# Лекция 3: Векторизация кода (Code vectorization)

**Курносов Михаил Георгиевич**

**к.т.н. доцент Кафедры вычислительных систем**
**Сибирский государственный университет**
**телекоммуникаций и информатики**

**http://www.mkurnosov.net**

# Векторные процессоры

- **Векторный процессор (Vector processor) –** процессор поддерживающий на уровне системы команд операции для работы с одномерными массивами – векторами.

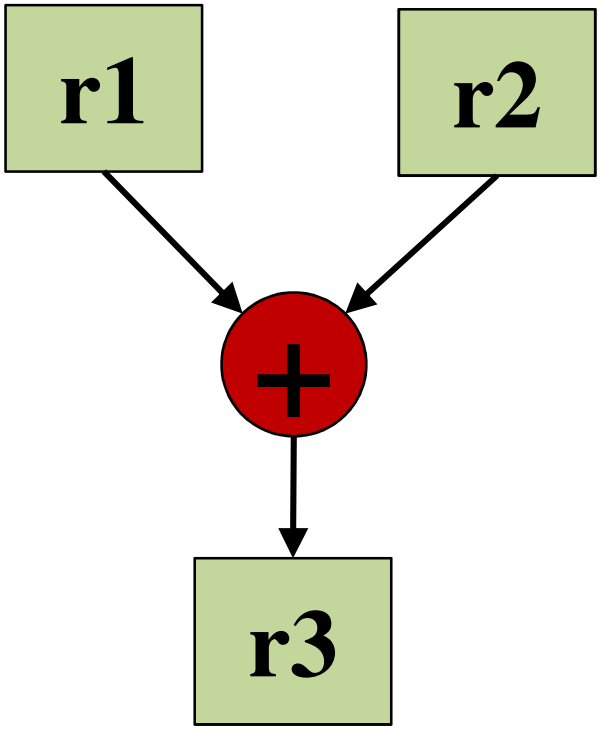
- **Векторные вычислительные системы**
  - CDC STAR-100 (1972 г., векторы до 65535 элементов)
  - Cray Research Inc.: Cray-1 (векторные регистры), Cray-2, Cray X-MP, Cray Y-MP

# Векторные вычислительные системы

- Cray 1 (1976) 80 MHz, 8 regs, 64 elems
- Cray XMP (1983) 120 MHz 8 regs,  64 elems
- Cray YMP (1988) 166 MHz 8 regs,  64 elems
- Cray C-90 (1991) 240 MHz 8 regs, 128 elems
- Cray T-90 (1996) 455 MHz 8 regs, 128 elems
- Conv. C-1 (1984) 10 MHz 8 regs, 128 elems
- Conv. C-4 (1994) 133 MHz 16 regs, 128 elems
- Fuj. VP200 (1982 133 MHz 8-256 regs, 32-1024 elems
- Fuj. VP300 (1996) 100 MHz 8-256 regs, 32-1024 elems
- NEC SX/2 (1984) 160 MHz 8+8K regs, 256+var elems
- NEC SX/3 (1995) 400 MHz 8+8K regs, 256+var elems

