SIMD

Александр Шишков
Itseez, 2016

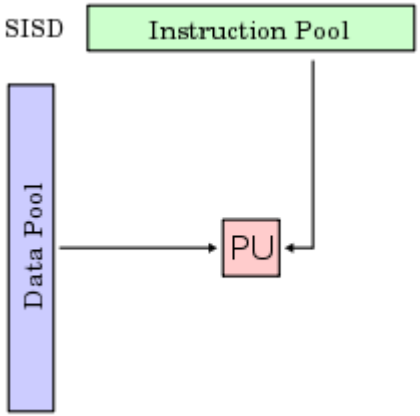
Параллелизм

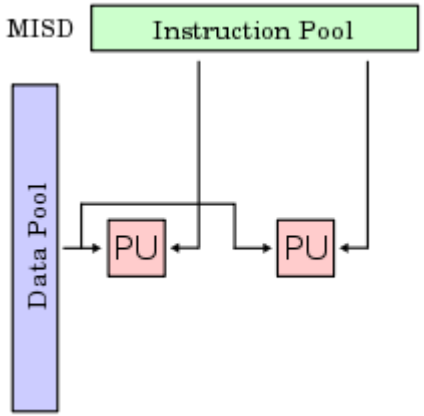
- системы памяти
- системный ввод-вывод
- мультиплексирование шин
- ISA
  - конвейеризация
  - суперскалярность
  - неупорядоченная выборка
  - механизм прерываний

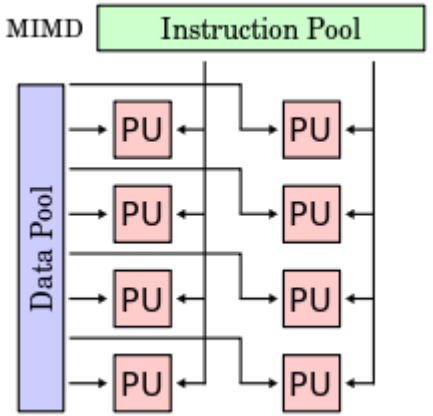
Классификация Флинна

Flynn M. (1966)

		Data Stream	
		Single	Multiple
Instruction Stream	Single	SISD	*SIMD*
	Multiple	MISD	MIMD

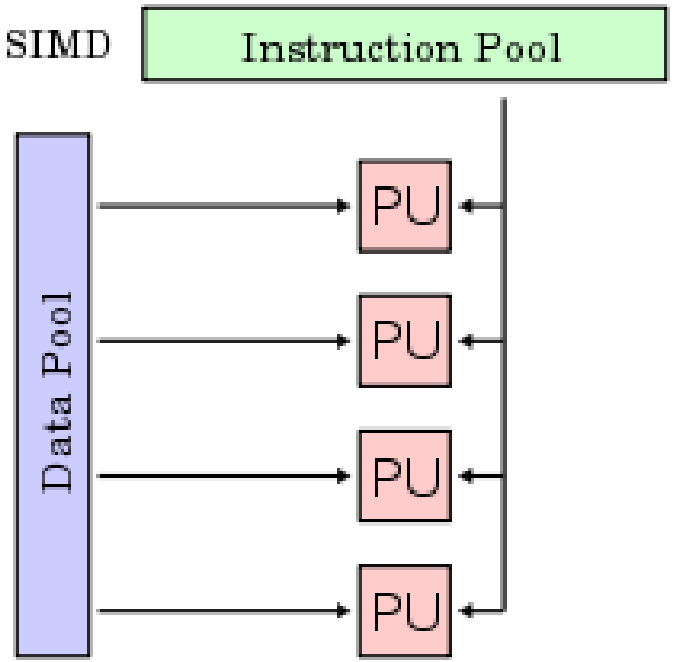






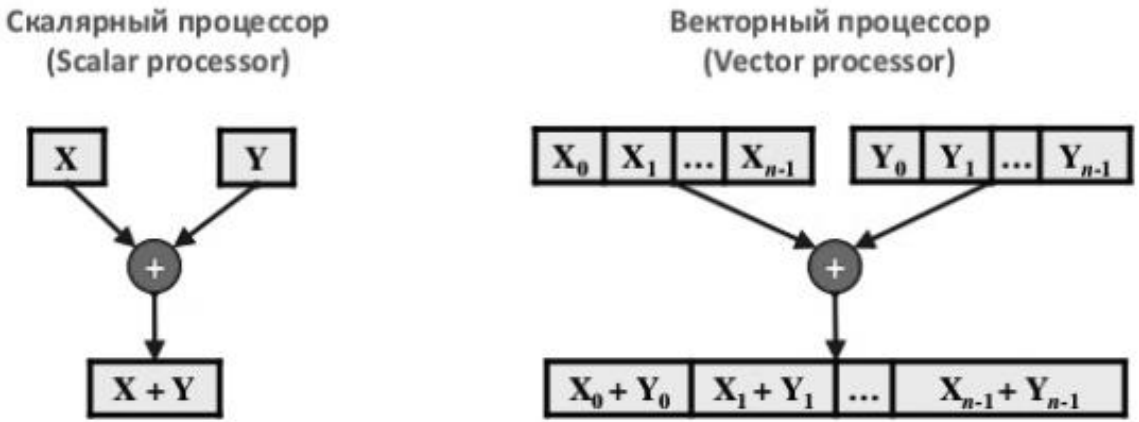
SIMD:

- *векторные процессоры*
- матричные процессоры
- …



Векторизация – выполнение операций над несколькими операндами (вектором) одновременно.

Векторный процессор – это процессор, поддерживающий на уровне системы команд операции для работы с одномерными массивами.



В чем плюсы?

- эффективное декодирование
- меньше операций с данными
- менший размер кода

```
for i = 1 to 10 do
IF – Instruction Fetch (next) ID – Instruction Decode
Load Operand1
Load Operand2
Add Operand1 Operand2
Store Result
end for
```

В чем плюсы?

- эффективное декодирование
- меньше операций с данными
- менший размер кода

```
IF – Instruction Fetch
ID – Instruction Decode
Load Operand1[0:9]
Load Operand2[0:9]
Add Operand1[0:9] Operand2[0:9]
Store Result
```

Cray 1 (1976) 80 MHz, 8 regs, 64 elems

Специализированные процессоры для векторной обработки

Высокая скорость работы с памятью

Нет кэша

Современные процессоры стандартной архитектуры (Intel и AMD) имеют векторные расширения.

Векторное расширение:

- Набор векторных регистров
- «Упаковка» — запись в один векторный регистр нескольких операндов — формирование вектора
- Набор команд для векторной обработки

Анализ большого количества приложений:

- графика
- MPEG видео
- сжатие речи
- обработка изображений
- игры
- видео конференции

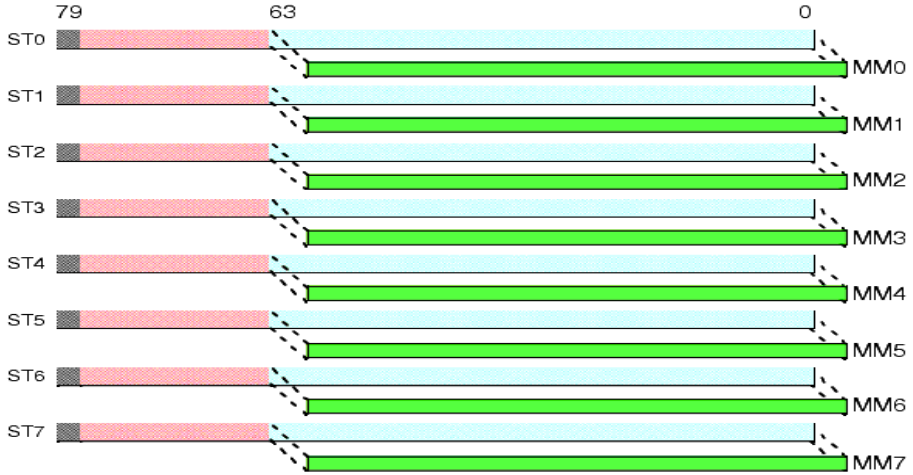
Результаты:

- Небольшие целочисленные типы (8-битные в случае графики, 16-битные для аудио)
- Простые циклы с большим количеством итераций
- Простые операции: сложение, умножение с накоплением и т.д.
- Итерации большинства циклов независимы

