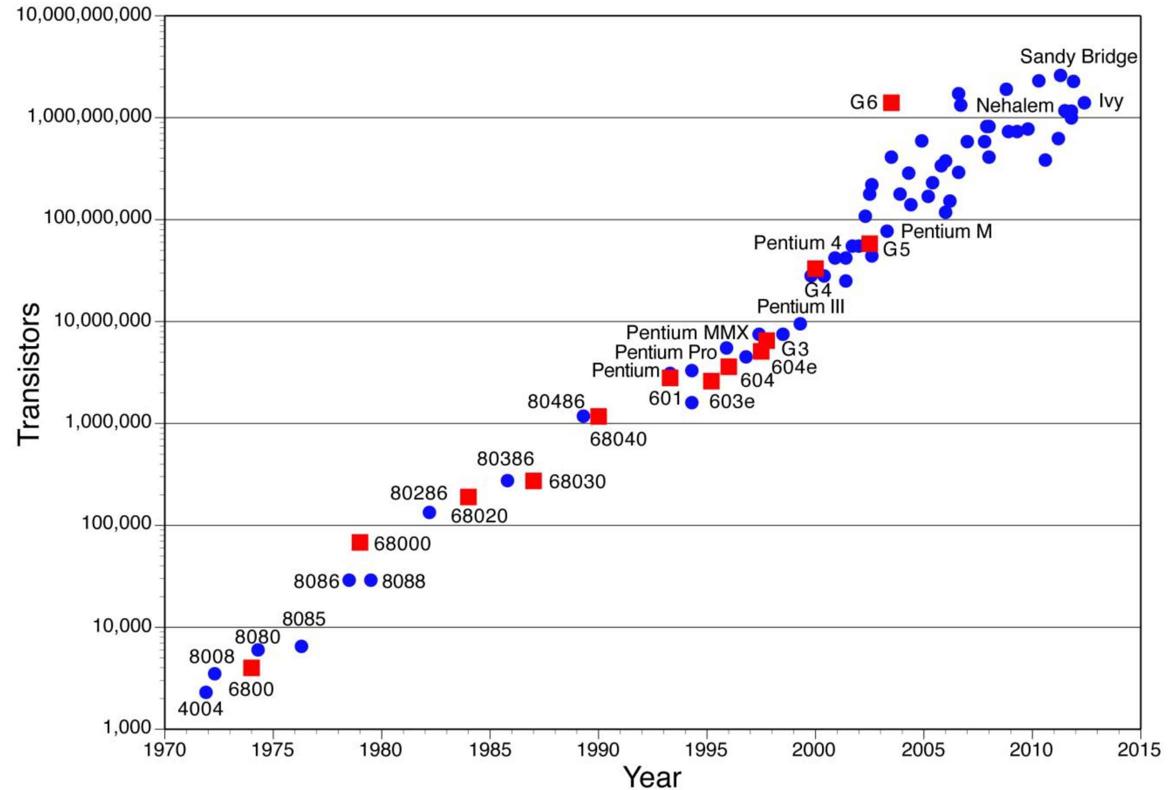
MISC CS TOPICS (IV)

Tony Wong

2016-02-20

Moore's Law

- Number of transistors double every two years



What makes a processor faster?

Pentium D 945 3.4GHz
Released July 2006
Release Price $163
Avg PassMark: 748
TDP: 95W

Pentium G4400 3.3 GHz
Released September 2015
Release Price $64
Avg PassMark: 3722
TDP: 54W

What makes a processor faster?

8 **CPUBoss Score**
Combination of all six facets

Pentium G4400	8.0
Pentium D 945	5.6
Pentium G3258	7.6

intel®

Winner

Intel Pentium D 945

CPUBoss recommends the Intel Pentium D 945 based on its value.