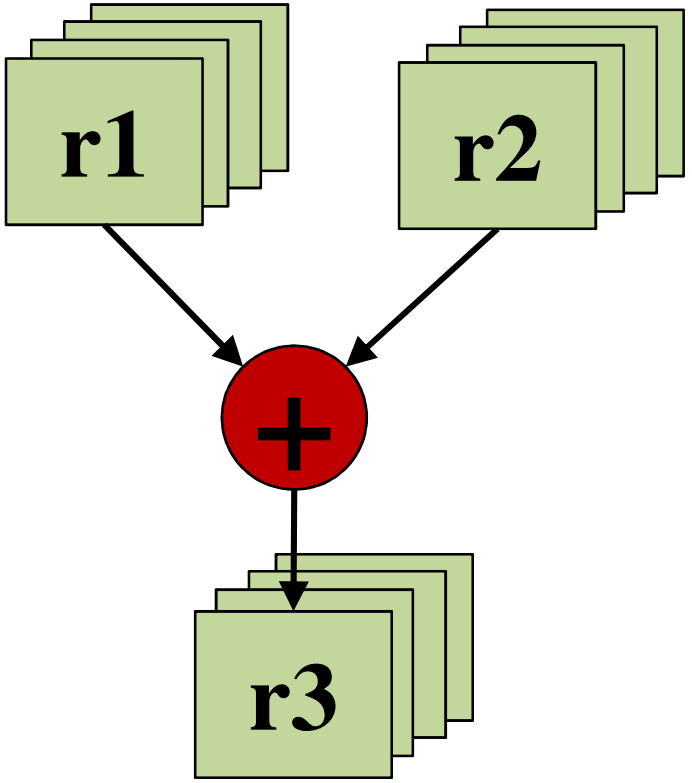
# Векторные инструкции

**Scalar instruction**



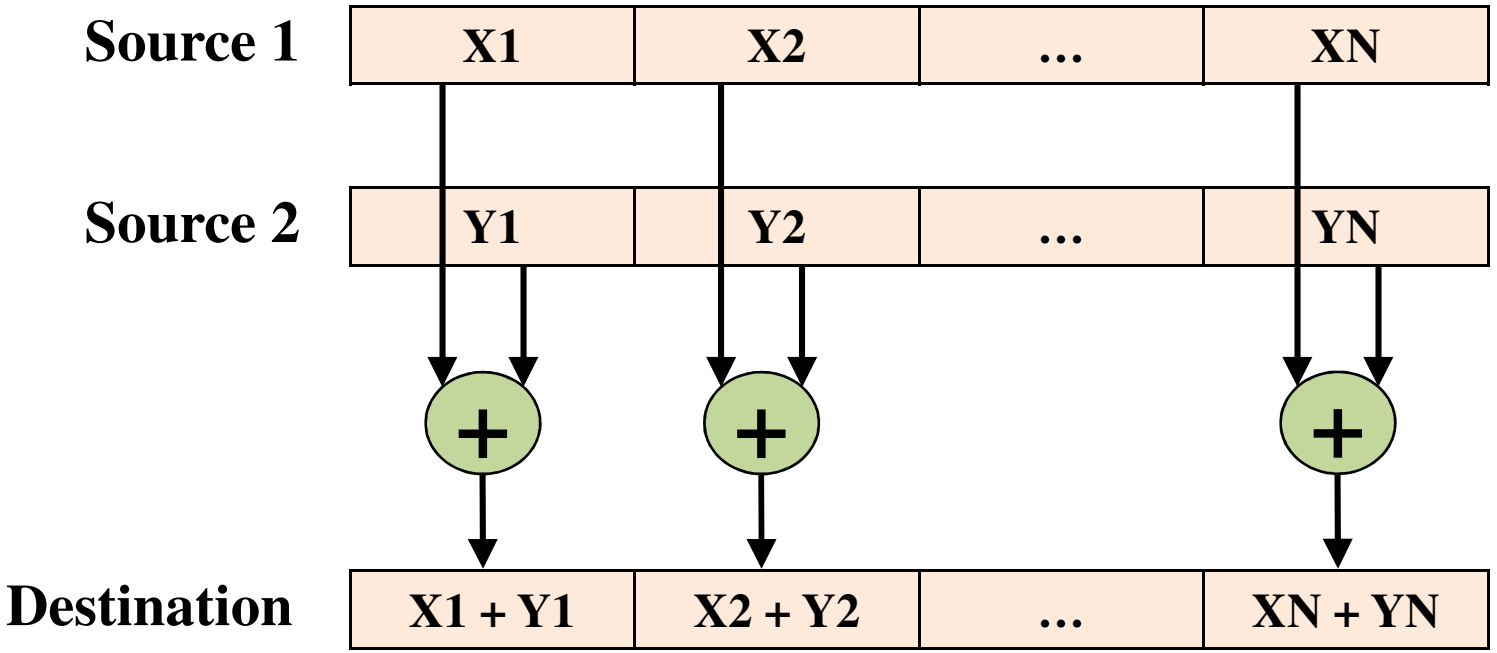

```
add r3, r1, r2
```

**Vector instruction**




```
addv r3, r1, r2
```

# Векторные инструкции




**SIMD** – Single Instruction Stream,
Multiple Data Stream