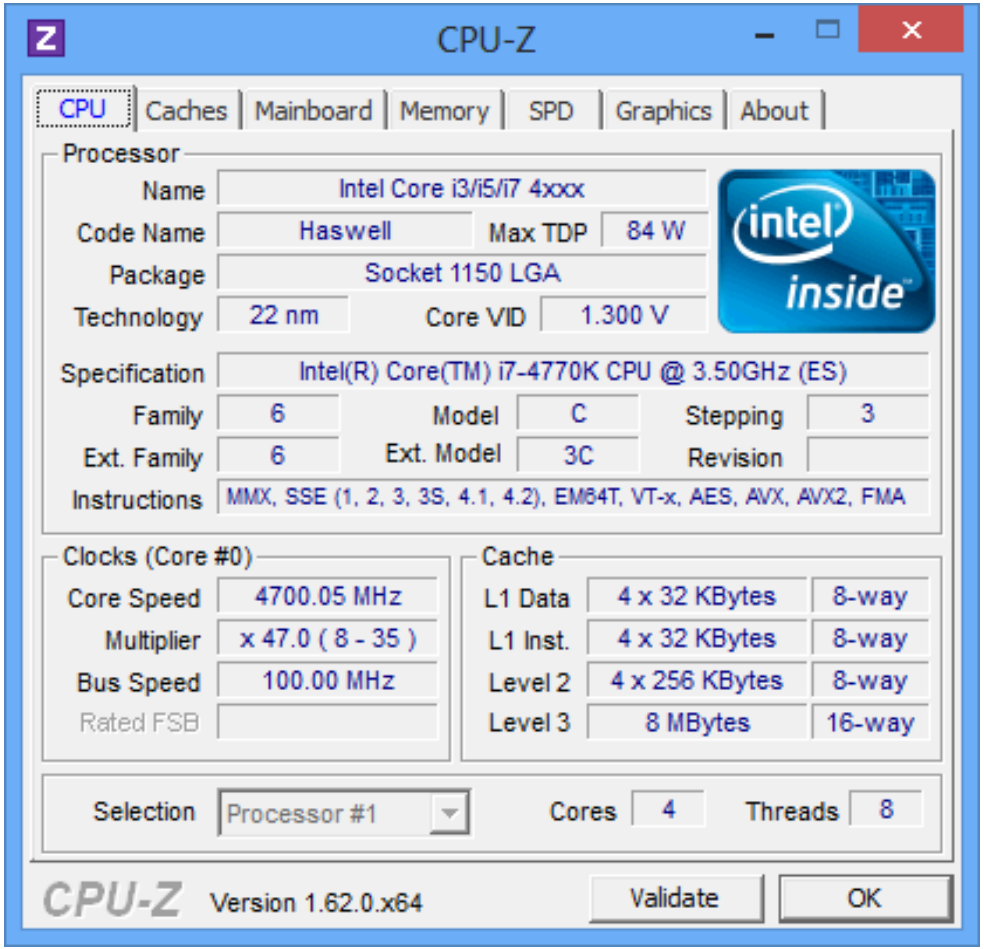
Intel MMX, 1997, Intel Pentium MMX

Набор базисных векторных инструкций для обработки целых чисел

- 57 новых инструкций
- 8 64-битных регистра
- 4 новых типа данных



Intel MMX: 1997, Intel Pentium MMX, IA-32
AMD 3DNow!: 1998, AMD K6-2, IA-32
Apple, IBM, Motorola AltiVec: 1998, PowerPC G4, G5, IBM Cell
Intel SSE (Streaming SIMD Extensions): 1999, Intel Pentium III
Intel SSE2: 2001, Intel Pentium 4, IA-32
Intel SSE3: 2004, Intel Pentium 4 Prescott, IA-32
Intel SSE4: 2006, Intel Core, AMD K10, x86-64
AMD SSE5 (XOP, FMA4, CVT16): 2007, 2009, AMD Buldozzer
Intel AVX: 2008, Intel Sandy Bridge
ARM Advanced SIMD (NEON): ARMv7, ARM Cortex A MIPS SIMD Architecture (MSA): 2012, MIPS R5
Intel AVX2: 2013, Intel Haswell
Intel AVX-512: Intel Xeon Phi
...


CPU-Z
CPU
Caches
Mainboard
Memory
SPD
Graphics
About
Processor
Name
Intel Core i3/i5/i7 4xxx
Code Name
Haswell
Max TDP
84 W
Package
Socket 1150 LGA
Technology
22 nm
Core VID
1.300 V
intel
inside
Specification
Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz (ES)
Family
6
Model
C
Stepping
3
Ext. Family
6
Ext. Model
3C
Revision
Instructions
MMX, SSE (1, 2, 3, 3S, 4.1, 4.2), EM64T, VT-x, AES, AVX, AVX2, FMA
Clocks (Core #0)
Core Speed
4700.05 MHz
Multiplier
x 47.0 (8 - 35)
Bus Speed
100.00 MHz
Rated FSB
Cache
L1 Data
4 x 32 KBytes
8-way
L1 Inst.
4 x 32 KBytes
8-way
Level 2
4 x 256 KBytes
8-way
Level 3
8 MBytes
16-way
Selection
Processor #1
Cores
4
Threads
8
CPU-Z
Version 1.62.0.x64
Validate
OK

Intel MMX

- для обработки целочисленных векторов длиной 64 бит
- 8 виртуальных регистров mm0, mm1, ..., mm7 – ссылки на физические регистры x87 FPU
- Типы векторов: 8 x 1 char, 4 x short int, 2 x int, 1 x int64
- MMX-инструкции разделяли x87 FPU
- с FP-инструкциями – требовалось оптимизировать поток инструкций (отдавать предпочтение инструкциям одного типа)

SSE (*Streaming SIMD Extensions*)

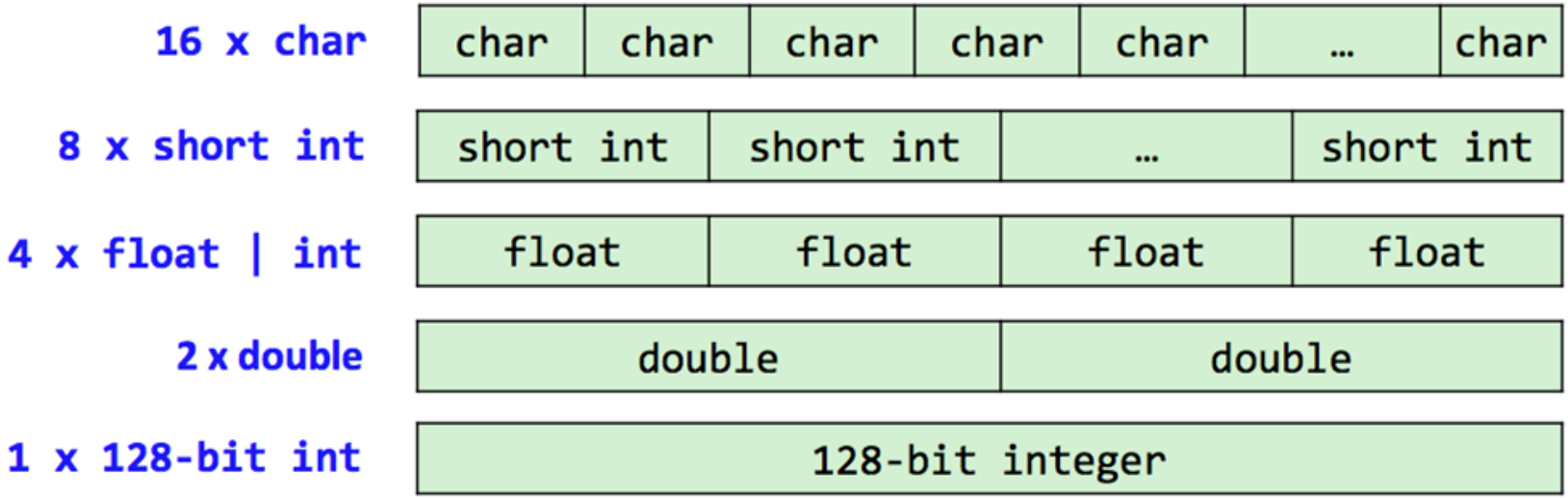
1999, Pentium III, ответ на 3DNow! от AMD

Проблемы MMX:

- Регистры совмещены с FPU
- Целочисленная арифметика
- 8 регистров 128 бит xmm0, …, xmm7
- float (4 элемента на вектор)
- 70 новых команд
- 32-битный управляющий регистр MXCSR

