

Низкоуровневое программирование

Лекция 2

Вычисления с плавающей запятой

Векторные вычисления

Числа с плавающей запятой

Формат чисел с плавающей запятой определяется стандартом IEEE-754.

Первоначально, стандарт определял два основных типа чисел – числа одинарной точности и числа двойной точности размером 32 и 64 бит соответственно.

Помимо формата чисел, стандарт определяет также правила округления и сравнения, реакцию на запрещенные операции.



$$x = (-1)^s \cdot 1, m_1 m_2 m_3 \dots m_M \cdot 2^{exp - bias}$$

Точность числа	M	bias
Одинарная	23	127
Двойная	52	1023

Специальные значения (пример)

Помимо самого порядка чисел, формат определяет также специальные значения: NaN и $\pm \infty$.

NaN (Not a Number) – это результат запрещенной операции: деления на 0, ситуации неопределенности (0/0, Inf/Inf, Inf*0) и пр. Данное число не равно никакому другому числу, а результат любой операции над NaN равен NaN.

Различают сигнальные NaN (**sNaN**), и “тихие” NaN (quiet NaN, **qNaN**). Возникновение сигнального NaN возбуждает аппаратное исключение.

Тип числа	Экспонента	Мантисса
0	00...00 ₂	00...00 ₂
Денормализованное	00...00 ₂	00...01 ₂ ... 11...11 ₂
Нормализованное	00...01 ₂ ... 11...10 ₂	00...00 ₂ ... 11...11 ₂
Бесконечность	11...11 ₂	00...00 ₂
sNaN	11...11 ₂	00...01 ₂ ... 01...11 ₂
qNaN	11...11 ₂	10...00 ₂ ... 11...11 ₂

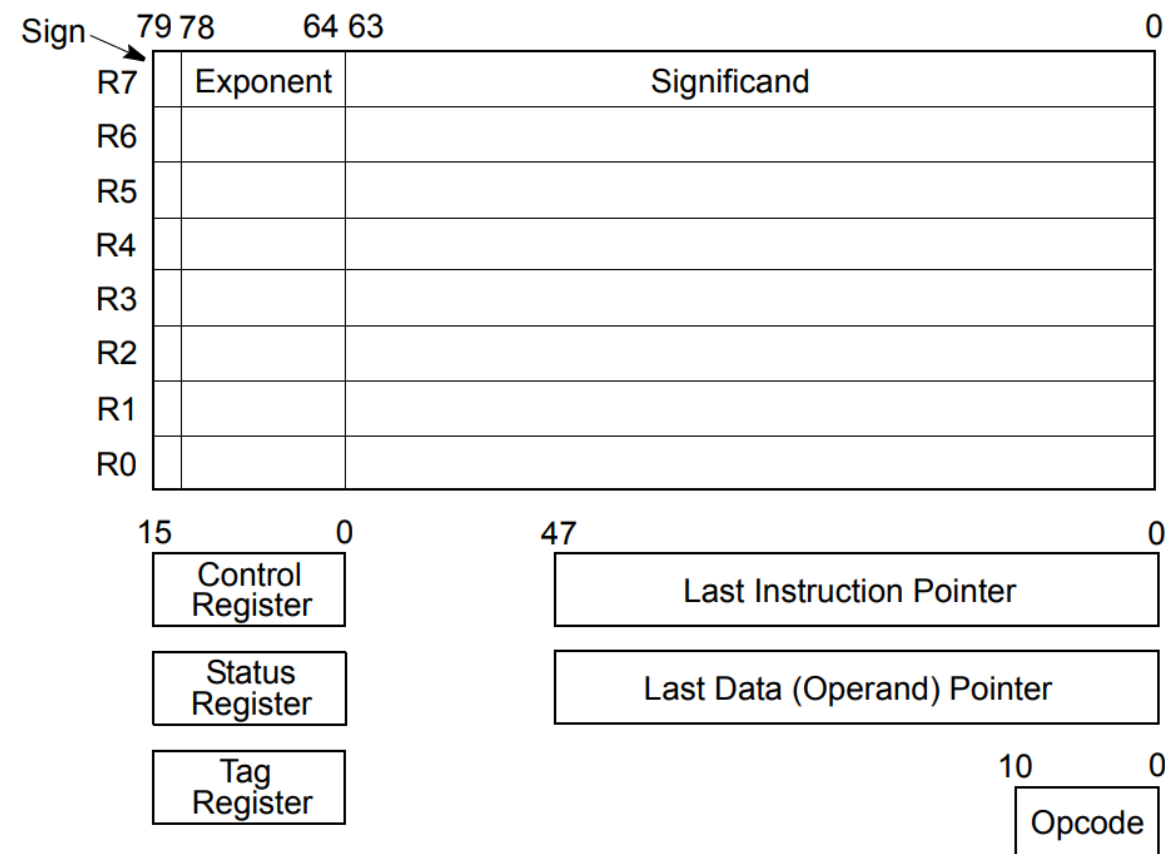
Сопроцессор x87



Процессор Intel 8086 не мог самостоятельно выполнять вычисления с плавающей запятой – для этого требовался отдельный сопроцессор (**FPU**, **f**loating **p**oint **u**nit).

Выпускаемый Intel сопроцессор имел маркировку 8087. В отсутствие сопроцессора операции с плавающей точкой программно эмулировались, что влекло огромную потерю производительности.

Набор регистров сопроцессора состоит из **8 регистров данных, регистра флагов, регистра состояния, регистра тэгов**, а также 3 служебных регистров, хранящих последний опкод, указатель на последние инструкцию и данные.



Инициализация и сохранение состояния сопроцессора

Для реинициализации состояния сопроцессора (сброса всех значений и состояния), используется инструкция `fninit`. По умолчанию после реинициализации все исключения маскированы (см. далее).

Сохранение полного состояния сопроцессора (108 байт) осуществляется инструкцией `fsave`.

Загрузка состояния осуществляется инструкцией `frstor`.