See full details

Buy now amazon.com HK$31

Intel Tick-Tock

- Tick: Die Shrink
  - Ivy Bridge (-3xxx) 32nm -> 22nm
  - Broadwell (-5xxx) 22nm -> 14nm
- Tock: New microarchitecture design
  - Sandy Bridge (-2xxx)
  - Haswell (-4xxx)
  - Skylake (-6xxx)

Intel ARK

Product Name	Intel® Pentium® D Processor 945 (4M Cache, 3.40 GHz, 800 MHz FSB)	Intel® Pentium® Processor G4400 (3M Cache, 3.30 GHz)
Code Name	Presler	Skylake
Essentials		
Status	End of Interactive Support	Launched
Launch Date	Q3'06	Q3'15
Processor Number	945	G4400
Cache	4 MB L2 Cache	3 MB Intel® Smart Cache
Bus Type	FSB	DMI3
System Bus	800 MHz	8 GT/s
FSB Parity	No	
Instruction Set	64-bit	64-bit
Embedded Options Available	No	Yes
Lithography	65 nm	14 nm
VID Voltage Range	1.200V-1.3375V	
Recommended Customer Price	TRAY: $163.00	BOX : $64.00 TRAY: $64.00
Datasheet	Link	Link
Instruction Set Extensions		SSE4.1/4.2
Scalability		1S Only
Thermal Solution Specification		PCG 2015C (65W)

Numbers

- Base / Max frequency
- Cores
  - Most Xeon server processors have > 10 cores
- Threads (hyper-threading)
- Cache
- Memory bandwidth
- Price

Better power management

- Especially important for mobile processors
- Multiple power states from sleeping to active
- “Turbo Boost”
  - Increase clock rate when required
- Lowers power consumption
  - CPU may start to throttle under high temperature
  - Stay at higher clock rate for longer time

New features

- Virtualization
  - VT-x and VT-d increase virtualization performance
- Security features
  - Disable execution bit
- Encryption instructions
  - AES-NI increase symmetric key encryption / decryption performance (e.g. web browsing)
  - Hardware random number generator

Improved architecture (what you can't see from ARK)

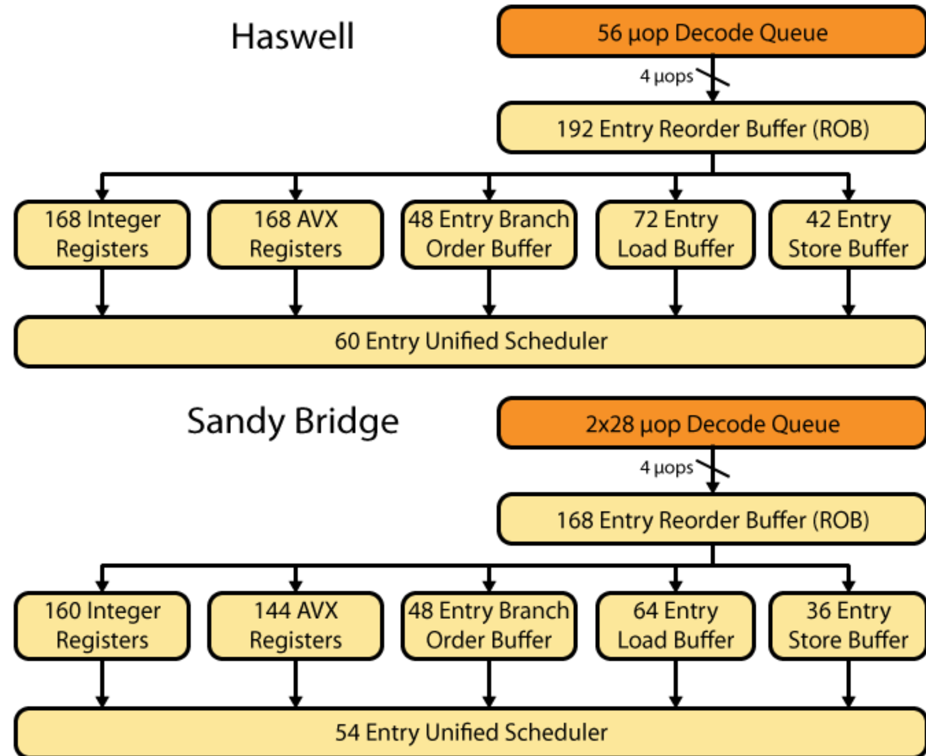
- Shorter instruction execution cycles
- Improved hardware optimizations
- Improved pipelines
- Improved branch prediction
- Improved cache and pre-fetch