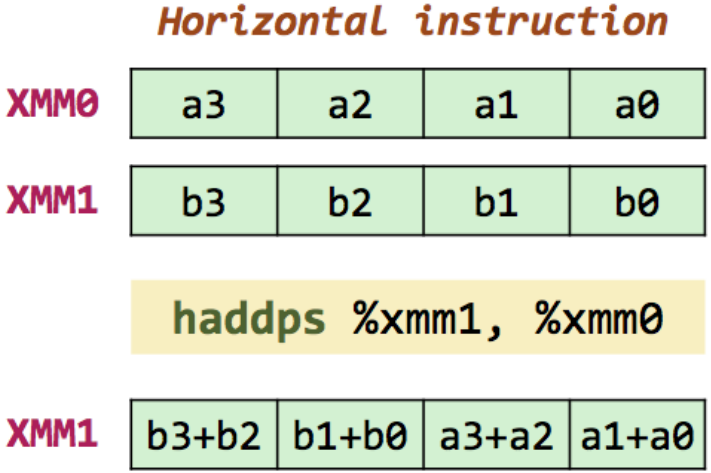
SSE2, Pentium IV, 2000

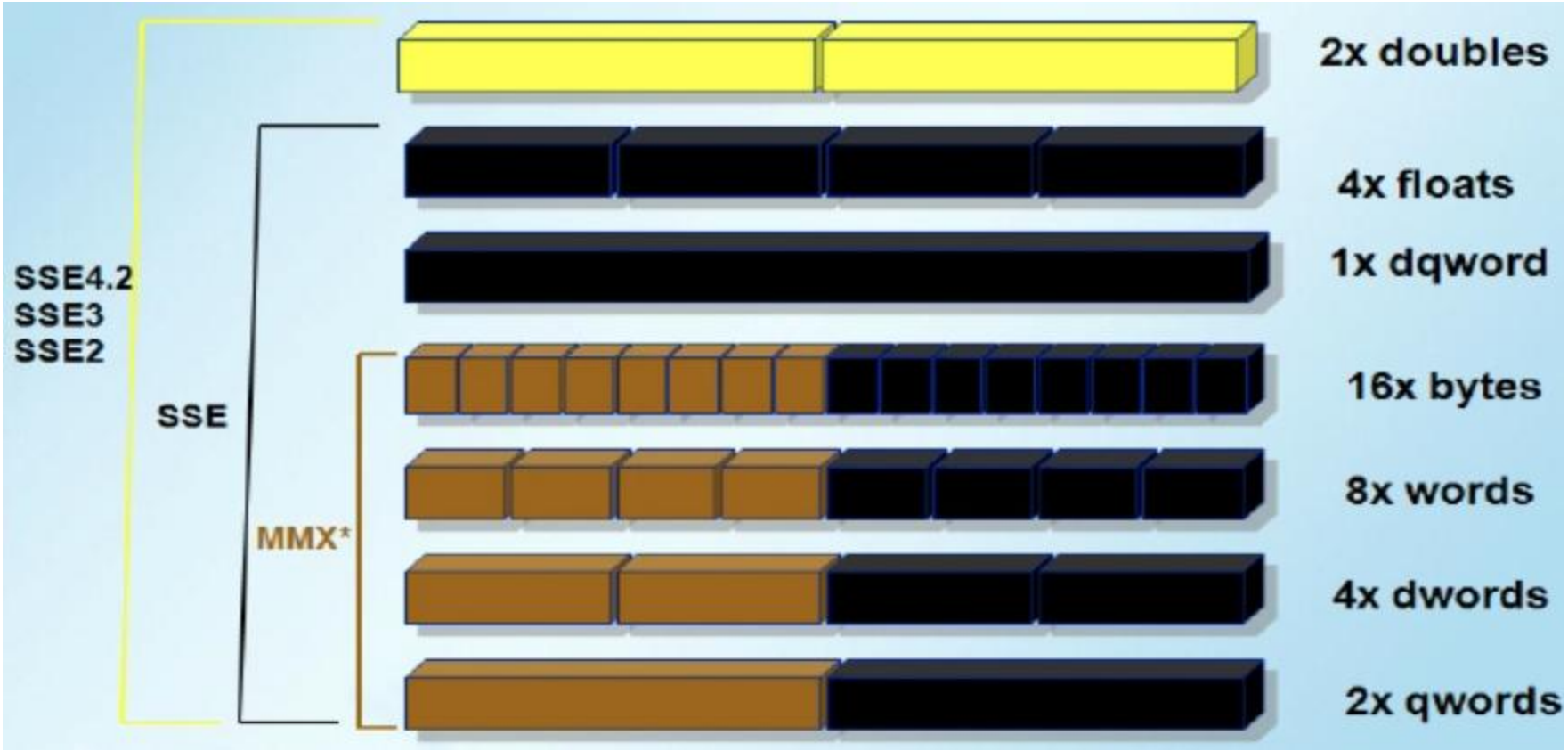
- Поддержка чисел с плавающей точкой двойной точности
- SSE2 содержит инструкции для потоковой обработки целочисленных данных в тех же 128-битных xmm регистрах. Вытеснение MMX.
- 16 векторных регистров шириной 128 бит: %xmm0, %xmm1, ..., %xmm7; %xmm8, ..., %xmm15
- Добавлено 144 инструкции к 70 инструкциям SSE
- По сравнению с SSE сопроцессор FPU (x87) обеспечивает более точный результат при работе с вещественными числами


16 x char
char
char
char
char
char
…
char
8 x short int
short int
short int
…
short int
4 x float | int
float
float
float
float
2 x double
double
double
1 x 128-bit int
128-bit integer

SSE3, Pentium IV, 2003

- 13 новых инструкций
- наиболее заметное изменение – горизонтальная работа с регистрами




SSE4.2
SSE3
SSE2
SSE
MMX*
2x doubles
4x floats
1x dqword
16x bytes
8x words
4x dwords
2x qwords

Advanced Vector Extensions (AVX), 2008

- Размер векторных регистров увеличивается с 128 до 256 бит (регистры YMM0 — YMM15).
  - Существующие 128-битные инструкции будут использовать младшую половину новых YMM регистров.
- Неразрушающие операции.
  - Набор инструкций AVX позволяет использовать любую двухоперандную инструкцию XMM в трёхоперандном виде без модификации двух регистров-источников, с отдельным регистром для результата.

Векторные регистры пе 0, ymm1, ..., ymm15

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

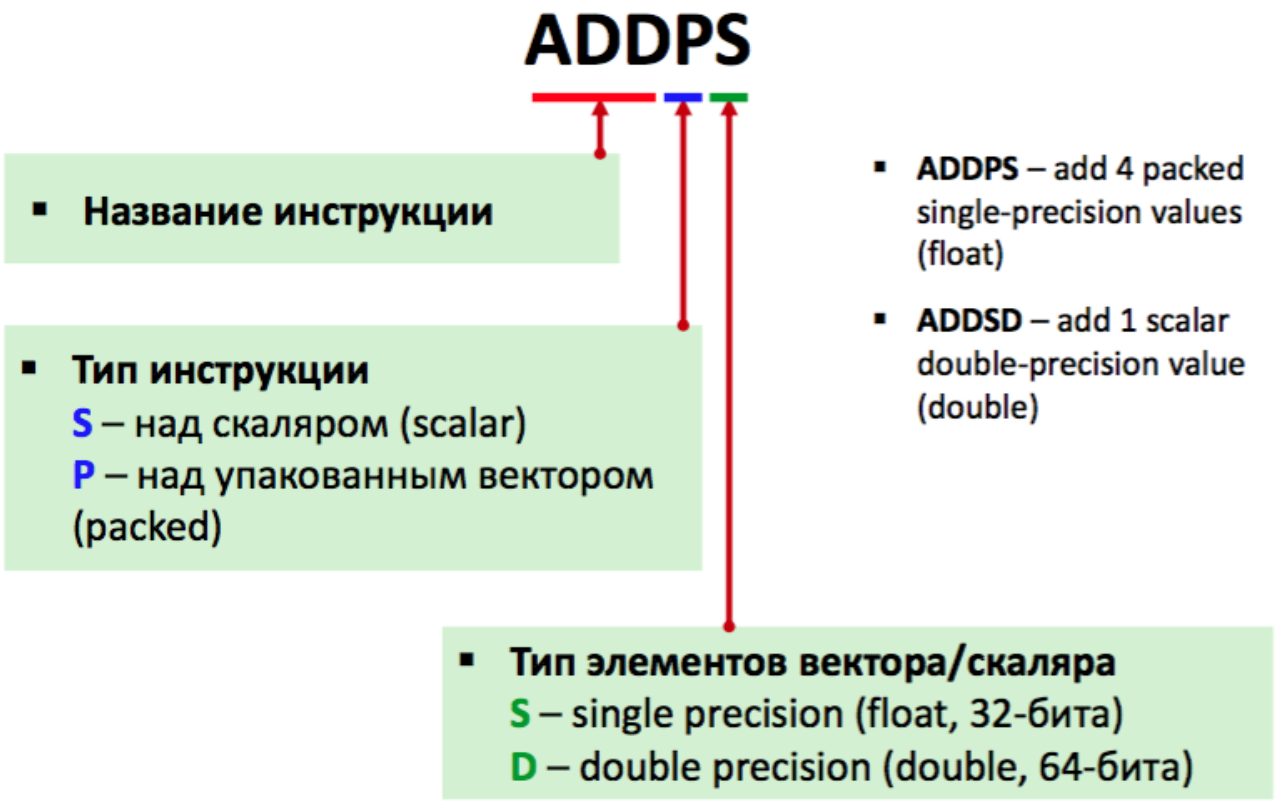
511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