# Современные процессоры

**SIMD-инструкции в составе наборов команд**

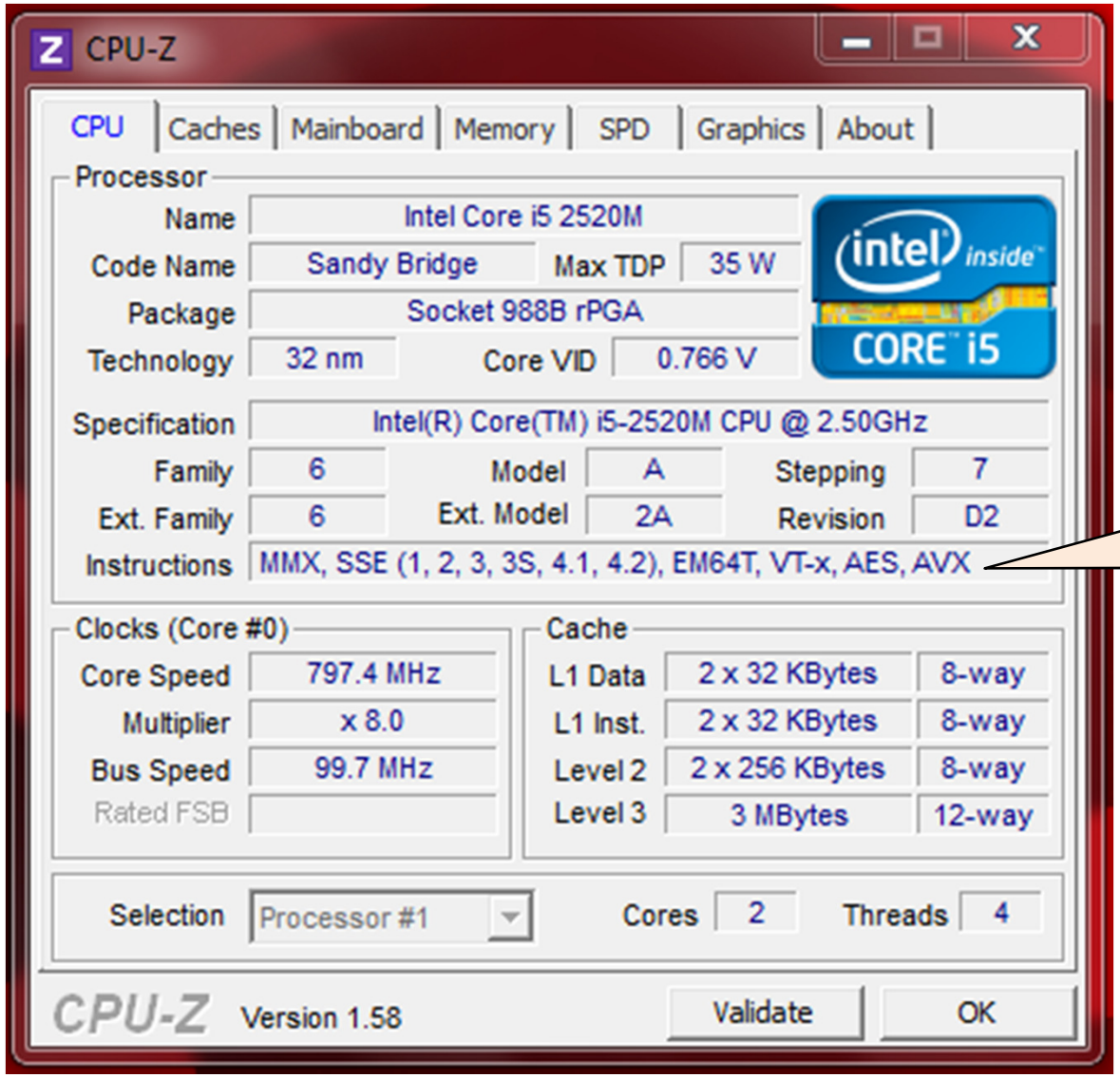
- Intel MMX (1996)
- Motorola PowerPC, IBM POWER AltiVec (1999)
- AMD 3DNow! (1998)
- Intel SSE (Intel Pentium III, 1999)
- Intel SSE2, SSE3, SSE4
- AVX (Advanced Vector Extension, Intel & AMD, 2008)

