CS433 Modern Architectures

Video 5

Modern Instructions sets – random numbers

This video is the copyright of Maynooth University and may not be copied, or reposted.

Created for streaming using Panopto within MU Moodle only.

"Advanced Architectures"

So far we have studied the following integer and floating point instructions available on x64 processors.

x86, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX

Apart from the basic machine code instructions these are all SIMD instructions.

We can now look at the following instructions and this will bring us up to date with the Assembly Language.

RDRand/RDSeed: - Instructions to digitally generate random numbers (uses some of the AESNI technology)

TSC: - Time stamp counter

AESNI: - Advanced encryption standard

Random Number Generators - RNG

An RNG is a utility or device of some type that produces a sequence of numbers on an interval [min, max] such that values appear unpredictable.

- Each new value must be statistically independent of the previous value.

- The overall distribution of numbers chosen from the interval is uniformly distributed.

- The sequence is unpredictable.

Coin toss - RNG

Applications of RNG's
- gambling,
- statistical sampling,
- computer simulation,
- cryptography.

https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

TRNG - True Random Number Generators
extract randomness from a physical system

Dice

Irish Sweepstakes

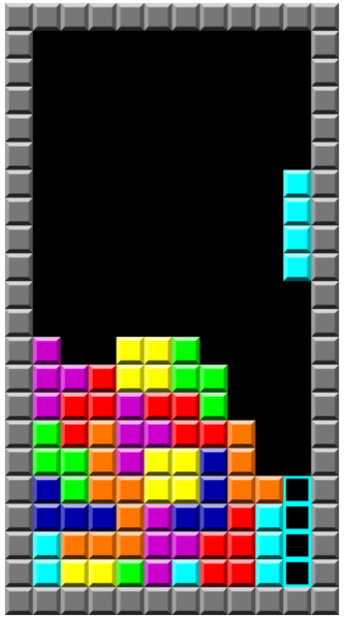
Telly Bingo

Types of random number generator

Encryption

PRNG - Pseudo-Random Number Generators
use an algorithm

Digital gambling



Computer games

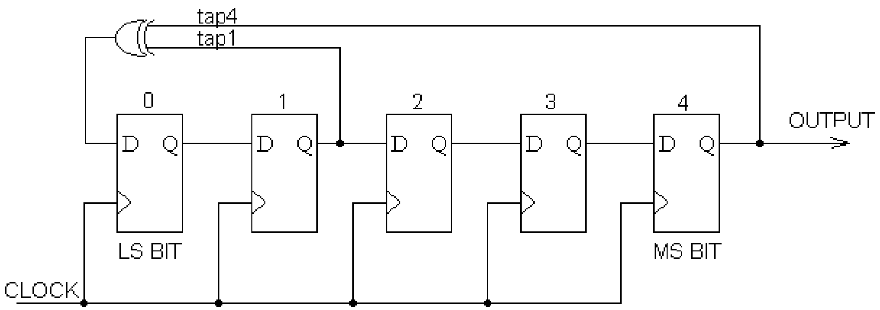
CAO

CAO Applicants on same points, selected at random

Pseudo-Random Number Generators (PRNGs)

They are deterministic – given the same starting conditions (seed) they will produce the same sequence.

This could be useful if you want to repeat a sequence each time you run….but it makes it predictable….



Shift register with two taps feeding into an exor gate.

$6254^2 = 39112516$
$1125^2 = 1265625$
$6562^2 = 43059844$
$0598^2 = 357604$

1125656205985760……

Middle square method
John von Neumann 1946

Encrypt
- M= 1011, 0110 …(message)
- K = Key 1010
- $C_{1\text{-}4} = K \otimes M_{1\text{-}4} = 0001$
- $C_{5\text{-}8} = C_{1\text{-}4} \otimes M_{5\text{-}8} = 0111$
- C=0001,0111 …(Cipher)

Decrypt
- $M_{1\text{-}4} = K \otimes C_{1\text{-}4} = 1011$
- $M_{5\text{-}8} = C_{1\text{-}4} \otimes C_{5\text{-}8} = 0110$