Типы данных FPU

По умолчанию, операции в FPU осуществляются с **расширенной точностью** – 80 бит вместо 64. Соответствующий тип в языке C – `long double` (игнорируется некоторыми компиляторами).

Можно переключить FPU в режим вычислений с одинарной и двойной точности для строгого соответствия с IEEE-754 (см. поле PC регистра управления).

Операции с целыми числами FPU не осуществляет (хотя мантисса занимает 64 бита – достаточно для точного представления `long long`).

Приведение типов и расширение/уменьшение точности производятся **автоматически** при загрузке/выгрузке данных с FPU.

Стек FPU

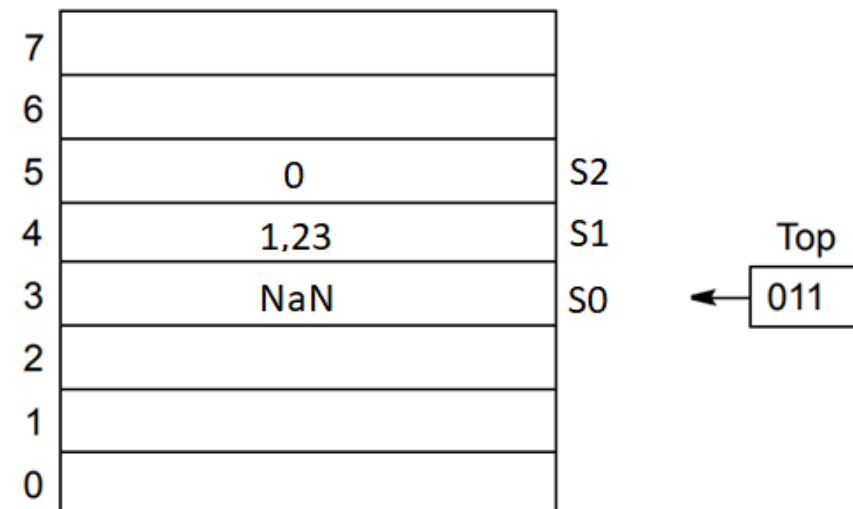
Регистры данных не являются независимыми друг от друга – они организованы в **стек**. Номер регистра, являющегося текущей вершиной стека, хранится в поле TOP регистра состояния.

Состояние каждого регистра хранится в **регистре тегов**.

Загрузка данных и их выгрузка из стека производятся с вершины стека.

Адресация регистров осуществляется относительно текущей вершины: **ST0** – вершина стека, **ST1** – след. элемент стека и т.д.

Попытка доступа к регистру, который не обозначен, как занятый, ведет к аппаратному исключению (см. далее).



TAG(7)	TAG(6)	TAG(5)	TAG(4)	TAG(3)	TAG(2)	TAG(1)	TAG(0)
11	11	01	00	10	11	11	11

Стек FPU растет вниз!

Загрузка/выгрузка вещественных чисел

Помещение данных в стек осуществляется инструкцией `fld` (**FPU load**).

Выгрузка данных производится инструкциями `fst`/`fstp`.

Инструкция `fstp` (**FPU store + pop**) дополнительно удаляет значение из вершины стека.

Тип загружаемого значения определяется операндом (`dword` – float, `qword` – double, `tbyte` – long double).

Инструкция `fst` может также дублировать данные из одного регистра в другой.

Для обмена данных между регистрами используется инструкция `fxch`.

```
fld qword[rbx]
```

TOP--; (стек растёт вниз)
ST0 = *(double*)RBX;

```
fld st5
```

TOP--;
ST0 = ST5;

```
fst dword[rbx]
```

(float)RBX=(float)ST0;
TOP++;

```
fstp st5
```

ST5=ST0
TOP++;

```
fxch st1, st2
```

swap(ST1 , ST2)

Приведение целых чисел к вещественным

Загрузка целых чисел осуществляется инструкцией `fild`. Данная инструкция также принимает 1 операнд – адрес в памяти. Операнд автоматически приводится к числу расширенной точности.

Инструкции `fist/fistp` выполняют выгрузку значения с приведением к целому.

Режим округления (вверх, вниз, до ближайшего, к 0) определяется кодом режима округления в управляющем регистре (поле RC, см. далее).

```
fild dword[rbx]    TOP--;  
                   ST0 = *(int*)RBX;
```

```
fild qword[rbx]    TOP--;  
                   ST0 = *(long long*)RBX;
```

```
fist dword[rbx]    *(dword*)RBX=(int)ST0 ;
```

```
fistp dword[rbx]   *(dword*)RBX=(int)ST0 ;  
                   TOP++;
```

Загрузка констант

Для загрузки констант используются специальные инструкции.

FLD1	TOP--; ST0 = 1.0;
FLDL2T	TOP--; ST0 = $\log_2 10$;
FLDL2E	TOP--; ST0 = $\log_2 e$;
FLDPI	TOP--; ST0 = π ;
FLDLG2	TOP--; ST0 = $\log_{10} 2$;
FLDLN2	TOP--; ST0 = $\ln 2$;
FLDZ	TOP--; ST0 = +0.0;

Арифметические операции

Арифметические операции выполняются инструкциями `fadd/fsub/fsubr/fmul/fdiv/fdivr`. Данные инструкции принимают от 0 до 2 операндов. При этом, если операнды являются регистрами – то **один из них обязан быть ST0**.

Инструкции `faddp/fsubp/fsubrp/fmulp/fdivp/fdivrp` действуют аналогично обычным арифметическим операциям, однако они **дополнительно выталкивают значение с вершины стека**. Операндами могут быть только регистры сопроцессора.

Инструкции без операндов (например, `fadd` и `faddp`) являются синонимами в языке ассемблера NASM и всегда 1)используют ST1 и ST0 в качестве операндов 2) выталкивают значение с вершины.