YM
YM
YM
YM
YM$_f$
YM
YM
YM
YM
YM
YM
YM
YM
YM
YM
YM

0

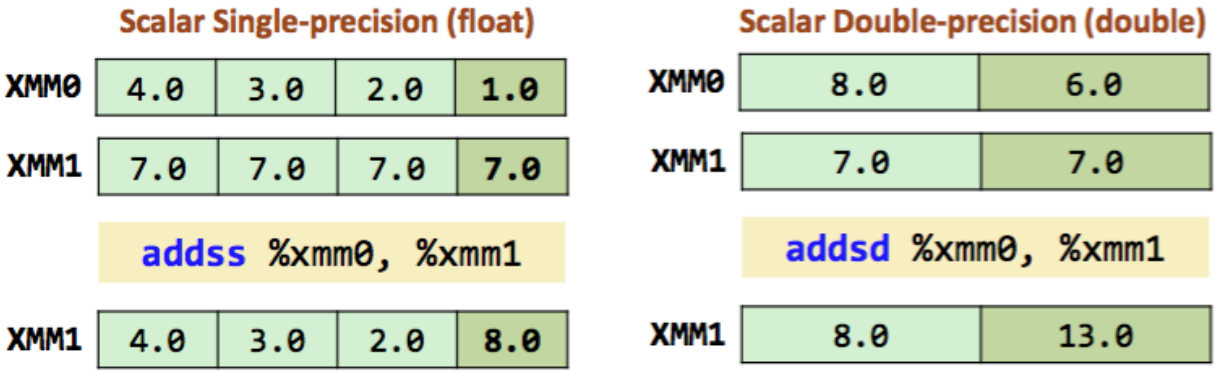
M0
M1
M2
M3
M4
M5
M6
M7
M8
M9
M10
M11
M12
M13
M14
M15

Формат инструкций



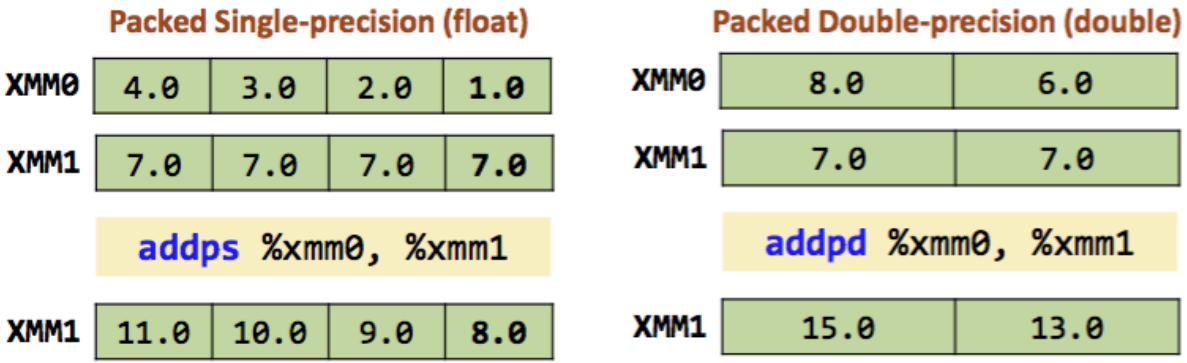
Скалярные SSE-инструкции (scalar instruction) – в операции участвуют только младшие элементы данных (скаляры) в векторных регистрах/памяти

ADDSS, SUBSS, MULSS, DIVSS, ADDSD, SUBSD, MULSD, DIVSD, SQRTSS, RSQRTSS, RCPSS, MAXSS, MINSS, …



SSE-инструкция над упакованными векторами (packed instruction) – в операции участвуют все элементы данных векторных регистров/памяти

ADDPS, SUBPS, MULPS, DIVPS, ADDPD, SUBPD, MULPD, DIVPD, SQRTPS, RSQRTPS, RCPPS, MAXPS, MINPS, ...



Арифметические SSE-инструкции

Arithmetic	Scalar Operator	Packed Operator
$y = y + x$	addss	addps
$y = y - x$	subss	subps
$y = y \times x$	mulss	mulps
$y = y \div x$	divss	divps
$y = \frac{1}{x}$	rcpss	rcpps
$y = \sqrt{x}$	sqrtss	sqrtps
$y = \frac{1}{\sqrt{x}}$	rsqrtss	rsqrtps
$y = \max(y, x)$	maxss	maxps
$y = \min(y, x)$	minss	minps

Операции копирования данных (mem-reg/reg-mem/reg-reg)

- Scalar: MOVSS
- Packed: MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS

Операции сравнения

- Scalar: CMPSS, COMISS, UCOMISS
- Packed: CMPPS

Поразрядные логические операции

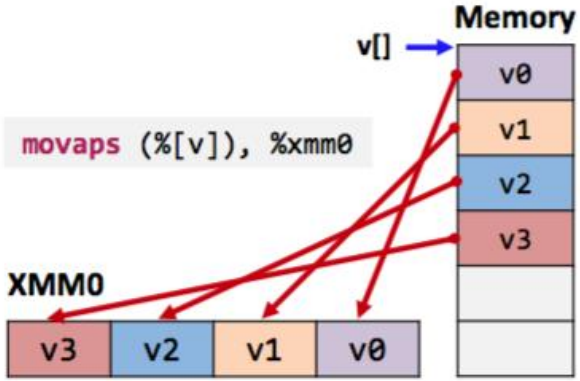
- Packed: ANDPS, ORPS, XORPS, ANDNPS

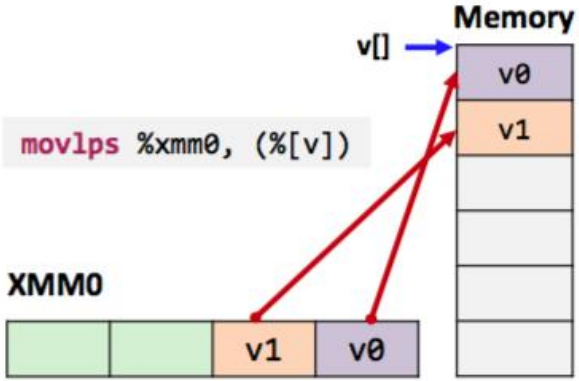
...

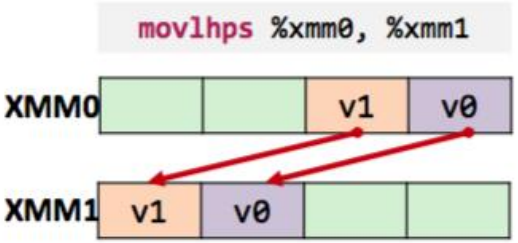
SSE-инструкции копирования данных

- MOVSS: Copy a single floating-point data
- MOVLPS: Copy 2 floating-point data (low packed)
- MOVHPS: Copy 2 floating-point data (high packed)
- MOVAPS: Copy aligned 4 floating-point data (fast)
- MOVUPS: Copy unaligned 4 floating-point data (slow)
- MOVHLPS: Copy 2 high elements to low position
- MOVLHPS: Copy 2 low elements to high position

SSE-инструкции копирования данных







Использование инструкций SSE

```
void add(float *a, float *b, float *c)
{
   int i;
    for (i = 0; i < 4; i++)
    {
       c[i] = a[i] + b[i];
    }
}
```

```
void add_sse_asm(float *a, float *b, float *c)
{
    __asm__ __volatile__ (
        "movaps (%[a]), %%xmm0 \n\t"
        "movaps (%[b]), %%xmm1 \n\t"
        "addps %%xmm1, %%xmm0 \n\t"
       "movaps %%xmm0, %[c] \n\t"
            : [c] "=m" (*c)
            : [a] "r" (a), [b] "r" (b)
            : "%xmm0", "%xmm1“      /* Clobbered regs */
    );
}
```

```
#include <fvec.h>  /* SSE classes */
void add(float *a, float *b, float *c)
{
    F32vec4 *av = (F32vec4 *)a;
    F32vec4 *bv = (F32vec4 *)b;
    F32vec4 *cv = (F32vec4 *)c;
    *cv = *av + *bv;
}
```

F32vec4 – класс, представляющий массив из 4 элементов типа float

Только для Intel C++ compiler

SSE Intrinsics (builtin functions)

Intrinsics – набор встроенных функций и типов данных, поддерживаемых компилятором, для предоставления высокоуровневого доступа к SSE-инструкциям

Компилятор самостоятельно распределяет XMM/YMM регистры, принимает решение о способе загрузки данных из памяти (проверяет выравнен адрес или нет) и т.п.