Block Cipher DES/AES

https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

PRNG – Linear congruential generators

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

unsigned int seed = 128;

void rand_lcg()
{
 unsigned int m = 256, a = (4 * 4321) + 1, c = 5423;
 seed = (a * seed + c) % m;
}

int main()
{
// Print values
    FILE* fp;
fopen_s(&fp,"C:\\x64\\list.txt", "wt");
for (int i = 0; i < 15; i++)
{
      //seed = rand() % 256;
  fprintf(fp,"%u\n", seed);
  rand_lcg();  // LCG-8
}
fclose(fp);
```

$$x_{n+1} = (ax_n + c) mod\ m$$

a=(4*4321)+1

c=5423

m=256 (should be 2^16=65536)

Rules for selecting a, c, m

The Hull-Dobell theorem

1 is the only number that divides evenly into both m and c (relatively prime up to m)

If q is prime number that divides m, then q divides a-1

If 4 divides m then 4 divides a-1

How good is the random number generator?

Look at the numbers

LCG-8	C\C++
128	41
175	35
26	190
177	132
36	225
227	108
30	214
197	174
136	82
215	144
226	73
153	241
172	241
139	187
102	233

Look at the statistics on 5,000 trials

C\C++
Mean:128.159
Standard deviation:74.270
Standard error:1.050

LCG-8
Mean: 127.623