# Поддерживаемые наборы команд

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts nopl xtopology nonstop_tsc
aperfmperf pni pclmulqdq dtes64 ds_cpl vmx smx est tm2
ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt
tsc_deadline_timer xsave avx lahf_lm ida arat epb xsaveopt
pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
```

# Поддерживаемые наборы команд

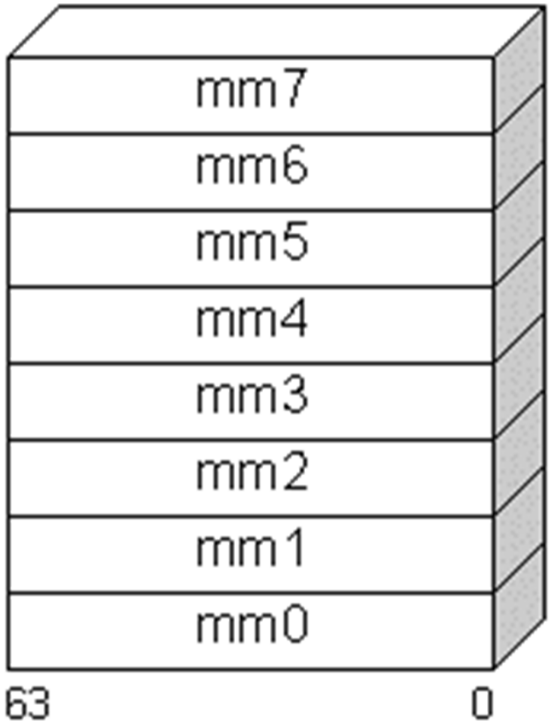
## Windows CPU-Z




MMX, …, VT-x, AES, AVX

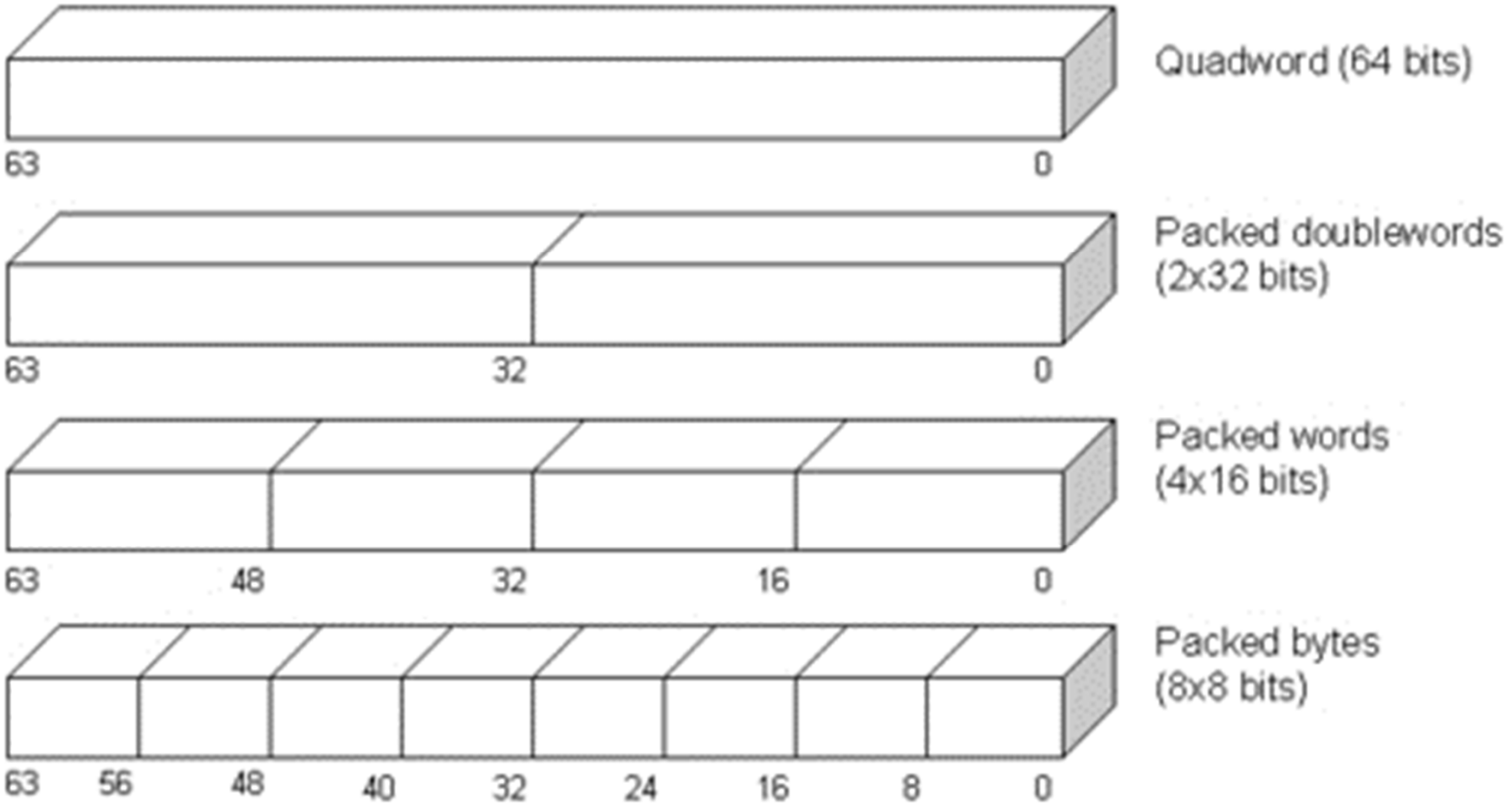
# Intel MMX

- MMX – инструкции обработки целочисленных векторов длиной 64 бит.




**Набор регистров MMX**

# Intel MMX


Quadword (64 bits)
63
0
Packed doublewords
(2x32 bits)
63
32
0
Packed words
(4x16 bits)
63
48
32
16
0
Packed bytes
(8x8 bits)
63
56
48
40
32
24
16
8
0

# Intel MMX

```
for (i = 0; i < 8; i++) {
    B[i] = B[i] + A[i]
}
```

```
    mov esi, offset A
    mov edi, offset B
    mov ecx, 8
L1: mov al, [esi]
    add [edi], al
    inc esi
    inc edi
    loop L1
```

# Intel MMX