Заголовочные файлы

```
#include <mmintrin.h>  /* MMX */
#include <xmmintrin.h> /* SSE, нужен также mmintrin.h */
#include <emmintrin.h> /* SSE2, нужен также xmmintrin.h */
#include <pmmintrin.h> /* SSE3, нужен также emmintrin.h */
#include <smmintrin.h>  /* SSE4.1 */
#include <nmmintrin.h> /* SSE4.2 */
#include <immintrin.h>  /* AVX */
```

Типы данных

```
void main() {
    __m128 f; /* float[4] */
    __m128d d; /* double[2] */

    __m128i i; /* char[16], short int[8], int[4],
    uint64_t [2] */
}
```

Названия intrinsic-функций

_mm_<intrinsic_name>_<suffix>

```
void main() {
   float v[4] = {1.0, 2.0, 3.0, 4.0};
   __m128 t1 = _mm_load_ps(v); // v must be 16-byte
aligned
   __m128 t2 = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
}
```

```
#include <xmmintrin.h>   /* SSE */

void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Хранимые в памяти операнды SSE-инструкций должны быть размещены по адресу, выравненному на границу в 16 байт:

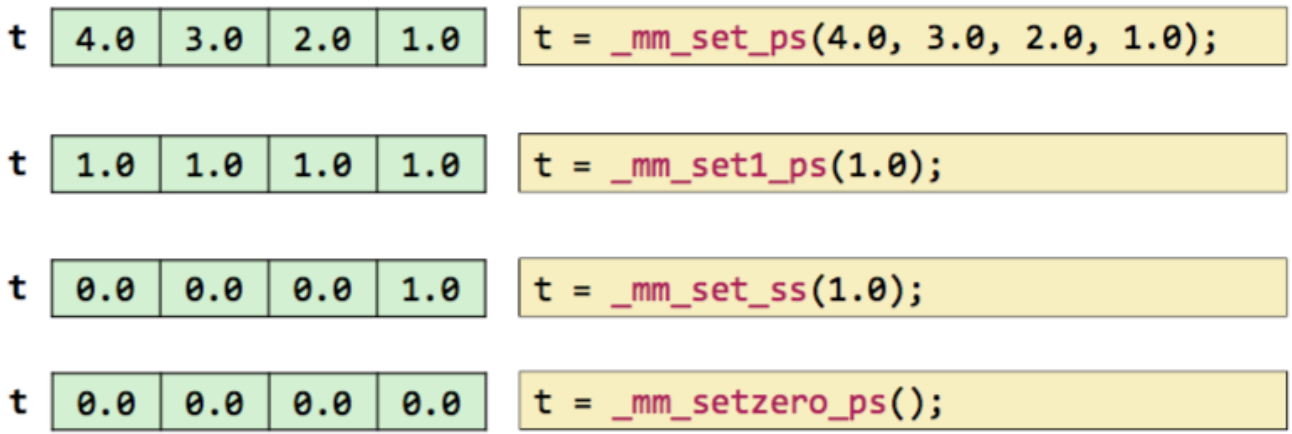
```
//MSVC
__declspec(align(16)) float A[N];
#include <malloc.h>
void *_aligned_malloc(size_t size, size_t
alignment); void _aligned_free(void *memblock);

//gcc
float A[N] __attribute__((aligned(16)))
```

Функции копирования данных

```
#include <xmmintrin.h>    /* SSE */
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`__m128 _mm_load_ss(float * p)`	Load the low value and clear the three high values	MOVSS
`__m128 _mm_load1_ps(float * p)`	Load one value into all four words	MOVSS + Shuffling
`__m128 _mm_load_ps(float * p)`	Load four values, address aligned	MOVAPS
`__m128 _mm_loadu_ps(float * p)`	Load four values, address unaligned	MOVUPS
`__m128 _mm_loadr_ps(float * p)`	Load four values in reverse	MOVAPS + Shuffling

Функции копирования данных



```
#include <xmmintrin.h>    /* SSE */
```

Intrinsic Name	Operation	Corresponding SSE Instruction
__m128 _mm_add_ss(__m128 a, __m128 b)	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

```
#include <emmintrin.h>     /* SSE2 */
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
`__m128d _mm_add_sd(__m128d a, __m128d b)`	Addition	ADDSD
`_mm_add_pd`	Addition	ADDPD
`_mm_sub_sd`	Subtraction	SUBSD
`_mm_sub_pd`	Subtraction	SUBPD
`_mm_mul_sd`	Multiplication	MULSD
`_mm_mul_pd`	Multiplication	MULPD
`_mm_div_sd`	Division	DIVSD
`_mm_div_pd`	Division	DIVPD
`_mm_sqrt_sd`	Computes Square Root	SQRTSD
`_mm_sqrt_pd`	Computes Square Root	SQRTPD
`_mm_min_sd`	Computes Minimum	MINSD
`_mm_min_pd`	Computes Minimum	MINPD
`_mm_max_sd`	Computes Maximum	MAXSD
`_mm_max_pd`	Computes Maximum	MAXPD

Автоматическая векторизация

- Выполняется компилятором без участия программиста
- Компилятор не всегда находит возможность для векторизации
- Программист может подсказать компилятору

- Visual C++ 2012 – векторизация включена по умолчанию (при использовании опции /O2, подробный отчет формируется опцией /Qvec-report)
- GNU GCC – векторизация включается при использовании опции -O3 или -ftree-vectorize
- Intel C++ Compiler – векторизация включена по умолчанию (при использовании опции /O2, -O2, подробный отчет формируется опцией /Qvec-report, -vec-report)

Intel C++ Compiler

Возможные типы инструкций:

AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2

Опции:

-x – генерирует код только для данного типа инструкций

-m – генерирует код для заданного типа инструкций с проверкой поддержки их процессором

-ax – генерирует код для заданного типа инструкций, а также универсальную версию кода

-xhost – генерирует код для компьютера, на котором запускается с максимальным уровнем инструкций

Код только для инструкций AVX:

```
$ icc -O2 -xAVX vect1.c -o vect1
```

Код для инструкций AVX с альтернативной версией для процессоров не от Intel:

```
$ icc -O2 -axAVX vect1.c -o vect1
```

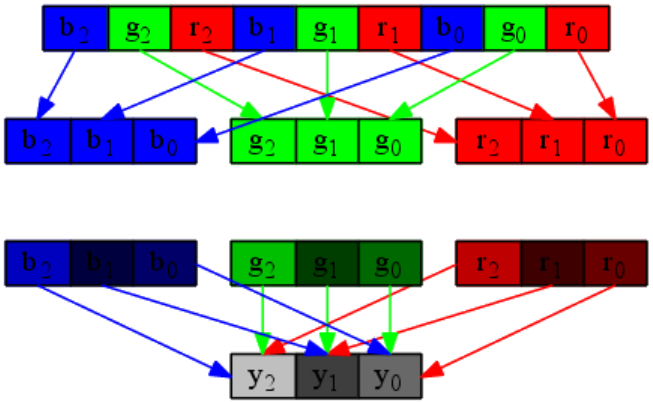
Код для инструкций SSE4.2 с проверкой совместимости:

```
$ icc -O2 -mSSE4.1 vect1.c -o vect1
```

```
#if defined (__INTEL_COMPILER)
#pragma vector always
#endif
for (i = 0; i < 100; i++)
{
   k = k + 10;
   a[i] = k;
}
```

```
...
movsxd  rax, dword ptr [rbp - 28]
mov     rcx, qword ptr [rbp - 8]
movss   xmm0, dword ptr [rcx + 4*rax]
movsxd  rax, dword ptr [rbp - 28]
mov     rcx, qword ptr [rbp - 16]
addss   xmm0, dword ptr [rcx + 4*rax]
movsxd  rax, dword ptr [rbp - 28]
mov     rcx, qword ptr [rbp - 24]
movss   dword ptr [rcx + 4*rax], xmm0
...
```

cvtColor RGB->gray



SSE операции с памятью

```
uint64_t *aPtr = ...