Standard deviation: :73.882
Standard error:1.045

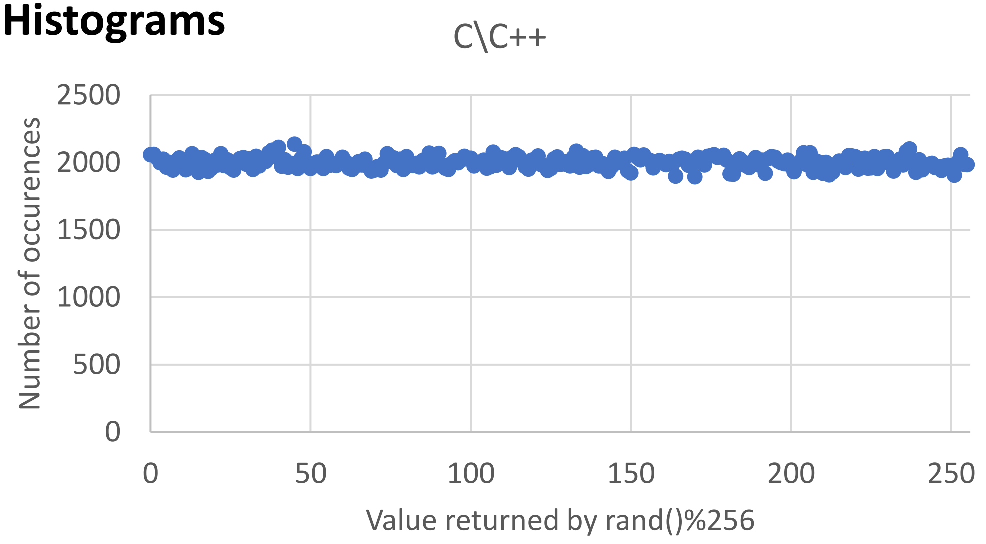
Looking for sequence 1, 2, 3

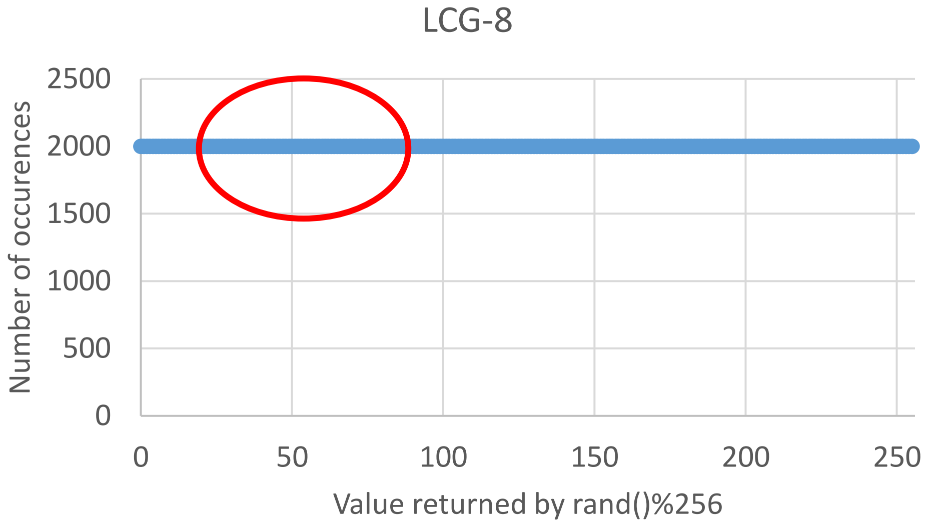
C\C++
Occurs at: 916978, 17694194…
LCG-8
Occurs at: Never (>10M tries)

Looking for repetition

C\C++
Repeats at 16777216, 33554432, 50331648
LCG-8
Repeats at 256, 512,768

Histograms

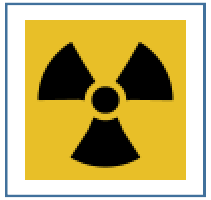




Some tests of randomness

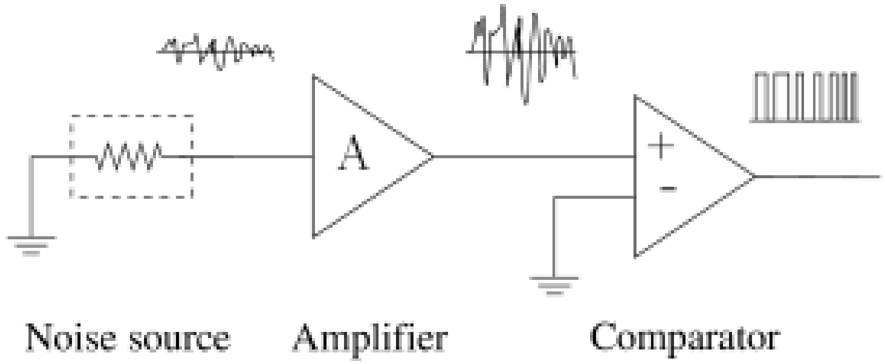
True-Random Number Generators (TRNGs)

An entropy source ES makes use of randomness in a physical system to create a random number.



1: Time between two disintegrations is random

2: The number of disintegrations has a Poisson distribution or Gaussian distribution.



Amplify a noise source and threshold using comparator.

Software whitening

Sometimes the bit streams can have a 1 or 0 bias. To over come this John von Neumann proposed this bit whitening approach. The binary bit stream is converted to a new stream. 0 to 0 and 1 to 1 are excluded, a 0 to 1 transition becomes a 0 bit a 1 to 0 transition becomes 1.

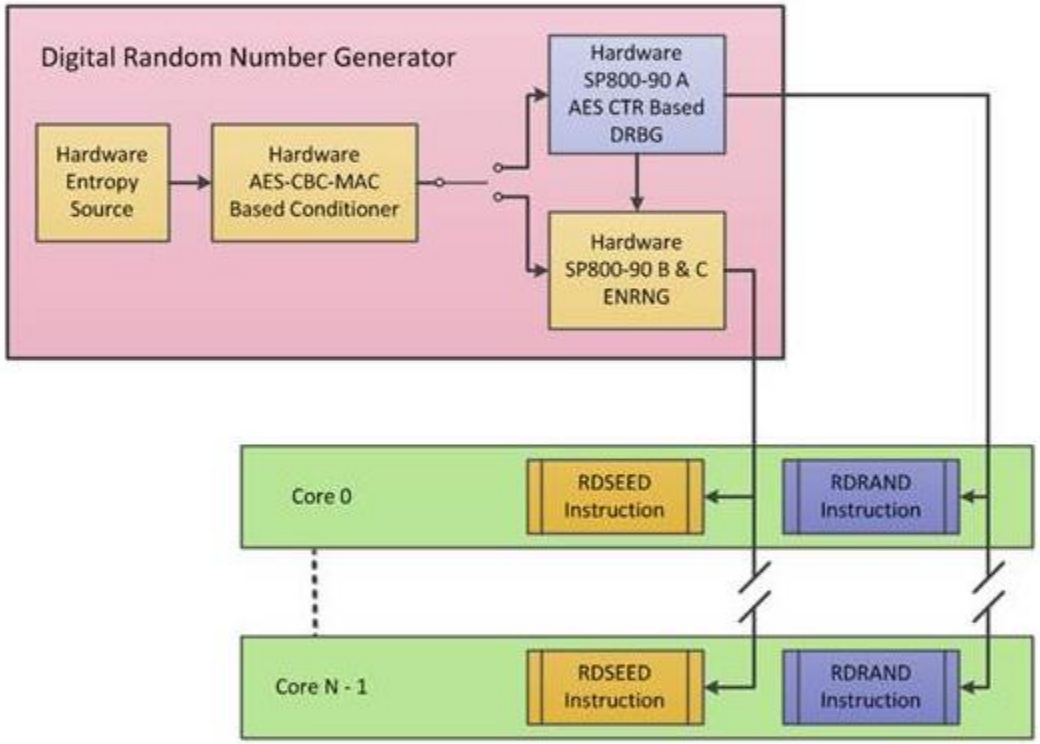
11-00-10-01-11-01 gives output XX-XX-1-0-XX-0 or 1 0 0