Shorter instruction execution cycles

- Estimate the no. of cycles required to perform the following operations in a “Skylake” processor
  - add(int, int)   mul(int, int)   div(int, int)
  - add(double, double)   mul   div
  - sqrt(double)

Improved hardware architecture

- More registers
- Out of order execution
  - Execute instructions as data become available
- Loop detection



Improved hardware architecture

2.2 THE HASWELL MICROARCHITECTURE

The Haswell microarchitecture builds on the successes of the Sandy Bridge and Ivy Bridge microarchitectures. The basic pipeline functionality of the Haswell microarchitecture is depicted in Figure 2-2. In general, most of the features described in Section 2.2.1 - Section 2.2.4 also apply to the Broadwell microarchitecture. Enhancements of the Broadwell microarchitecture are summarized in Section 2.2.6.

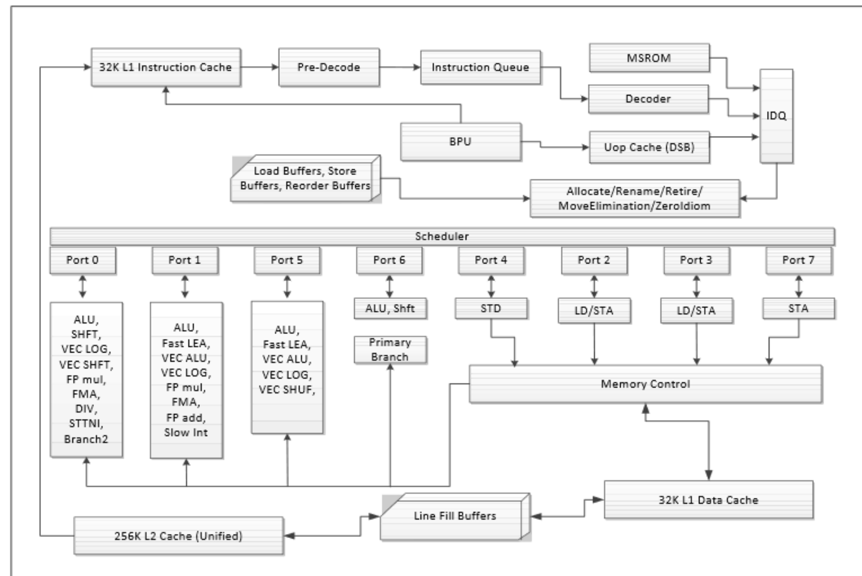


Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture

2.1 THE SKYLAKE MICROARCHITECTURE

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 2-1.