```
movq mmreg1, A
movq mmreg2, B
paddb mmreg2, mmreg1
movq B, mmreg2
```

# Intel SSE (Streaming SIMD Extensions)

- SSE (1999, Pentium III) – 70 инструкций (floating point data)
- SSE2 (2001, Pentium 4, AMD64)
- SSE3 (2004) – 144 инструкций
- SSSE3 (2006, Intel Core) – 16 новых инструкций
- SSE4 (2006-2007, Intel Core) – 54 инструкций
  - SSE4.1 – подмножество из 47 инструкций
  - SSE4.2 – подмножество из 7 инструкций
- SSE4a (2008, AMD Barcelona) – 4 инструкции
- AVX (2008, Intel Sandy Bridge, AMD Bulldozer)

# SIMD Instruction Sets (Cont.)

- PowerPC AltiVec (POWER7, IBM Cell)
- ARM Cortex-A8, Cortex-A9 (Apple iPhone 4S, Google Galaxy Nexus: NEON module – Advanced SIMD)

# SSE (Intel 64, EM64T)

**SSE**
**128 бит**

| xmm0 |
|---|
| xmm1 |
| xmm2 |
| xmm3 |
| xmm4 |
| xmm5 |
| xmm6 |
| xmm7 |

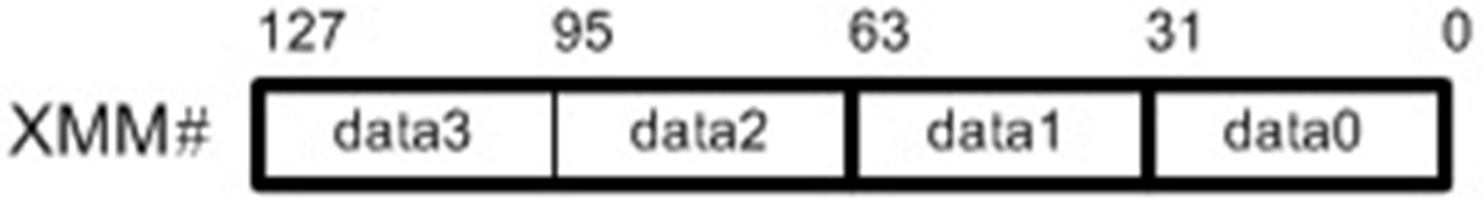
4 вещественных числа
одинарной точности
(float, 32 bit)

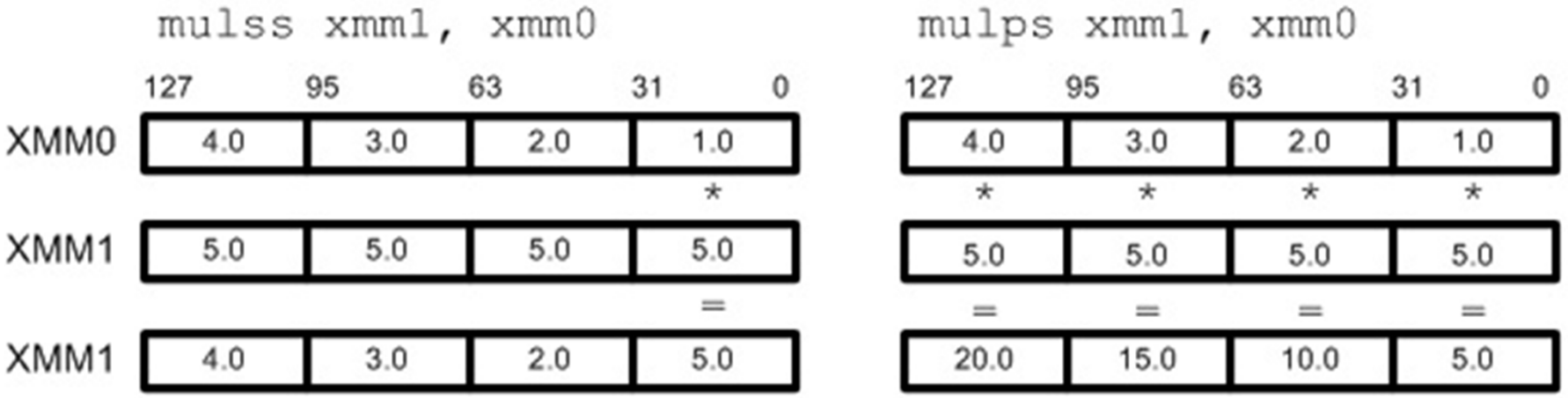
**SSE2, …, SSE4**
**128 бит**

| xmm0 |
|---|
| xmm1 |
| xmm2 |
| xmm3 |
| xmm4 |
| xmm5 |
| xmm6 |
| xmm7 |
| xmm8 |
| xmm9 |
| … |
| xmm15 |

# Типы инструкций

- Операции над скалярными величинами – суффикс SS