Reduces data rate (bad) and bit bias (good).

https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

X64 rdrand and rdseed

Intel x64 -

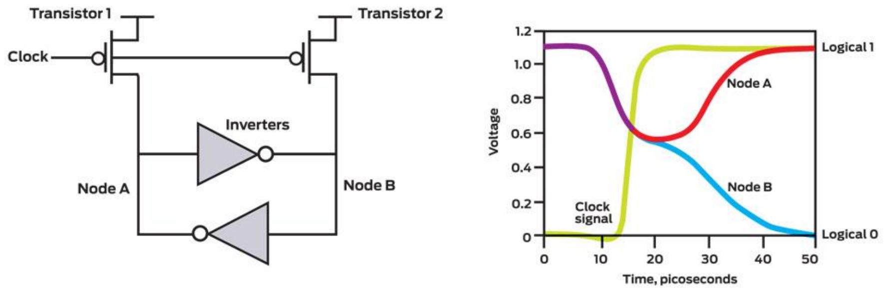


DRBG : deterministic random bit generator () seeded from the conditioner.

ENRNG enhanced, nondeterministic random number generator () that provides seeds from the entropy conditioner.

Entropy source

Johnson–Nyquist noise (thermal) noise in silicon sampled at 3 GHz



$$V = \sqrt{4RkT.\Delta f}$$

where,

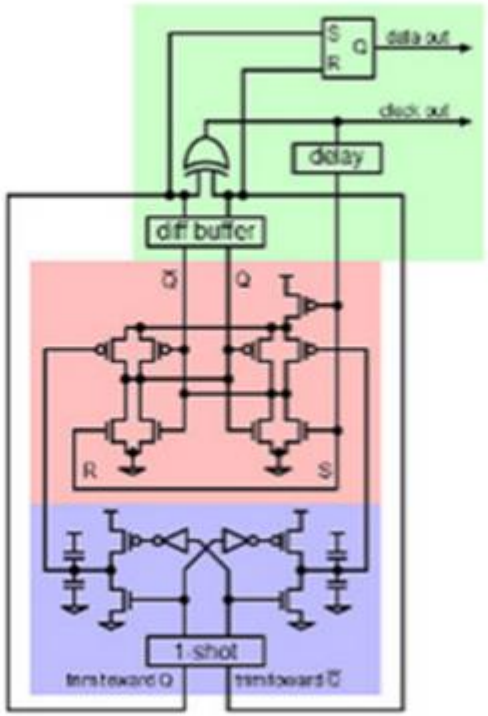
V is the root-mean-square noise voltage,
k is Boltzmann's constant (1.38 × 10−23 joules/kelvin),
T is the absolute temperature in kelvins,
Δf is the bandwidth in hertz

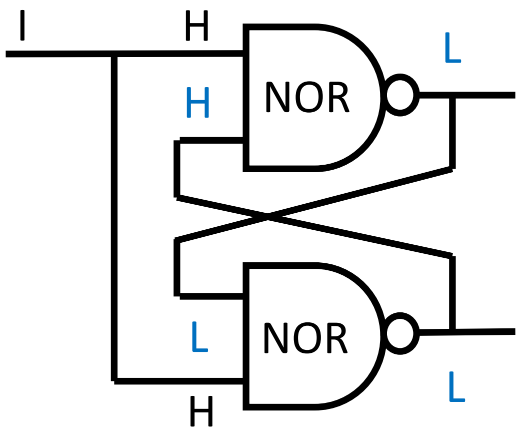
Entropy conditioner

https://spectrum.ieee.org/computing/hardware/behind-intels-new-randomnumber-generator

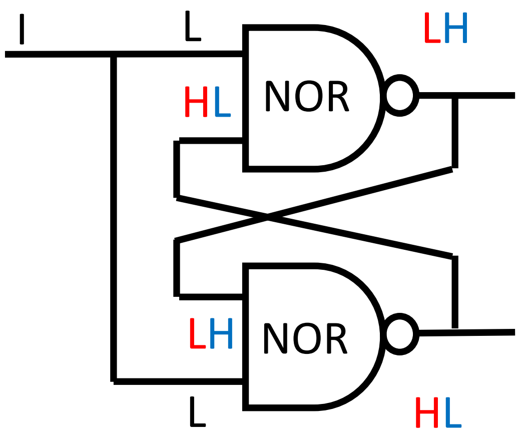
An interesting find – to explain operation of Intel random number generator

S	R	Q	State
0	0	Previous State	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Forbidden





I low – two outputs should be opposite,