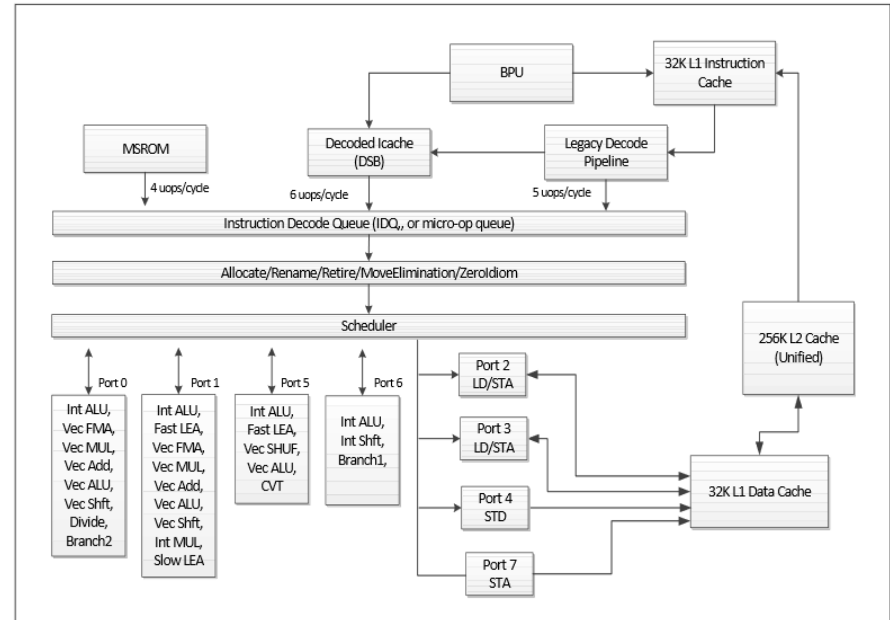
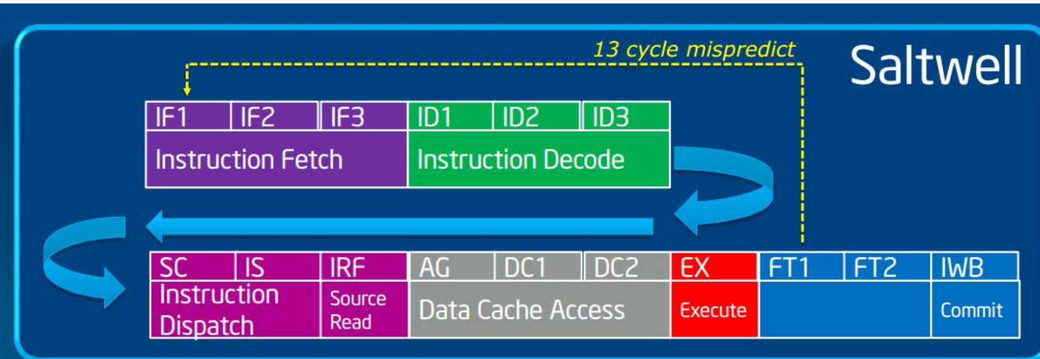
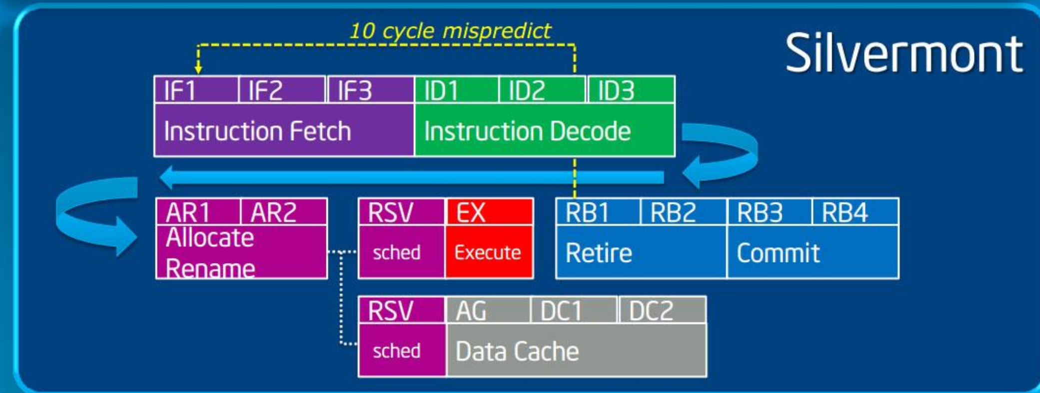


Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture

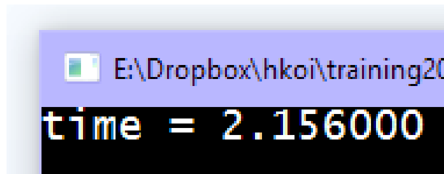
Improved Pipelines





Improved branch prediction

- Every array element is 123456789
- The branch on line 14 is always evaluated to false