Арифметические операции

fadd ST1 += ST0; TOP++

faddp ST1 += ST0; TOP++;

fsub tbyte[rbx] ST0 -= *(long double*)RBX

fmulp st5 ST5 *= ST0; TOP++;

fmul st5 ST0 *= ST5;

fdivp st1, st0 ST1 /= ST0; TOP++;

fdiv st1, st0 ST1 /= ST0;

fsub ST1-=ST0; TOP++;

fadd st0, st7 ST0 += ST7

fsubr ST1=ST0-ST1; TOP++;

fdiv st0, st1 ST0 /=ST1

fdiv**r** st0, st1 ST0 = ST1/ST0;

Тригонометрические функции, квадратный корень и логарифм

`fsin` `ST0 = sin(ST0)`

`fsqrt` `ST0 = sqrt(ST0)`

`fcos` `ST0 = cos(ST0)`

`fptan` `TOP--;`
 `ST1 = tan(ST0);`
 `ST0 = 1`

`fy12x` `ST1 = ST1*log2 ST0;`
 `TOP++;`

`fpatan` `ST1 = arctan(ST1/ST0);`
 `TOP++;`

`fsincos` `s = sin(ST0); c = cos(ST0);`
 `TOP--;`
 `ST0 = c; ST1 = s;`

Остаток от деления (пример)

Вещественный остаток от деления вычисляется инструкциями `fprem` и `fprem1` (Partial REMainder).

`fprem1` функционирует аналогично [`std::remainder\(\)`](#).

`fprem` функционирует аналогично [`std::fmod\(\)`](#).

Разница в значениях ST0 и ST1 не должна превышать 2^{63} , иначе результатом инструкции является частичный остаток (при этом флаг C2 регистра состояния равен 1). Как следствие, *остаток от деления должен вычисляться в цикле*, если нет гарантии, что данное условие выполняется.

Д/З: отличие `std::remainder()` от `std::fmod()`

Возведение в степень (см. раздел 2.10 указаний к ЛР2)

Для возведения в дробную степень 2 используется инструкция `f2xm1`. Данная инструкция требует $|ST0| < 1$.

`f2xm1`

$$ST0 = 2^{ST0} - 1, |ST0| < 1$$

Для возведения в целую степень 2 используется инструкция `fscale`. *Данная инструкция игнорирует дробную часть числа.*

`fscale`

$$ST0 = ST0 * 2^{[ST1]}$$

Отделить целую и дробную части можно инструкцией `fprem`.

Сравнение

Для установки флагов регистра FLAGS
используются инструкции
`fcomi/fcomip/fucomu/fucomip`.

Инструкция `fcomip` после выполнения
сравнения выталкивает значение из
вершины стека.

Отличие `fcomi` и `fucomi` в то, что первая
инструкция генерирует исключение при
попытке сравнения NaN, а вторая - нет

`fcomi`

`compare(ST0, ST1)`

`fcomip st3`

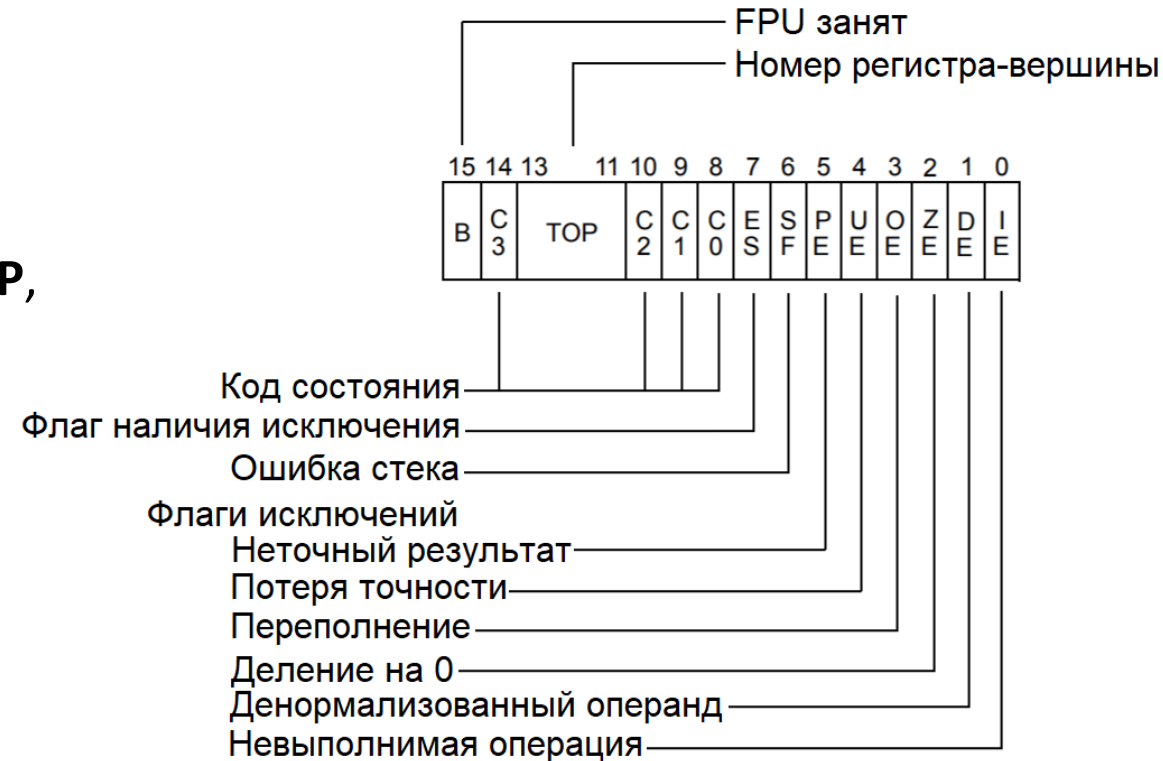
`compare(ST0, ST3)`
TOP++

Результат сравнения	ZF	PF	CF
ST0 > ARG	0	0	0
ST0 < ARG	0	0	1
ST0 == ARG	1	0	0
ST0 is NaN ARG is NaN	1	1	1

Регистр состояния

Регистр состояния содержит слово состояния сопроцессора. Отдельные биты слова являются флагами. Среди значимых флагов находятся **С0-С3**, которые хранят результаты сравнения, а в случае аппаратного исключения – уточняющие данные; **ТОР**, являющийся указателем на текущую вершину стека. Флаги 0-7 являются флагами исключений.