// Загрузить 2 unsigned 64-bit int из памяти

__m128i a = _mm_loadu_si128(aPtr);

// Сохранить 2 unsigned 64-bit ints в память

_mm_storeu_si128(aPtr, a);


int8_t *bPtr = ...

// Загрузить 16 signed 8-bit ints из памяти
```

SSE арифметические операции

Сделать операцию **<op>** для типа **<type>** над **a** и **b**, записать результат в **c**:

```
__m128i c = _mm_<op>_<type>(a, b);
```

SSE арифметические операции

суффикс **<type>** определяет как интерпретировать __m128:

- **epi8** = **e**xtended **p**acked **8**-bit **i**nteger:

```
__m128i c = _mm_min_epi8(a, b);
```

16 попарных минимумов **8**-bit signed **i**nts

- **epu16** = **e**xtended **p**acked **16**-bit **u**nsigned integer:

```
__m128i c = _mm_min_epu16(a, b);
```

SSE арифметические операции

```
__m128i c = _mm_add_epi16(a, b);
```

8 попарных сумм 16-bit ints

```
__m128i c = _mm_sub_epi32(a, b);
```

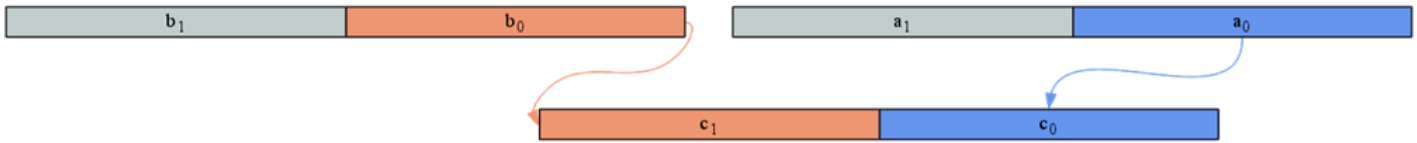
4 попарные разности 32-bit ints

```
__m128i c = _mm_srli_epi16(a, b);
```

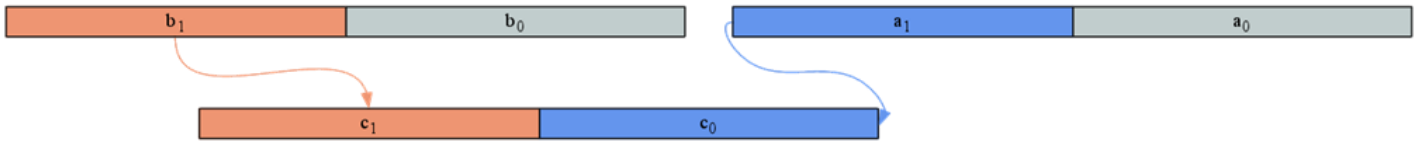
8 попарных сдвигов вправо для 16-bit ints в a на константу b

SSE операции распаковки

```
c = _mm_unpacklo_epi64(a, b);
```

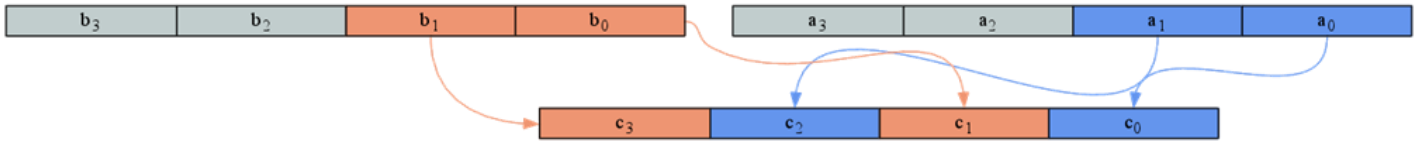



```
c = _mm_unpackhi_epi64(a, b);
```

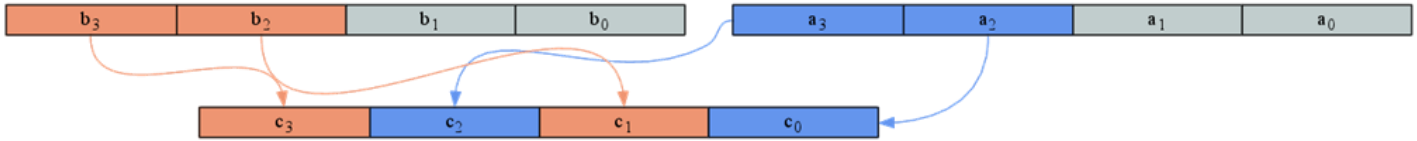


SSE операции распаковки

```
c = _mm_unpacklo_epi32(a, b);
```

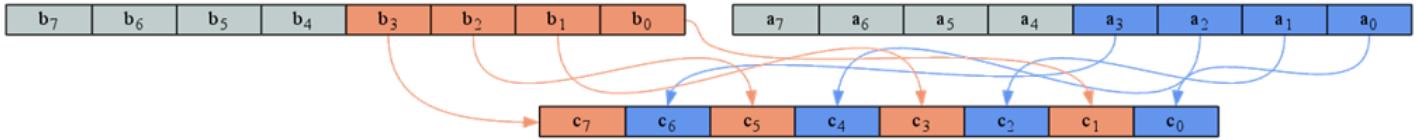



```
c = _mm_unpackhi_epi32(a, b);
```

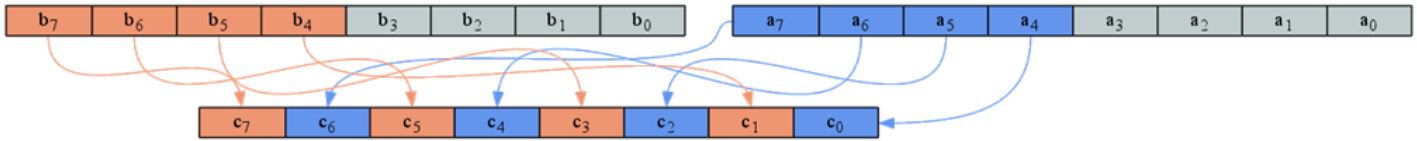


SSE операции распаковки