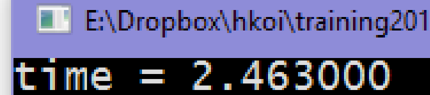



```
1  #include <cstdio>
2  #include <ctime>
3  int a[1000000];
4  int main() {
5    // data
6    for (int i = 0; i < 1000000; i++) {
7      a[i] = 123456789;
8    }
9    // run
10   clock_t t = clock();
11   long long sum = 0;
12   for (int i = 0; i < 1000; i++) {
13     for (int j = 0; j < 1000000; j++) {
14       if (a[j] >= 500000000) {
15         sum += a[j];
16       }
17     }
18   }
19   t = clock() - t;
20   printf("time = %f\n", (double)t / CLOCKS_PER_SEC);
21   return 0;
22 }
```

Improved branch prediction

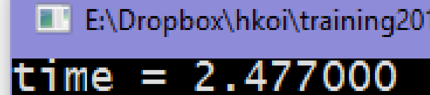
- Alternating true / false

```
// data
for (int i = 0; i < 1000000; i++) {
    a[i] = 500000000 * (i % 2) + i;
}
```



- 2 false 1 true (cycle length = 3)

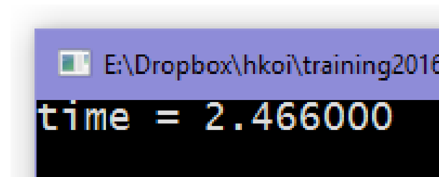
```
// data
for (int i = 0; i < 1000000; i++) {
    a[i] = 300000000 * (i % 3) + i;
}
```



Improved branch prediction

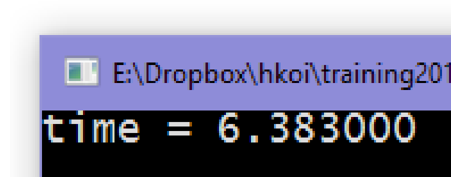
- Pattern (cycle length = 16)

```
int b[16] = {1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0};
int main() {
  // data
  for (int i = 0; i < 1000000; i++) {
    a[i] = 500000000 * b[i % 16] + i;
  }
```



- Random data

```
// data
a[0] = 1;
for (int i = 1; i < 1000000; i++) {
  a[i] = 1LL * a[i - 1] * 137 % 1000000007;
}
```



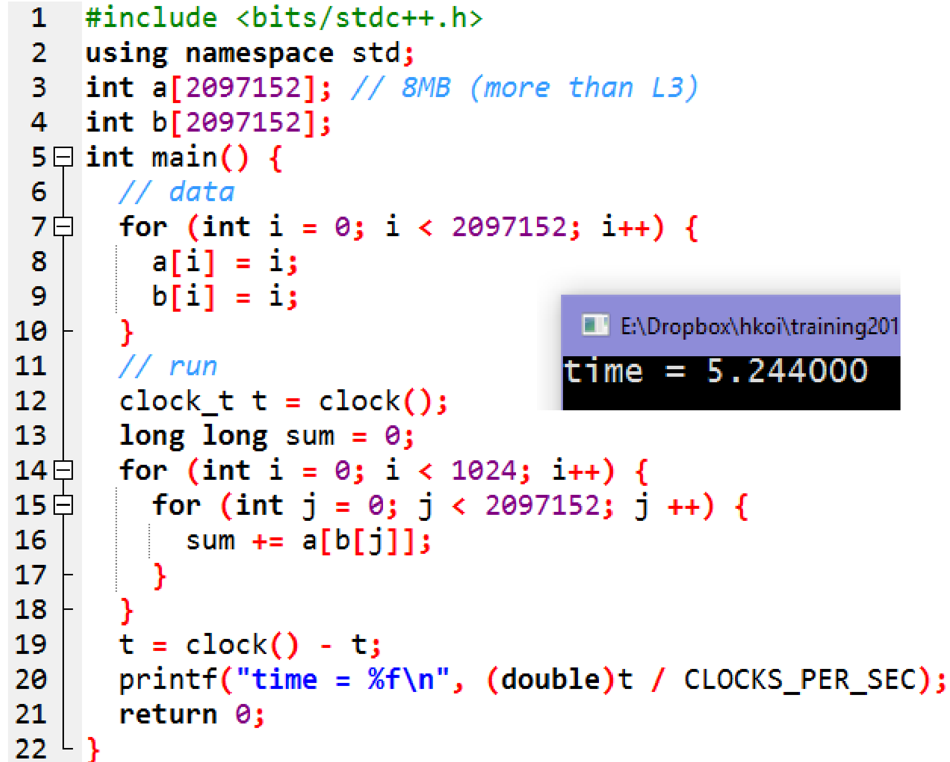
Improved cache and pre-fetch

- Linear array scan

Table 2-4. Cache Parameters of the Skylake Micr

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)
Instruction	32 KB/8	64	N/A	N/A
Second Level	256KB/4	64	12 cycle	64
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32