All 4 possible states produce LL output



I low – output settles
to either H or L

Ivy Bridge's entropy source is an RS-NOR latch with the set and reset inputs wired together (red). When the R/S input is de-asserted, the latch becomes metastable, and its output eventually settles to 0 or 1, depending on thermal noise. The tricky part is consistently reaching that metastable state. This is accomplished by a negative feedback circuit (blue). This circuit adjusts the charge on a set of large capacitors, which is used as an extra input to the latch. The feedback nudges the latch slightly more toward 0 whenever it produces a 1 and vice-versa. The buffering circuit (green) detects when the latch has settled, and stores its output. After a delay it asserts and then de-asserts the latch's R/S input to produce another bit.

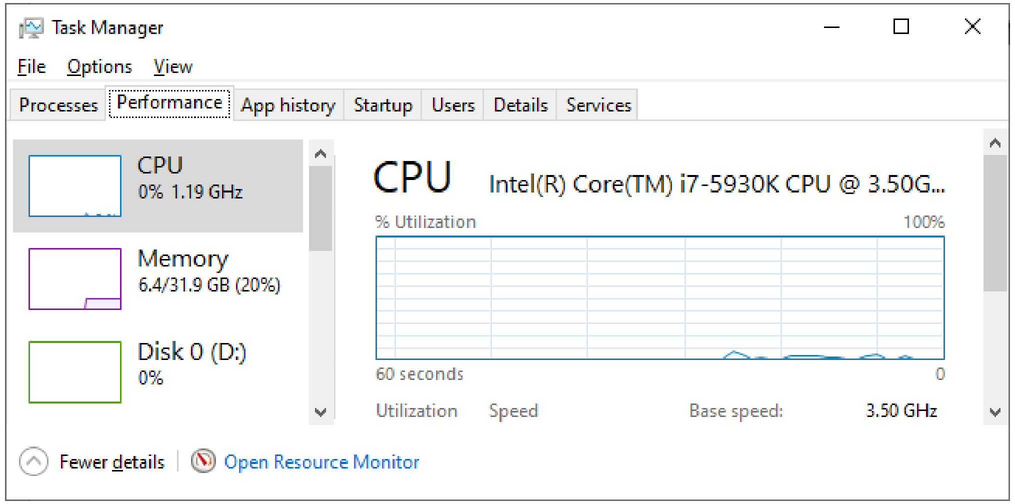
https://www.electronicdesign.com/resources/article/21796238/understanding-intels-ivy-bridge-random-number-generator

X64 Intel digital random number generator - rdrand and rdseed

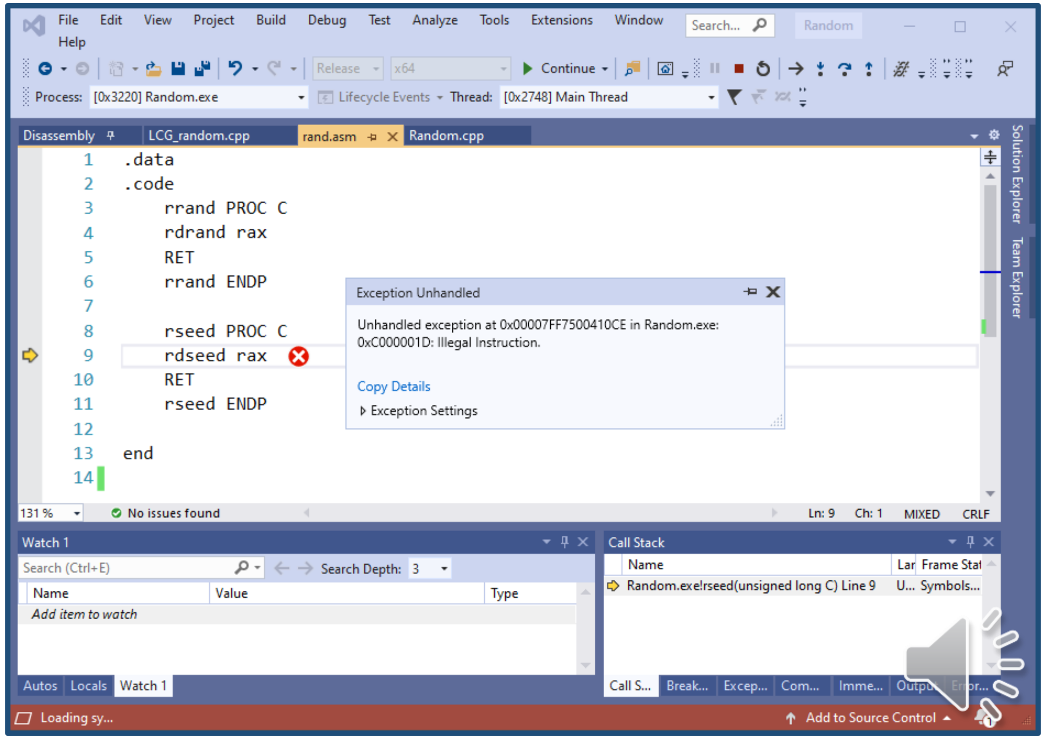
```
.data
.code
    rrand PROC C ; PRNG
    rdrand rax
    RET
    rrand ENDP