Выгрузка слова состояния (в ОЗУ или в регистр АХ) осуществляется инструкцией `fstsw`. Затем слово состояния анализируется обычными инструкциями ЦП.



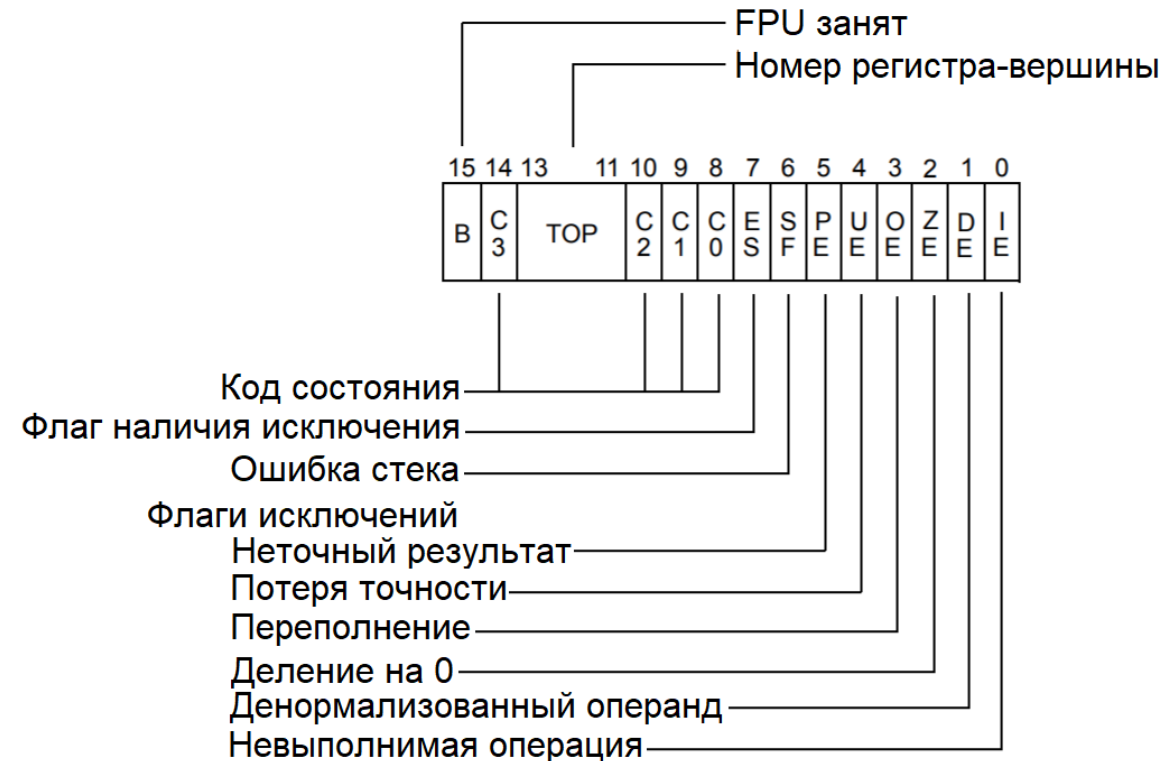
Исключения сопроцессора (пример)

В случае появления ошибки при вычислениях сопроцессор сигнализирует об этом процессору, который бросает *аппаратное* исключение .

Причину исключения можно узнать, анализируя соответствующие флаги регистра состояния.

Как аппаратные исключения обрабатываются — будет рассмотрено позже. Обычно аппаратное исключение приводит к аварийному завершению программы.

Исторически, аппаратные исключения генерируются при начале выполнения инструкции, следующей за «виновной». Для проверки наличия немаскированных исключений используется инструкция `fwait`.



Управляющий регистр и маска исключений

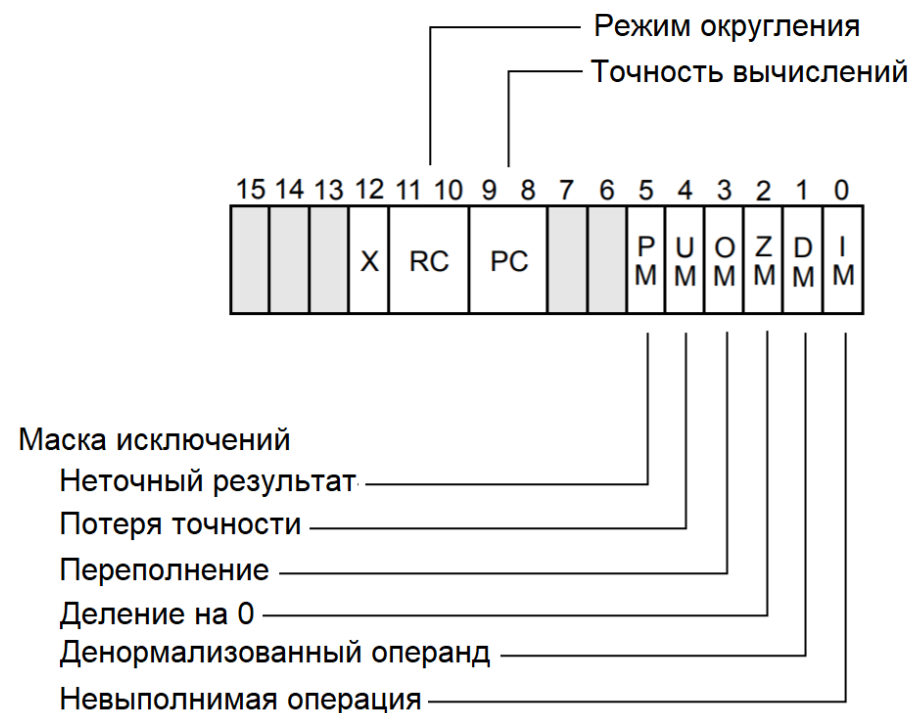
(см. пример и приложение Б ЛР2)

Исключения определенного типа могут быть явно запрещены (маскированы) путем установки флагов **управляющего регистра**.