RAM latency: ~10 ns



Improved cache and pre-fetch

```
if (run == 0) b[j] = j;
if (run == 1) b[j] = 2097151 - j;    // reverse
if (run == 2) b[j] = j / 4 * 4;      // blocks of 16 bytes
if (run == 3) b[j] = j / 16 * 16;    // 64 byte lines
if (run == 4) b[j] = j / 16 * 16 + (15 - j % 16); // block reverse
if (run == 5) b[j] = j / 32 * 32 + (16 - j / 16 % 2 * 16); // 1 0 3 2 5 4 ...
if (run == 6) // random 64 byte lines
  b[j] = (j == 0 ? 16 : (j % 16 == 0 ? b[j - 1] / 16 * 91LL % 131072 * 16 : b[j - 1]));
if (run == 7) // random 64 byte lines floo
  b[j] = (j == 0 ? 16 : (j % 16 == 0 ? b[j - 1] / 16 * 91LL % 131072 * 16 : b[j - 1] + 1));
if (run == 8) b[j] = (j == 0 ? 1 : b[j - 1] * 91LL % 2097152); // random
```

- The processor can predict what data will be required next and pre-fetch them from RAM into processor cache
- Ivy Bridge cache line size = 64 bytes

```
run 0: time = 5.516000
run 1: time = 5.189000
run 2: time = 5.334000
run 3: time = 5.191000
run 4: time = 5.197000
run 5: time = 5.067000
run 6: time = 12.447000
run 7: time = 11.595000
run 8: time = 19.757000
```

SIMD

```
for (int i = 0; i < n; i++) {
  z[i] = x[i] * y[i];
}
```

Scalar	SIMD			
X	X3	X2	X1	X0
*	*			
Y	Y3	Y2	Y1	Y0
X * Y	X3 * Y3	X2 * Y2	X1 * Y1	X0 * Y0

- Single Instruction Multiple Data
- Instead of performing a single computation, the same operation is applied to **vector** of data
- Integer / Floating point
- Intel SSE / ARM NEON

Intel i7-6700K
“Skylake” 4.0GHz

Theoretical limits:

32-bit FP FMA	4.0GHz x 32 / cycle x 4 cores = 512Gflops
8-bit integer add	4.0GHz x 96 / cycle x 4 cores = 1536G”b”ops

Applications of SIMD

- Compression / decompression
- Graphic / video processing
  - Rendering web page content
- Gaming
  - Compute coordinates of objects / projectiles
- Signal processing
  - Amplification
  - Fast Fourier Transform
- Machine learning
  - Matrix multiplication

“foreach” loops make good use of SIMD instructions to process objects in parallel and independently

SIMD Registers

<table>
<tr><th></th><th colspan="16">xmm0 (128 bits)</th></tr>
<tr><td>4 x floats</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td></tr>
<tr><td>2 x doubles</td><td colspan="8">double</td><td colspan="8">double</td></tr>
<tr><td>16 x chars</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td></tr>
<tr><td>8 x shorts</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td></tr>
<tr><td>4 x ints</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td></tr>
<tr><td>2 x long longs</td><td colspan="8">long long</td><td colspan="8">long long</td></tr>
</table>