    rseed PROC C ; TRNG
    rdseed rax
    RET
    rseed ENDP
end
```

```
#include <stdio.h>

using namespace std;

extern "C"
{
 long long rrand(void);
 long long rseed(void);
}

int main(int argc, char* argv[])
{
 for (int i = 0; i < 10; i++)
 {
  printf("%0.16llX, %0.16llX\n", rrand(),rseed());
 }
 return 0;
}
```

How do you check instruction rdseed availability?



PC-Broadwell - Ivy Bridge



CPUID

https://en.wikipedia.org/wiki/CPUID

CPU - Identification allowing software to discover details of the processor functionality (instruction sets available)

CUPID is a serialization instruction.

Serialisation: an instruction that flushes the store buffer and the out-of-order instruction pipeline before any later instructions can execute. If a program runs each instruction before starting the next instruction it is said to serialised (pipeline is emptied).

```
printf("RDSSEED:%C\n",rseed(), (char)cpuid_check());
```

```
cpuid_check PROC C
    mov eax, 7             } Setup
    mov ecx, 0
    cupid                  } Run cupid instruction
    test ebx, 40000h       } and ebx, 40000 ->zf, sf, pf
jz  L1                     } Use data returned
    mov eax,'Y'
    jmp L2
L1: mov eax,'N'
L2: RET                    } Use data returned
    cpuid_check ENDP
```

CPU vendor = GenuineIntel
CPU Brand String = Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz
# of cores = 6
# of logical cores = 12
Is CPU Hyper threaded = 1
CPU SSE = 1
CPU SSE2 = 1
CPU SSE3 = 1
CPU SSE41 = 1
CPU SSE42 = 1
CPU AVX = 1
CPU AVX2 = 1