В этом случае аппаратные исключения не будут возникать, но биты в слове состояния будут выставляться.

В управляющем регистре можно также выбрать **точность вычислений** и **метод округления**.

Для того, чтобы изменить флаги, слово управления следует выгрузить в память инструкцией `fstcw`, изменить его, и загрузить обратно инструкцией `fldcw`.



Округление (см. приложение А ЛР2)

Поскольку в ходе вычислений с плавающей запятой неизбежно возникают автоматические округления в младшем разряде, а также явные округления до целого числа, в стандарте IEEE-754 определены 4 возможных режима округления.

Код режима указывается в поле RC управляющего регистра.



Режим округления	Код
К ближайшему целому	00_2
Вниз	01_2
Вверх	10_2
К нулю	11_2

Набор инструкций SSE

Хотя математический сопроцессор есть во всех современных x86-процессорах, вычисления с плавающей запятой используется более простой и современных подход.

В 1999 году был внедрен набор инструкций **SSE**. В настоящее время большинство процессоров x86 как минимум имеют набор SSE3.

SSE – это набор инструкций для векторных вычислений. Тем не менее, его можно использовать и для обычных вычислений с плавающей запятой.

В процессоре с поддержкой SSE3 есть 16 регистров **XMM0-XMM15**, которые независимы друг от друга (подобно RAX, RBX, R8 и др.).

XMM0			
float	float	float	float
double		double	

Загрузка/выгрузка данных

Для загрузки данных в регистр и выгрузки данных из него используются инструкции `movss` (**m**ove **s**calar **s**ingle) и `movsd` (**m**ove **s**calar **d**ouble), загружающие в регистр число одинарной или двойной точности из памяти или другого регистра XMM.

Автоматического приведения типов в этом случае не происходит.

`movss xmm0, [rbx]` `XMM0 = *(float*)RBX`

`movsd [rbx], xmm0` `*(double*)RBX = XMM0`

`movss xmm0, xmm1` `XMM0 = XMM1`

Приведение типов

Приведение между целыми числами и числами с плавающей запятой осуществляется инструкциями `cvtss2si/cvtsd2si` и `cvtsi2ss/cvtsi2sd`. Режим округления задается в регистре `MXCSR`.

Приведение `float` к `double` осуществляется инструкцией `cvtss2sd`. Приемником всегда должен быть XMM-регистр. Обратное преобразование осуществляется инструкцией `cvtsd2ss`.

Приемником для `cvtss2si/cvtsd2si` может быть только **регистр общего назначения**, а для `cvtsi2ss/cvtsi2sd/cvtss2sd/cvtsd2ss` – только **XMM-регистр**

```
cvtsi2ss xmm0, dword[rbx]
```

`XMM0 = (float)*(int*)RBX`

```
cvtsd2si rax, xmm0
```

`RAX = (long long)XMM0`

```
cvtss2sd xmm0, [rbx]
```

`XMM0 = (double)*(float*)RBX`

```
cvtsd2ss xmm0, xmm1
```

`XMM0 = (float)XMM1`

Математические операции

Математические операции выполняются инструкциями

`addsx/subsx/divsx/mulsx/sqrtsx/rcpsx/rsqrtsx`, где X – s или d.

Смысл первых 5 инструкций очевиден из названия. Инструкция `rcpsX` вычисляет $1/\text{arg}$. Инструкция `rsqrtX` вычисляет $1/\sqrt{\text{arg}}$.

```
addss xmm0, [rbx]
```

`XMM0 += *(float*)RBX`

```
subsd xmm1, xmm2
```

`XXMM1 -= XMM2;`

Сравнение

Сравнение чисел с плавающей точкой осуществляется инструкциями `cmp \mathbf{eq} sX` (`==`), `cmp \mathbf{lt} sX` (`<`), `cmp \mathbf{le} sX` (`<=`), `cmp \mathbf{unord} sX`, `cmp \mathbf{neq} sX` (`!=`), `cmp \mathbf{nlt} sX` (`>=`), `cmp \mathbf{nle} sX` (`>`), `cmp \mathbf{ord} sX` (оба результата не-NaN и не-бесконечность).

Результат сравнения записывается в регистр-приемник. Если сравнение вычисляется в `False`, приемник = 0, если в `True`, то приемник = `0xFF..FF`.

`cmp \mathbf{eq} ss XMM1, XMM2` `XMM1 = XMM1==XMM2 ? 0xF...F : 0`

`cmp \mathbf{nlt} sd XMM1, [rbx]` `XMM1 = XMM1>*(double*)RBX ? 0xF...F : 0`

Если сравнение необходимо для дальнейшего выполнения условного перехода, то следует использовать инструкции `comisX` или `ucomisX`. Инструкция `ucomisX` используется, если NaN является допустимым значением.

Инструкции сравнивают операнды и выставляют флаги `ZF`, `CF` и `PF` регистра `FLAGS`. (флаги те же, что для `FPU`).

Выбор минимума/максимума

Иногда сравнение нужно только для определения наибольшего из 2 чисел. Вместо сравнения в комбинации с условным переходом, можно использовать инструкции `minsx/maxsx`.

```
minss XMM1, XMM2
```

`XMM1 = XMM1 < XMM2 ? XMM1 : XMM2`

```
maxsd XMM1, [rbx]
```

`XMM1 = XMM1 > *RBX ? XMM1 : *RBX`

Регистр состояния и исключения (пример)

При возникновении ошибок во время вычислений, возникает аппаратное исключение. Тип исключения можно узнать, анализируя флаги регистра MXCSR. Здесь же можно отключить возникновение исключений и выбрать режим округления

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE

Режим "сдвиг к нулю" (Flush to Zero) —

Режим округления (Rounding control) —

Маски исключений:

Точность —

Антипереполнение —

Переполнение —

Деление на ноль —

Денормализованный операнд —

Недействительная операция —

Флаги исключений:

Точность —

Антипереполнение —

Переполнение —

Деление на ноль —

Денормализованный операнд —

Недействительная операция —

Режим «денормализованные=0» —

Сохранение регистра MXCSR в память осуществляется инструкцией `stmxcsr`.