```
c = _mm_unpacklo_epi16(a, b);
```

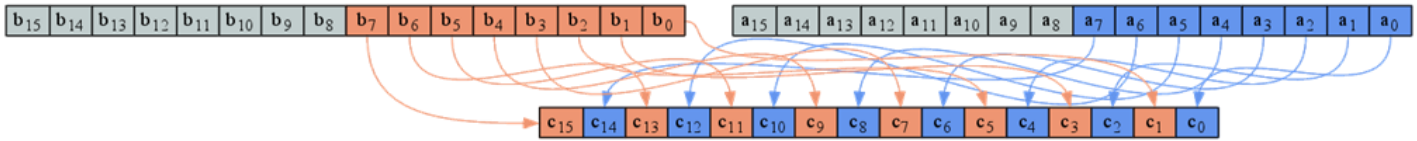



```
c = _mm_unpackhi_epi16(a, b);
```

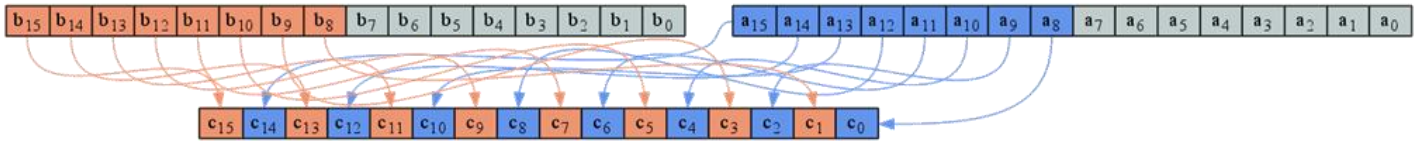


SSE операции распаковки

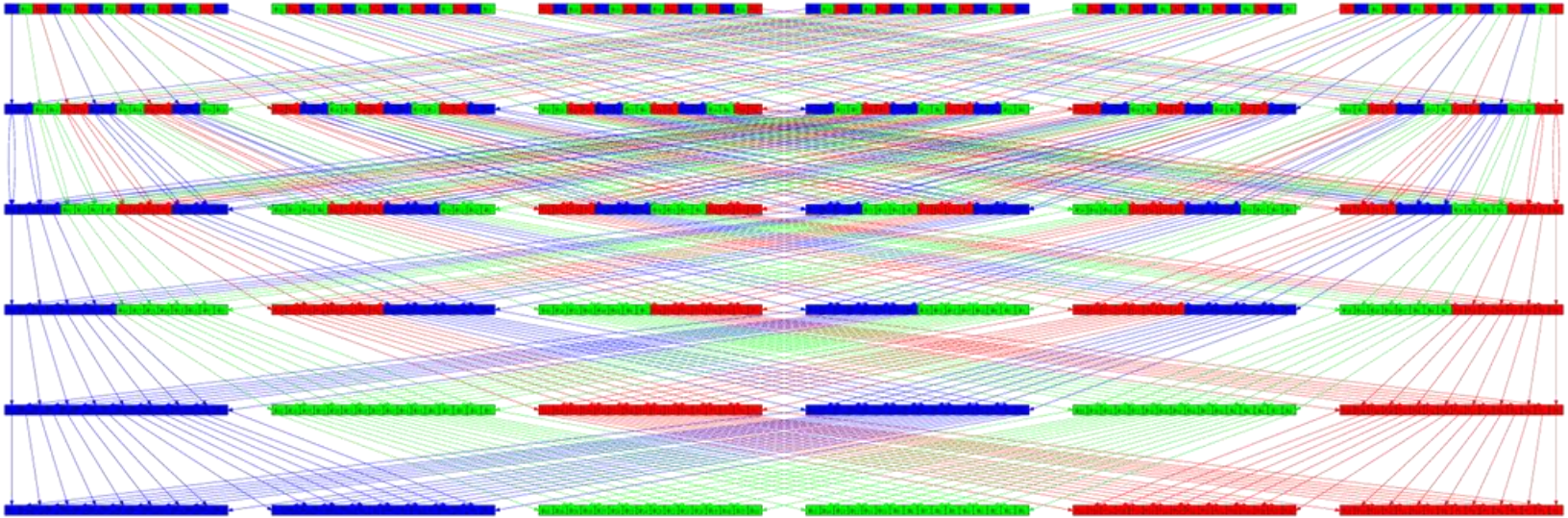
```
c = _mm_unpacklo_epi8(a, b);
```



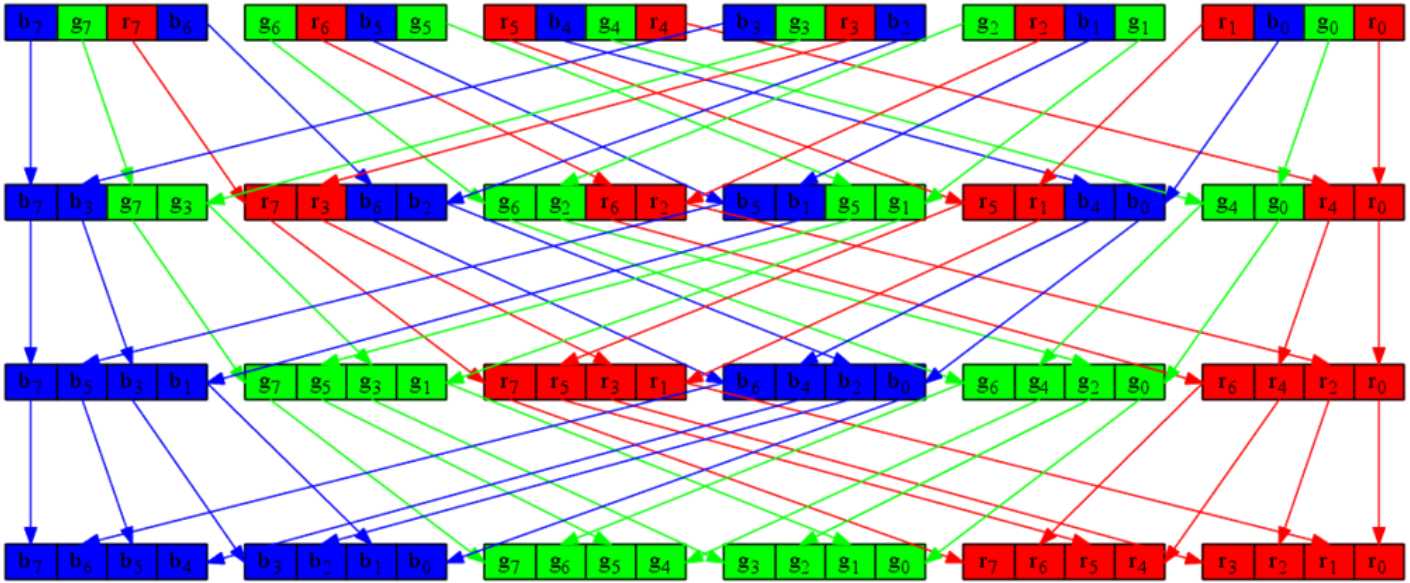

```
c = _mm_unpackhi_epi8(a, b);
```



RGB deinterleaving



RGB deinterleaving



BGR deinterleaving

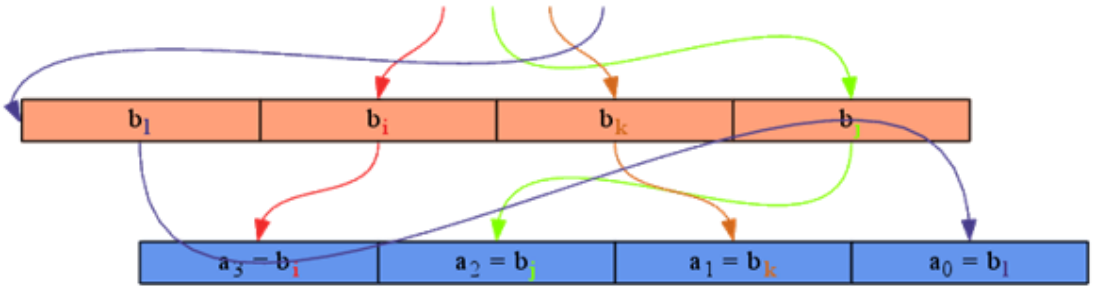
```
__m128i layer0_chunk0 = _mm_loadu_si128((__m128i*)source_pixels);
__m128i layer0_chunk1 = _mm_loadu_si128((__m128i*)(source_pixels + 16));
__m128i layer0_chunk2 = _mm_loadu_si128((__m128i*)(source_pixels + 32));
__m128i layer0_chunk3 = _mm_loadu_si128((__m128i*)(source_pixels + 48));
__m128i layer0_chunk4 = _mm_loadu_si128((__m128i*)(source_pixels + 64));
__m128i layer0_chunk5 = _mm_loadu_si128((__m128i*)(source_pixels + 80));