- Операции над векторами (упакованными скалярами) – суффикс PS



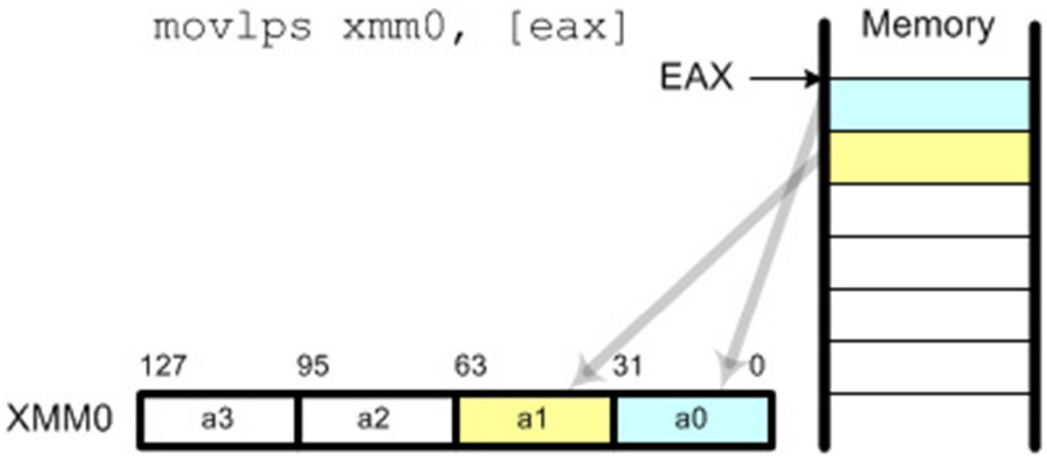
# SSE: Floating point & Integer instructions

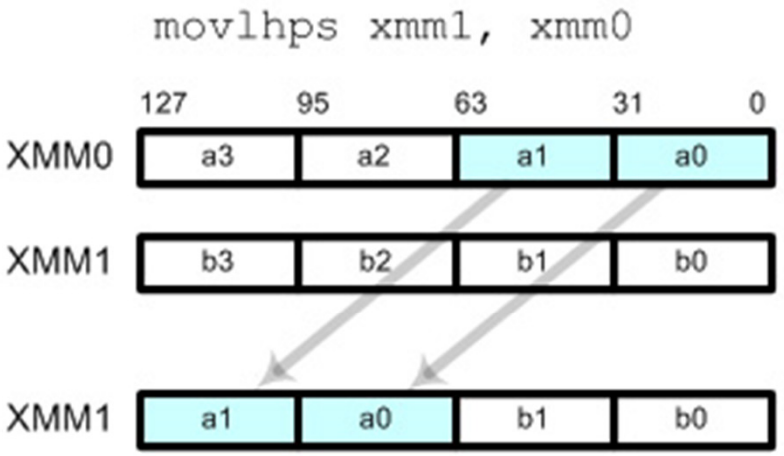
- Копирование данных (mem-reg/reg-mem/reg-reg)
  - `MOVSS`
  - `MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS`
- Арифметические
  - `ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS`
  - `ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS`
- Операции сравнения
  - `CMPSS, COMISS, UCOMISS`
  - `CMPPS`
- Поразрядные логические операции
  - `ANDPS, ORPS, XORPS, ANDNPS`

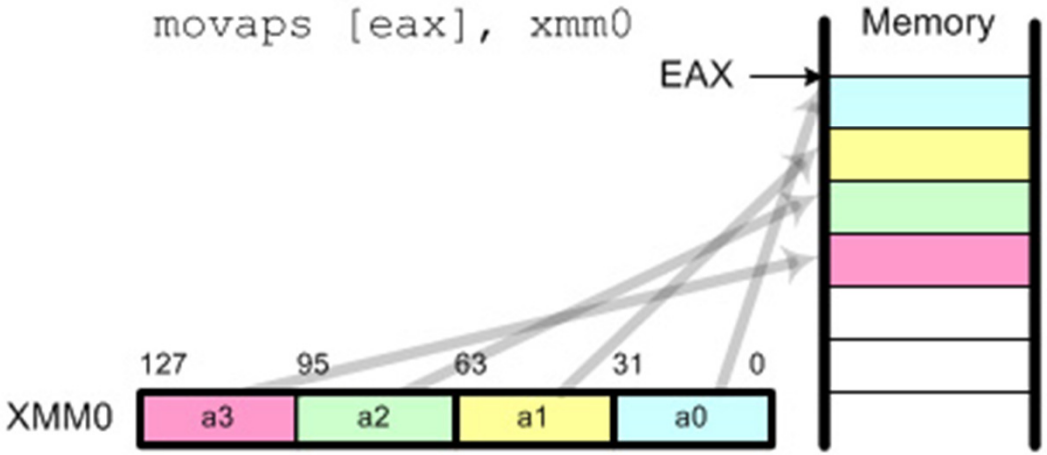
# SSE: Other instructions

- Управление оперативной и кеш-памятью
  - `MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE`

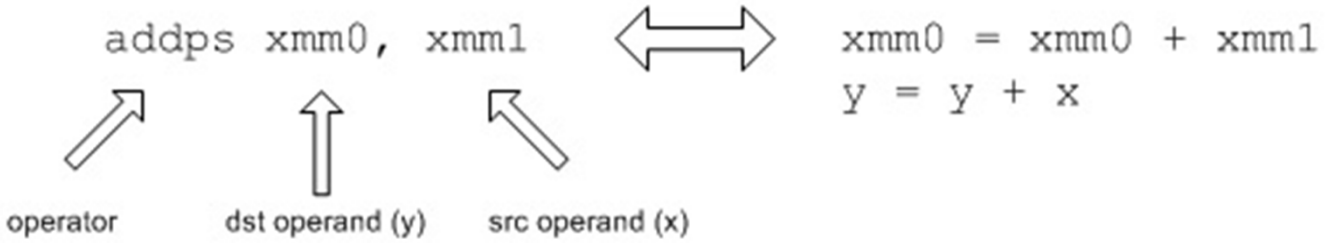
# Копирование данных



# Арифметические операции




| 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 |

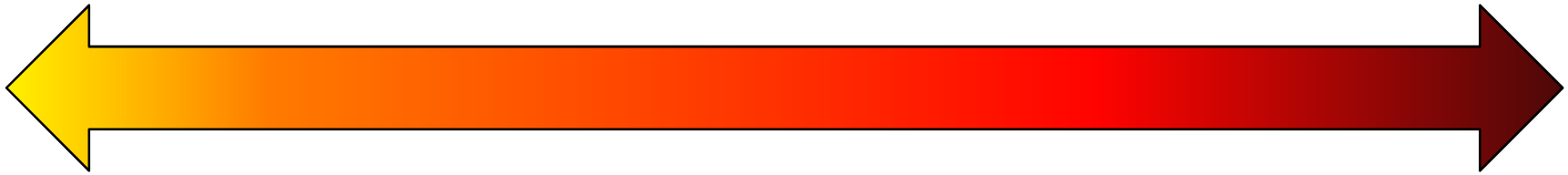
# Использование инструкций SSE

**Ассемблер**

**Встроенные функции компилятора (Intrinsic)**

**C++ классы**

**Автоматическая векторизация компилятора**



Лучшая управляемость

Простота использования

# Автовекторизация компилятором