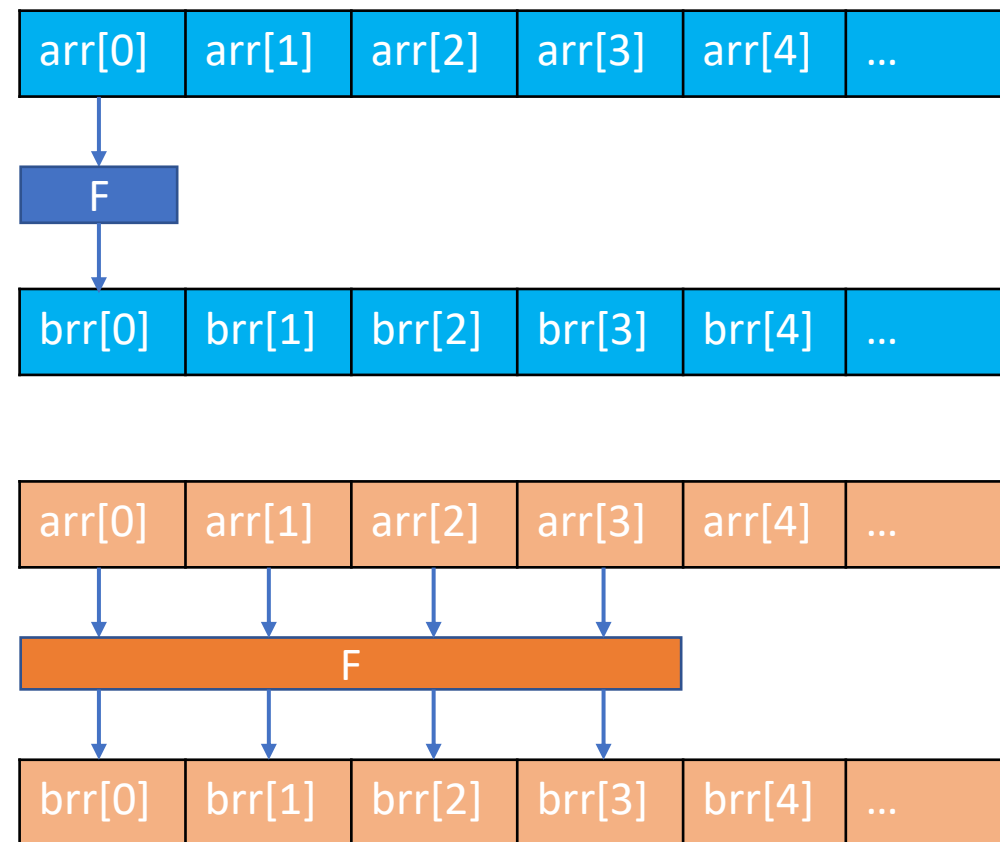
Загрузка содержимого регистра из памяти осуществляется инструкцией `ldmxcsr`.

SIMD

Для поэлементной обработки лучше всего подходит парадигма SIMD (**Single Instruction Multiple Data**, одна команда/много данных).

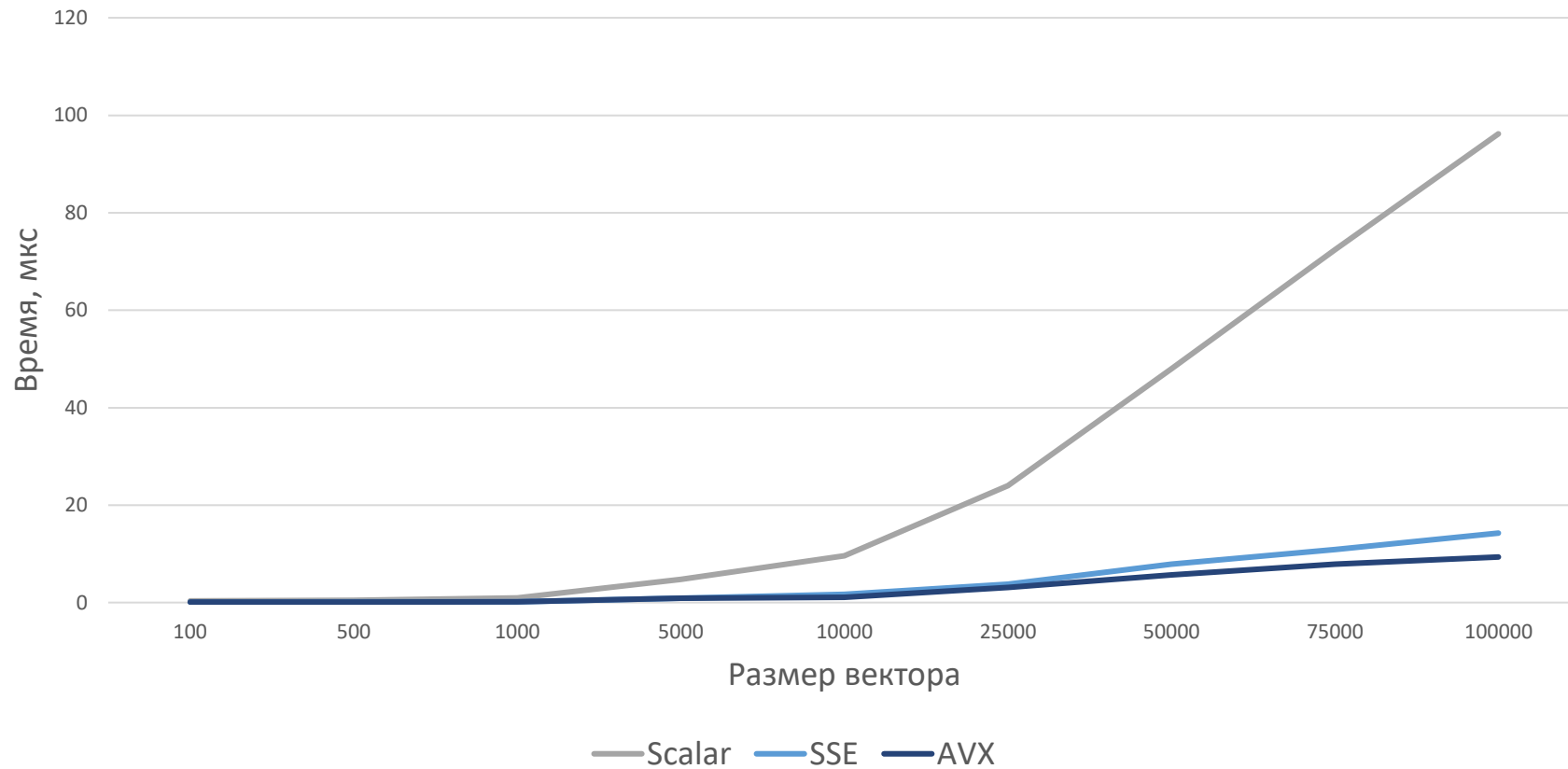
В случае SIMD активно используются векторные операции и специальные векторные регистры.

