrypt
Key
Encrypt
Plaintext1
Plaintext2
Ciphertext2
Ciphertext2
mult H
mult H
mult H
AuthData1
len(A)||len(C)
mult H
Multiplication of input with fixed hash key in GF(2^128)
AuthTag

pclmulqdq instruction

- Not an AES-NI instruction per se
- Performs a “carry-less multiplication” (polynomial multiplication over GF(2))

pclmulqdq xmm1, xmm2/[mem128], imm8

```
/* Split in two quadwords */
xmm1  := [xmm1[1]|xmm1[0]]
xmm2  := [xmm2[1]|xmm2[0]]
if(imm8 == 0x00)
  xmm1  ← xmm2[0] × xmm1[0]
if(imm8 == 0x01)
  xmm1  ← xmm2[0] × xmm1[1]
if(imm8 == 0x10)
  xmm1  ← xmm2[1] × xmm1[0]
if(imm8 == 0x11)
  xmm1  ← xmm2[1] × xmm1[1]
```

Using pclmulqdq

- GCM multiplies two 128-bit values over GF(2^{128})
- Two issues:
  - Carry-less multiplication of two 128-bit operands to give a 255-bit value
    $\Rightarrow$ use schoolbook or Karatsuba algorithms
  - Reduction of the resulting value over GF(2^{128}) with the GCM irreductible polynomial ($x^{128} + x^7 + x^2 + x + 1$)
    $\Rightarrow$ Intel's manual gives many optimized reduction algorithms

Using pclmulqdq

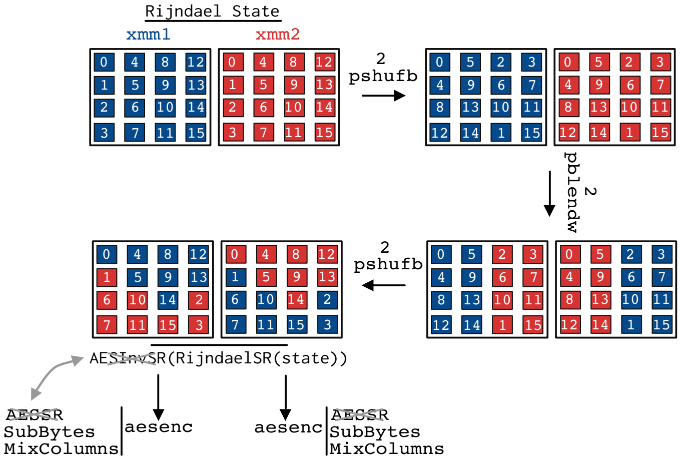
- GCM multiplies two 128-bit values over GF(2^{128})
- Two issues:
  - Carry-less multiplication of two 128-bit operands to give a 255-bit value
    $\Rightarrow$ use schoolbook or Karatsuba algorithms
  - Reduction of the resulting value over GF(2^{128}) with the GCM irreductible polynomial ($x^{128} + x^7 + x^2 + x + 1$)
    $\Rightarrow$ Intel's manual gives many optimized reduction algorithms
- Result on Westmere: AES GCM performs at 3.54 c/B with 4 parallel blocks CTR encryption, to compare with 10.68 c/B bitsliced AES GCM with table lookups (21.99 c/B without table lookups, Käsper et al.)

Rijndael

- Rijndael uses the same building blocks as AES, with a state up to 256-bit and extended possible key lengths
- AES-NI Key Schedule instructions fit the Rijndael Key Schedule
- The main issue comes from the ShiftRows on states of length > 128-bit that don't fit the AES

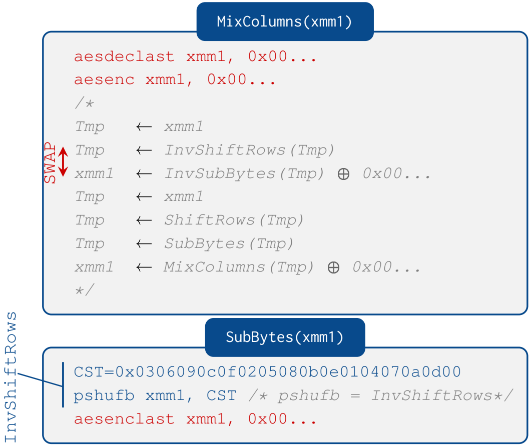
Rijndael (256-bit state example)