```
#include <stdio.h>

#define N 100

int main()
{
    int i, a[N];

    for (i = 0; i < N; i++) {
        a[i] = a[i] >> 2;
    }
    return 0;
}
```

# Автовекторизация компилятором GCC

```
$ gcc -O2 -ftree-vectorize \
      -ftree-vectorizer-verbose=2 -msse2
      -S ./prog.c
```

```
...
.L2:
  movdqa   (%rax), %xmm0
  psrad    $2, %xmm0
  movdqa     %xmm0, (%rax)
  addq     $16, %rax
  cmpq     %rbp, %rax
  jne       .L2
  movq     %rsp, %rbx
...
```

# Автовекторизация компилятором Intel

```
$ icc -xP -o prog ./prog.c
```

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

# C++ классы SSE (Intel Compiler only)

```
void add(float *a, float *b, float *c)
{
    int i;

    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

# C++ классы SSE (Intel Compiler only)

```
#include "fvec.h"

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.

**Intel C++ Class Libraries for SIMD Operations User's Guide**

# SSE Intrinsics (builtin functions)

- **Intrinsics –** набор встроенных функций и типов данных, поддерживаемых компилятором, для предоставления высокоуровневого доступа к SSE-инструкциям.
- Компилятор распределяет XMM регистры и выравнивает адреса доступа к данным.
- **Заголовочные файлы:**
  - `#include <mmintrin.h>  /* MMX */`
  - `#include <xmmintrin.h>  /* SSE */`
  - `#include <emmintrin.h>  /* SSE2 */`
  - `#include <pmmintrin.h>  /* SSE3 */`
  - `#include <smmintrin.h>  /* SSE4 */`
  - `#include <immintrin.h>  /* AVX */`

# SSE Intrinsics: типы данных

- `__m128  /* float[4] */`
- `__m128d  /* double[2] */`
- `__m128i  /* integer: byte[8], int[4] */`
- `__m64  /* MMX integer SIMD */`

# Выравнивание памяти: Microsoft Windows

- **Выравнивание памяти**

  Хранимые в памяти операнды SSE-инструкций должны быть размещены по адресу выровненному на границу в 16 байт.

**Microsoft Windows (Visual C++)**

- Статические массивы

```
__declspec(align(16)) float A[N];
```

- Динамические массивы

```
A = (float *)_aligned_malloc(N * sizeof(float), 16);
_aligned_free(A);
```

# Выравнивание памяти: GNU/Linux

- Статические массивы

```
float A[N] __attribute__((aligned(16)));
```

- Динамические массивы

```
A = (float *)_mm_malloc(N * sizeof(float), 16);
_mm_free(A);
```

```
int posix_memalign(void **memptr,
                   size_t alignment, size_t size);
```

# SSE Intrinsics

```
void add(float *a, float *b, float *c)
{
    int i;

    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

# SSE Intrinsics

```
#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);
}
```

# SSE Intrinsics

```
/* gcc -O2 -S ./intr.c */

add:
.LFB517:
    .cfi_startproc
    movaps   (%rdi), %xmm0
    addps    (%rsi), %xmm0
    movaps   %xmm0, (%rdx)
    ret
    .cfi_endproc
```

# Вставки на ассемблере

```
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"
    );
}
```

# Вставки на ассемблере

```
/* gcc -O2 -S ./add_sse_asm.c */

add_sse_asm:
.LFB11:
    .cfi_startproc
    movaps (%rdi), %xmm0
    movaps (%rsi), %xmm1
    addps %xmm1, %xmm0
    movaps %xmm0, (%rdx)
    ret
    .cfi_endproc
```