<table>
<tr><th></th><th colspan="32">ymm0 (256 bits)</th></tr>
<tr><td>8 x floats</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td></tr>
<tr><td>4 x doubles</td><td colspan="8">double</td><td colspan="8">double</td><td colspan="8">double</td><td colspan="8">double</td></tr>
<tr><td>32 x chars</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td><td>c</td></tr>
<tr><td>16 x shorts</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td><td colspan="2">s</td></tr>
<tr><td>8 x ints</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td><td colspan="4">int</td></tr>
<tr><td>4 x long longs</td><td colspan="8">long long</td><td colspan="8">long long</td><td colspan="8">long long</td><td colspan="8">long long</td></tr>
</table>

Intel SIMD Instructions

Technology	Year	Register Size	Features
MMX	1997	64 bit	Integers only
SSE	1999	128 bit	Integers and singles only
SSE2	2001		Added doubles
SSE3	2004		Added hadd and hsub for FP
SSSE3	2006		Added abs, hadd and hsub etc for integers
SSE4.1 4.2	2007		Added various conversion instructions
AVX	2011	256 bit	Added most 256 bit variants for FP
AVX2	2013		Added variants for integers
FMA	2014		Fused Multiply Add $0 = $0 + $1 x $2 etc

Intel SIMD

- Compilers are not good (enough) to **vectorize** code to use SIMD instructions

```
while (i < 10000) {
  c[i] = a[i] + b[i];
  i++;
}
```

```
00408290 <+64>:  mov    edx,DWORD PTR [r8+rax*1]
00408294 <+68>:  add    edx,DWORD PTR [rcx+rax*1]
00408297 <+71>:  mov    DWORD PTR [r9+rax*1],edx
0040829b <+75>:  add    rax,0x4
0040829f <+79>:  cmp    rax,0x9c40
004082a5 <+85>:  jne    0x408290 <main()+64>
```

- Luckily, we can use **Intel Intrinsics** (C functions) to use SIMD instructions in C programs