В векторный регистр сначала загружается несколько элементов массива, а затем выполняется инструкция. При этом *1 инструкцией* обрабатывается несколько элементов.



SIMD

Умножение векторов

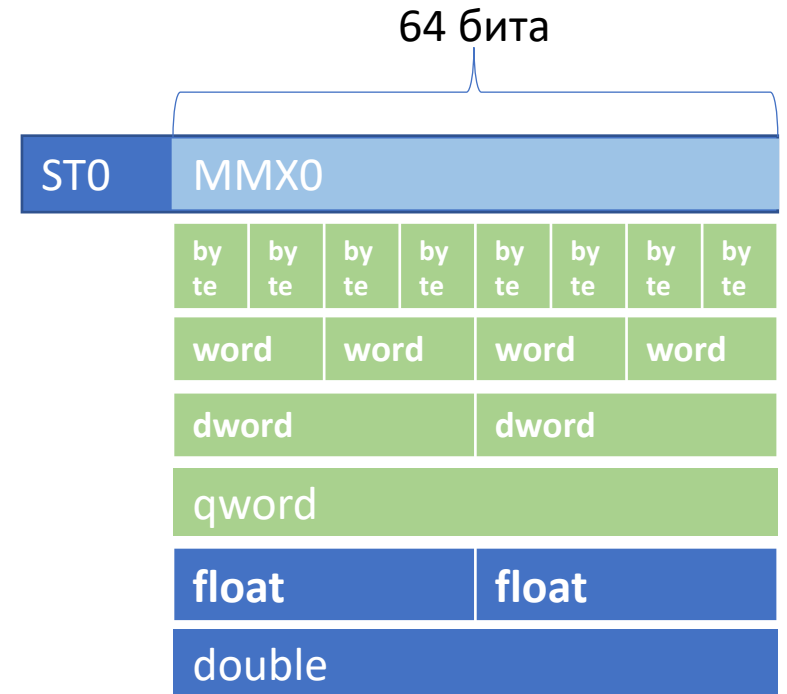


SIMD-расширения

1997 – MMX (Intel). Векторные целочисленные операции в регистрах FPU

1998 – 3DNow (AMD). Векторные вещественные операции в регистрах FPU

1999 – SSE (Intel). Векторные вещественные операции в новом наборе регистров (XMM0-7)



XMM0

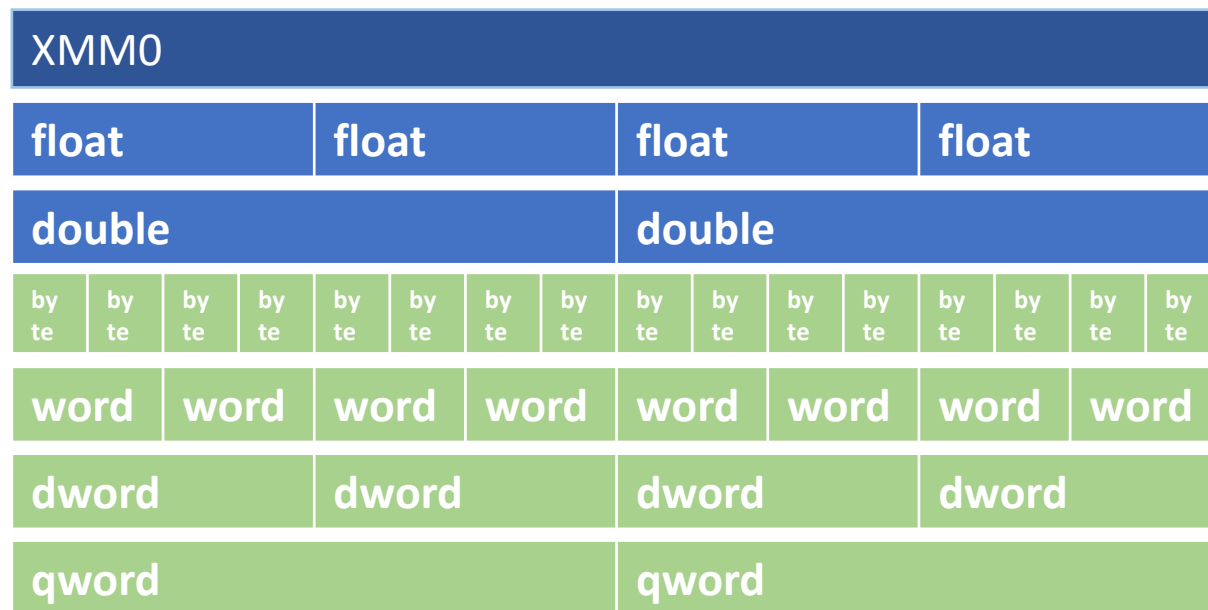
SIMD-расширения

1999 – SSE (Intel). Векторные вещественные операции в новом наборе регистров (XMM0-7)