__m128i layer1_chunk0 = _mm_unpacklo_epi8(layer0_chunk0, layer0_chunk3);
__m128i layer1_chunk1 = _mm_unpackhi_epi8(layer0_chunk0, layer0_chunk3);
__m128i layer1_chunk2 = _mm_unpacklo_epi8(layer0_chunk1, layer0_chunk4);
__m128i layer1_chunk3 = _mm_unpackhi_epi8(layer0_chunk1, layer0_chunk4);
__m128i layer1_chunk4 = _mm_unpacklo_epi8(layer0_chunk2, layer0_chunk5);
__m128i layer1_chunk5 = _mm_unpackhi_epi8(layer0_chunk2, layer0_chunk5);
..
__m128i red_chunk0 = _mm_unpacklo_epi8(layer4_chunk0, layer4_chunk3);
__m128i red_chunk1 = _mm_unpackhi_epi8(layer4_chunk0, layer4_chunk3);
__m128i green_chunk0 = _mm_unpacklo_epi8(layer4_chunk1, layer4_chunk4);
__m128i green_chunk1 = _mm_unpackhi_epi8(layer4_chunk1, layer4_chunk4);
__m128i blue_chunk0 = _mm_unpacklo_epi8(layer4_chunk2, layer4_chunk5);
__m128i blue_chunk1 = _mm_unpackhi_epi8(layer4_chunk2, layer4_chunk5);
source_pixels += 96;
```

SSE shuffle инструкции

c = _mm_**shuffle_epi32**(**a**, **n**);

- Переупорядочивает 4 32-bit integers в **__m128i**
- _MM_SHUFFLE макрос для упрощения маски

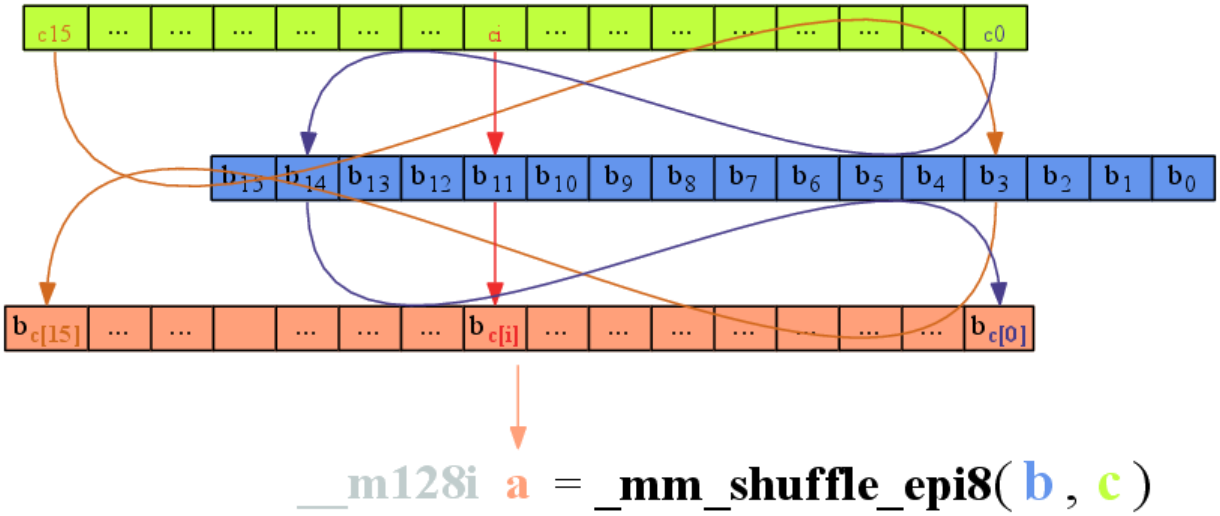
a = _mm_shuffle_epi32(b , _MM_SHUFFLE(i , j , k , l))



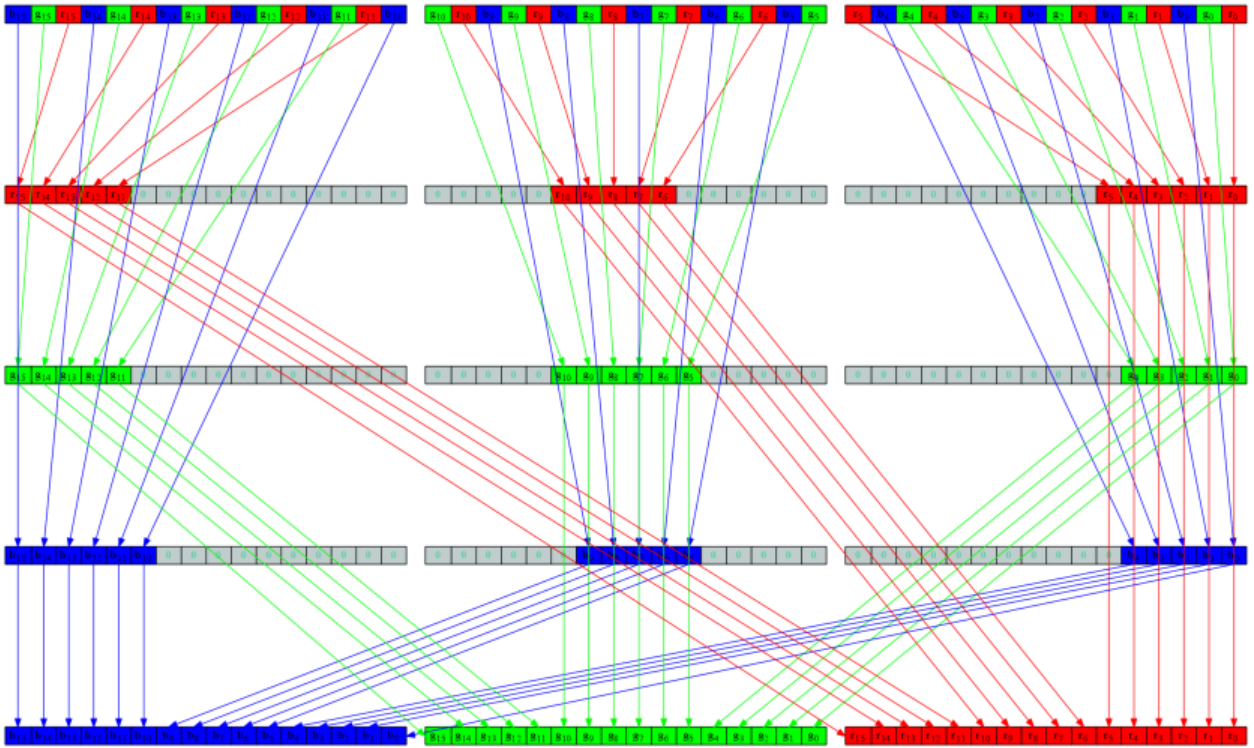
SSE shuffle инструкции

```
a = _mm_shuffle_epi8(b, c);
```

- Если позиция <0, то в **a** пишется ноль



RGB deinterleaving



BGR deinterleaving

```
__m128i ssse3_blue_indices_0 = _mm_set_epi8(-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 15, 12, 9, 6, 3, 0);
__m128i ssse3_blue_indices_1 = _mm_set_epi8(-1, -1, -1, -1, -1, 14, 11, 8, 5, 2, -1, -1, -1, -1, -1, -1);
__m128i ssse3_blue_indices_2 = _mm_set_epi8(13, 10, 7, 4, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1);
__m128i ssse3_green_indices_0 = _mm_set_epi8(-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 13, 10, 7, 4, 1);
__m128i ssse3_green_indices_1 = _mm_set_epi8(-1, -1, -1, -1, -1, 15, 12, 9, 6, 3, 0, -1, -1, -1, -1, -1);
__m128i ssse3_green_indices_2 = _mm_set_epi8(14, 11, 8, 5, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1);
__m128i ssse3_red_indices_0 = _mm_set_epi8(-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 14, 11, 8, 5, 2);
__m128i ssse3_red_indices_1 = _mm_set_epi8(-1, -1, -1, -1, -1, -1, 13, 10, 7, 4, 1, -1, -1, -1, -1, -1);
__m128i ssse3_red_indices_2 = _mm_set_epi8(15, 12, 9, 6, 3, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1);
```

BGR deinterleaving

```
const __m128i chunk0 = _mm_loadu_si128((const __m128i*)(row_data+x*3));
const __m128i chunk1 = _mm_loadu_si128((const __m128i*)(row_data + x*3 + 16));
const __m128i chunk2 = _mm_loadu_si128((const __m128i*)(row_data + x*3 + 32));

const __m128i red = _mm_or_si128(_mm_or_si128(_mm_shuffle_epi8(chunk0,
ssse3_red_indices_0),_mm_shuffle_epi8(chunk1, ssse3_red_indices_1)),
_mm_shuffle_epi8(chunk2, ssse3_red_indices_2));
```

Приведение 8bit в 16bit:

_mm_cvtepu8_epi16(...)

_mm_unpackhi_epi8(...)

Обратная склейка:

_mm_packus_epi16(...)

Отладочная печать

```
template <typename T>
std::string __m128i_toString(const __m128i var) {
    std::stringstream sstr;
    const T* values = (const T*) &var;
    if (sizeof(T) == 1) {
        for (unsigned int i = 0; i < sizeof(__m128i); i++) {
            sstr << (int) values[i] << " ";
        }
    } else {
        for (unsigned int i = 0; i < sizeof(__m128i) / sizeof(T); i++) {
            sstr << values[i] << " ";
        }
    }
    return sstr.str();
}

std::cout << __m128i_toString<uint8_t>(chunk0) << std::endl;
```

Дополнительные материалы

- Курс “Высокопроизводительные вычислительные системы”
  - http://www.slideshare.net/mkurnosov
- Using Intel® AVX without Writing AVX
  - http://software.intel.com/en-us/articles/using-intel-avx-without-writing-avx
- Intel SSE4 Programming Reference
  - http://software.intel.com/sites/default/files/m/9/4/2/d/5/17971-intel_20sse4_20programming_20reference.pdf
- Intel 64 and IA-32 Architectures Software Developer’s Manual (Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C)
  - http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

Вопросы