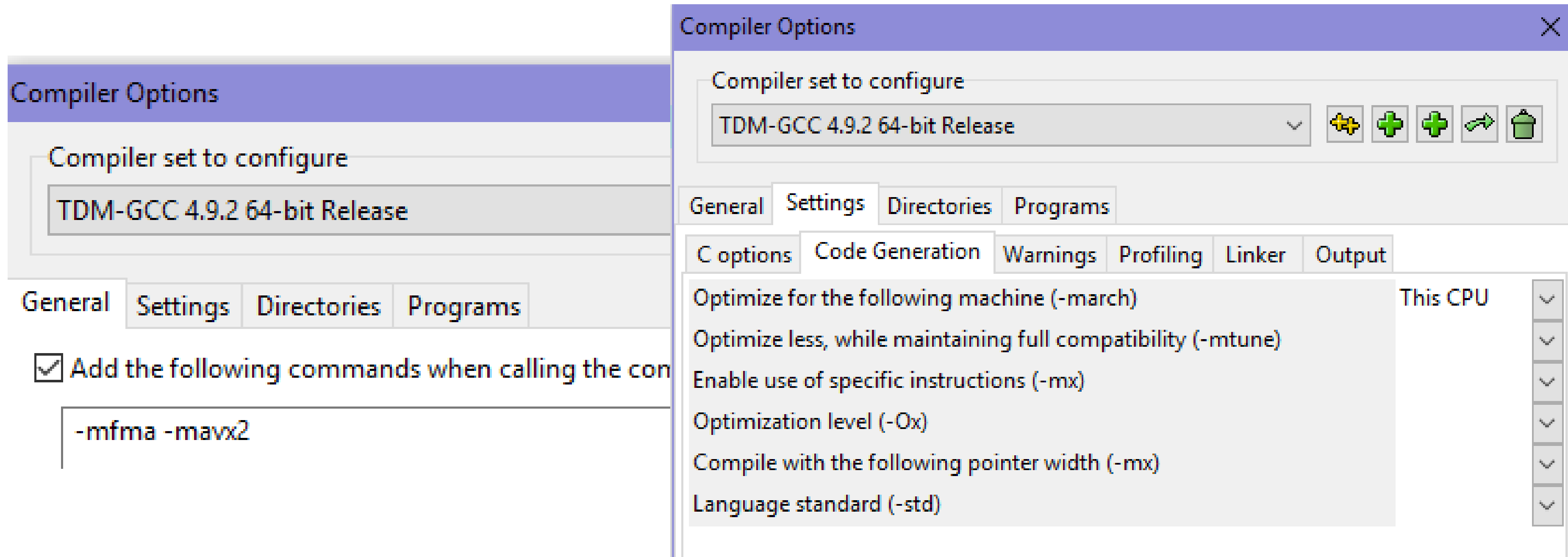
```
#include <immintrin.h>
```

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B, 3C and 3D

ocument contains all three volumes of the Intel 64 and IA-32 Architectures Software
Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order
383; *System Programming Guide*, Order Number 325384. Refer to all three volumes
ing your design needs

Configuring Dev-C++



Intel Intrinsics

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☑ AVX
- ☑ AVX2
- ☑ FMA
- ☐ AVX-512

__m256d (4 doubles)

__m256 (8 floats)

__m256i (integers)

`__m256d _mm256_add_pd (__m256d a, __m256d b)`

Synopsis

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
#include "immintrin.h"
Instruction: vaddpd ymm, ymm, ymm
CPUID Flags: AVX
```

Description

Add packed double-precision (64-bit) floating-point elements in a and b, and store the results in dst.

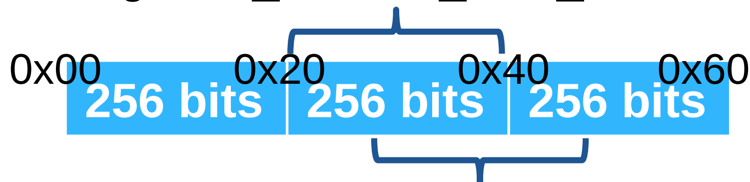
Operation

```
FOR j := 0 to 3
        i := j*64
        dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput
Haswell	3	1
Ivy Bridge	3	1
Sandy Bridge	3	1

aligned: _mm256_load_*



unaligned: _mm256_loadu_*

Latency and Throughput

- Throughput: How many burgers / minute
  - 1 = 1 instruction / cycle
  - 0.33 = 3 instructions / cycle (e.g. ADD)
- Latency: Time required for burger to be ready
  - Not relevant unless there are dependent instructions

C[i] = A[i] + B[i] example

```
1  #include <immintrin.h>
2  #include <stdio.h>
3  #include <time.h>
4  int a[10000], b[10000], c[10000];
5  int main() {
6    int i, j;
7    for (i = 0; i < 10000; i++) {
8      a[i] = i / 128;
9      b[i] = i % 128;
10   }
11   clock_t t = clock();
12   for (j = 0; j < 1000000; j++) {
13     i = 0;
14     while (i < 10000) {
15       c[i] = a[i] + b[i];
16       i++;
17     }
18   }
19   printf("%d\n", c[5000]);
20   t = clock() - t;
21   printf("%f seconds\n", (float)t / CLOCKS_PER_SEC);
22   return 0;
23 }
```

```
mov     0x201931(%rip),%eax
add     0x2151ab(%rip),%eax
mov     %eax,0x20b565(%rip)
cmp     $0x1,%r12d
je      400a30 <main+0x580>
mov     0x201919(%rip),%eax
add     0x215193(%rip),%eax
mov     %eax,0x20b54d(%rip)
cmp     $0x2,%r12d
je      400a20 <main+0x570>
```

When –O3 is specified, compiler unrolls the loop, but not vectorizes it

```
while (i < 10000) {
  __m256i x = _mm256_loadu_si256((void*) (a + i));
  __m256i y = _mm256_loadu_si256((void*) (b + i));
  __m256i z = _mm256_add_epi32(x, y);
  _mm256_storeu_si256((void*) (c + i), z);
  i += 8;
}
```

```
vmovdqu 0x6010c0(%rdx),%ymm0
vpaddd  0x614940(%rdx),%ymm0,%ymm0
add     $0x20,%rdx
vmovdqu %ymm0,0x60ace0(%rdx)
cmp     $0x9c40,%rdx
jne     400730 <main+0x280>
```

vectorized program