- **Solution:** prepare the state (xmm1, xmm2) with AESShiftRows^{-1}(RijndaelShiftRows(state))



Building blocks isolation

- We can apply the same function composition strategy to isolate `ShiftRows`, `MixColumns`, `SubBytes`, `RotByte`



MixColumns(xmm1)

```
aesdeclast xmm1, 0x00...
aesenc xmm1, 0x00...
/*
Tmp   ← xmm1
Tmp   ← InvShiftRows(Tmp)
xmm1  ← InvSubBytes(Tmp) ⊕ 0x00...
Tmp   ← xmm1
Tmp   ← ShiftRows(Tmp)
Tmp   ← SubBytes(Tmp)
xmm1  ← MixColumns(Tmp) ⊕ 0x00...
*/
```

SubBytes(xmm1)

```
CST=0x0306090c0f0205080b0e0104070a0d00
pshufb xmm1, CST /* pshufb = InvShiftRows*/
aesenclast xmm1, 0x00...
```

Building blocks isolation

- The same building block can have multiple decompositions:

```
InvMixColumns = {aesimc}
              = {aesenclast + aesdec}
```

Building blocks isolation

- The same building block can have multiple decompositions:
  ```
  InvMixColumns = {aesimc}
                = {aesenclast + aesdec}
  ```

- One must check the resulting latency and throughput, and use the optimal decomposition (or combine decompositions)
  - using aesenclast and pshufb to compose SubBytes seems clearly more efficient than composing aesenc, aesimc and pshufb
  - depends on the microarchitectural details

Building blocks isolation

- In order to achieve maximum throughput, composed building blocks must be parallelized atomically for each instruction (remove critical paths)

Maximum throughput MixColumns (Westmere)

```
aesdeclast xmm0, 0x00...
aesdeclast xmm1, 0x00...
aesdeclast xmm2, 0x00...
aesdeclast xmm3, 0x00...
aesenc xmm0, 0x00...
aesenc xmm1, 0x00...
aesenc xmm2, 0x00...
aesenc xmm3, 0x00...
```

MixColumns whith critical paths

```
aesdeclast xmm0, 0x00...
aesenc xmm0, 0x00...
aesdeclast xmm1, 0x00...
aesenc xmm1, 0x00...
aesdeclast xmm2, 0x00...
aesenc xmm2, 0x00...
aesdeclast xmm3, 0x00...
aesenc xmm3, 0x00...
```

Building blocks isolation

- Parts of the building blocks can also be isolated
- `MixColumns` sub-matrix multiplication isolation:

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \times \begin{pmatrix} x_0 \\ 0 \\ x_1 \\ 0 \end{pmatrix}$$

Hash functions

- The versatility of AES-NI instructions allows them to be used in other areas than AES or Rijndael:
  - All cryptographic algorithms that use AES building blocks can benefit from AES-NI ...
  - ... with performance benefits and/or constant time implementation
- More specifically, many candidates of the recent SHA-3 competition have used AES-NI to improve performance or provide resisance against side channel attacks

Hash functions

- Some SHA-3 candidates results:

	Algorithm	Previous 256/512	AES-NI 256/512
Final Round	Grøstl	19.9 / 29.2	11.3 / 16.2
Round 2	ECHO	28.5 / 53.5	6.8 / 12.6
	Shavite-3	26.7 / 38.2	5.6 / 5.5
Round 1	Cheetah	9.3 / 13.1	7.6 / -
	Lane	25.7 / 56.5	4.9 / 13.5
	Lesamnta	52.7 / 51.2	29.5 / 19.0
	LUX	10.5 / 9.26	6.6 / -
	Vortex	46.3 / 56.1	4.4 / 5.2

AES-NI performance benefits on Westmere
(results in cycles per byte)

Concluding thoughts

- Since Intel's White Paper in 2008, AES-NI has become a reality with Westmere and Sandy/Ivy Bridge
- Adding AES in the ISA rather than in a dedicated coprocessor has advantages (software compliance across platforms)
  - ARM and SPARC plan to add similar instructions in their next generation CPUs
- What could be the future of AES-NI?
  - AVX2 (in the forthcoming Haswell microarchitecture) don't include 256-bit AES ymm support: it might be planned for future release (?)
  - the latency (8 cycles) can be improved, and μop decomposition reduced