# SSE Intrinsics

```
void fun(float *a, float *b, float *c,
         int n)
{
    int i;

    for (i = 0; i < n; i++) {
        c[i] = sqrt(a[i] * a[i] +
                    b[i] * b[i]) + 0.5f;
    }
}
```

# SSE Intrinsics

```
void fun(float *a, float *b, float *c, int n)
{
    int i, k;
    __m128 x, y, z;
    __m128 *aa = (__m128 *)a;
    __m128 *bb = (__m128 *)b;
    __m128 *cc = (__m128 *)c;

    k = n / 4;
    z = _mm_set_ps1(0.5f);
    for (i = 0; i < k; i++) {
        x = _mm_mul_ps(*aa, *aa);
        y = _mm_mul_ps(*bb, *bb);
        x = _mm_add_ps(x, y);
        x = _mm_sqrt_ps(x);
        *cc = _mm_add_ps(x, z);
        aa++;
        bb++;
        cc++;
    }
}
```

# SSE Intrinsics

| Intrinsic | Operation |
|---|---|
| `_mm_add_ss` | Addition |
| `_mm_add_ps` | Addition |
| `_mm_sub_ss` | Subtraction |
| `_mm_sub_ps` | Subtraction |
| `_mm_mul_ss` | Multiplication |
| `_mm_mul_ps` | Multiplication |
| `_mm_div_ss` | Division |
| `_mm_div_ps` | Division |
| `_mm_sqrt_ss` | Squared Root |
| `_mm_sqrt_ps` | Squared Root |
| `_mm_rcp_ss` | Reciprocal |
| `_mm_rcp_ps` | Reciprocal |
| `_mm_rsqrt_ss` | Reciprocal Squared Root |
| `_mm_rsqrt_ps` | Reciprocal Squared Root |
| `_mm_min_ss` | Computes Minimum |
| `_mm_min_ps` | Computes Minimum |
| `_mm_max_ss` | Computes Maximum |
| `_mm_max_ps` | Computes Maximum |

# AVX – Advanced Vector Extensions

- Размер векторов увеличен до 256 бит (YMM0, …, YMM15)
- Существующие операции работают над младшей частью регистров YMM#
- Трехоперандный синтаксис: c = a + b

- Использование YMM регистров требует поддержки со стороны операционной системы
  - Linux ядра >= 2.6.30
  - Apple OS X 10.6.8
  - Windows 7 (поддержка добавлена в SP 1)

- Поддержка компиляторами
- GCC 4.6
- Intel C++ Compiler 11.1
- Microsoft Visual Studio 2010
- Open64 4.5.1
- PathScale

# AVX – Advanced Vector Extensions

- Типы данных: `__m256, __m256d, __m256i`

# AVX – Advanced Vector Extensions

```
void fun_avx(float *a, float *b, float *c, int n)
{
    int i, k;
    __m256 x, y;
    __m256 *aa = (__m256 *)a;
    __m256 *bb = (__m256 *)b;
    __m256 *cc = (__m256 *)c;

    k = n / 8;

    for (i = 0; i < k; i++) {
        x = _mm256_mul_ps(*aa, *aa);
        y = _mm256_mul_ps(*bb, *bb);
        x = _mm256_add_ps(x, y);
        *cc = _mm256_sqrt_ps(x);
        aa++;
        bb++;
        cc++;
    }
}
```

# ARM NEON SIMD engine

- Android NDK (C intrinsics)
- Apple iOS (GCC assembler)
- ARM NEON: 16 векторных регистров шириной 128 бит: q0, q1, …, q15

# ARM NEON SIMD engine

```
#include <arm_neon.h>

int sum_array(int16_t *array, int size)
{
    /* Init the accumulator vector to zero */
    int16x4_t vec, acc = vdup_n_s16(0);
    int32x2_t acc1;
    int64x1_t acc2;

    for (; size != 0; size -= 4) {
        /* Load 4 values in parallel */
        vec = vld1_s16(array);
        array += 4;
        /* Add the vec to the accum vector */
        acc = vadd_s16(acc, vec);
    }
    acc1 = vpaddl_s16(acc);
    acc2 = vpaddl_s32(acc1);
    return (int)vget_lane_s64(acc2, 0);
}
```

# ARM GCC (NEON)

```
void multiplyBy2(int *p, int count)
{
    for (int i = 0; i < count; ++i) {
        *p++ <<= 1;
    }
}
```

# ARM GCC (NEON)

```
#include <arm_neon.h>

void multiplyBy2Neon(int* p, int count)
{
    assert(count % 4 == 0);
    int n = n / 4;

    for (int i = 0; i < n; ++i) {
        // Load 4 ints into a quadword register
        int32x4_t in = vld1q_s32(p);
        // Shift each int left by 1
        in = vshlq_n_s32(in, 1);
        // Store register back into memory
        vst1q_s32(p, in);
        p += 4;
    }
}
```