2003 – (AMD) Удвоение набора XMM-регистров с 8 до 16

2007 – AVX (Intel). Расширение регистров с 128 до 256 бит (XMM->YMM)

2015 – AVX-512 (Intel). Расширение регистров с 256 до 512 бит (YMM->ZMM), удвоение количества регистров



YMM

Инструкции SSE

Инструкции SSE используют обычный двухоперандных синтаксис ($a+=b$)
`instr dest, src.`

Инструкции из скалярного поднабора имеют **суффикс s*** (scalar):
`MOVSS, MOVSD, ADDSS, SUBSD.`

Векторные *вещественные* инструкции имеют **суффикс p*** (packed):
`MOVPS, MOVPD, ADDPS, SUBPD`

Векторные *целочисленные* инструкции имеют **префикс p*** (packed):
`PADDB, PSUBW, PSIGND, PMULQ.`

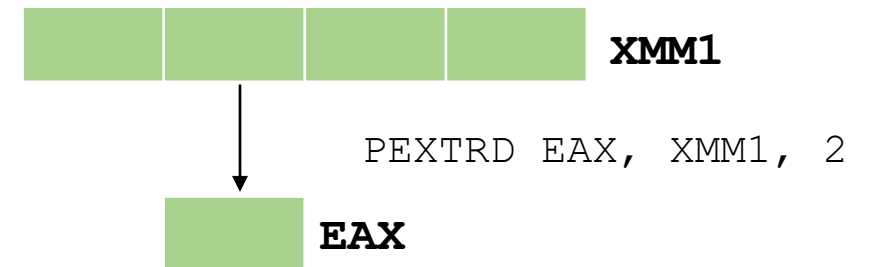
Чтение и запись

Если адрес в памяти гарантированно выровнен по границе 16 байт (т.е. его адрес делится на 16), то для чтения/записи используется инструкция `movap*`. Иначе - `movup*`. Использование неверной инструкции может привести к аппаратному исключению.

Для чтения/записи отдельных элементов используются инструкции `pextr*`/`pinsr*`. Данные инструкции являются *трехоперандными*. Первый операнд является приемником, второй – источником, третий – индексом элемента.

```
movaps xmm0, [rax]
movups xmm1, [rbx]
movups xmm1, xmm0
```

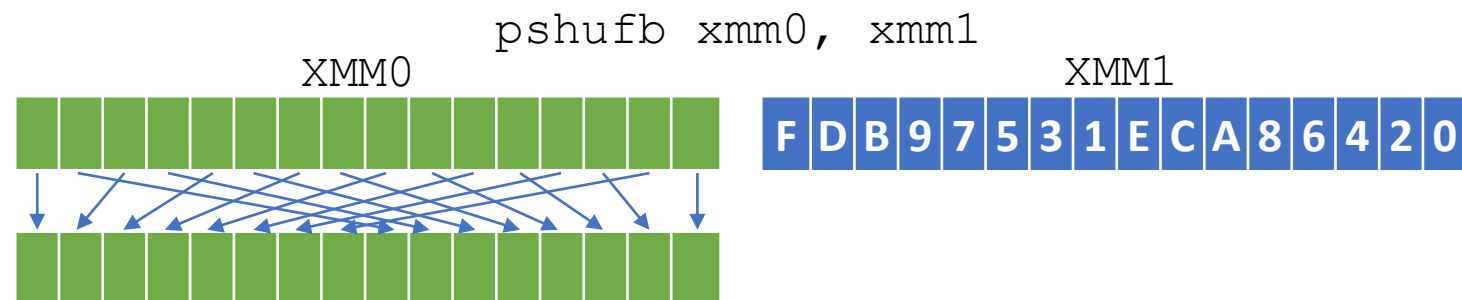
```
pextrb al, xmm15, 15
pinsrw xmm10, [rax], 7
pextrd eax, xmm5, 3
pinsrq xmm0, rbx, 1
```



Переупорядочение элементов

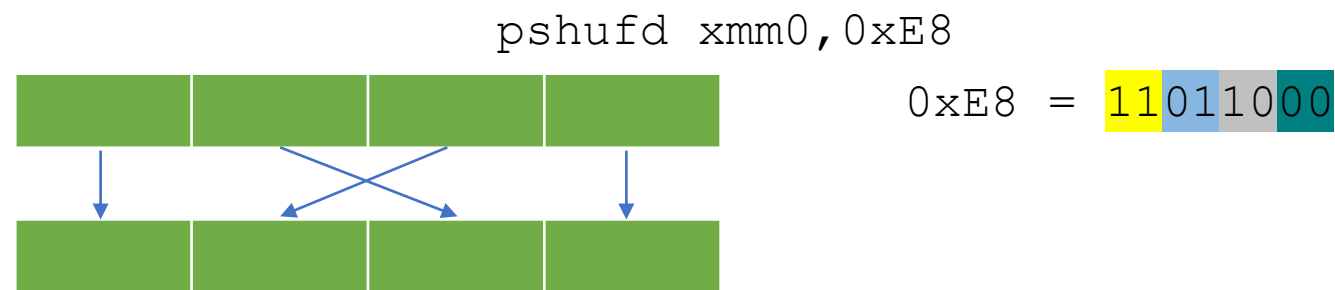
Переупорядочение байтов

`pshufb <xmm-dest>, <xmm-idx>`



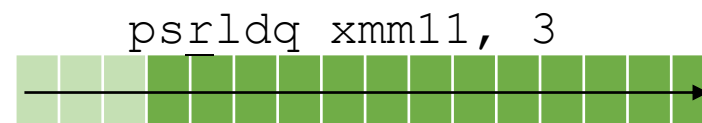
Переупорядочение DWORD

`pshufd <xmm>, <number>`



Побайтовый сдвиг XMM-регистра

`psrldq/pslldq <xmm>, <number>`



Приведение типов

Инструкции `cvt*`, `pmovsx*` и `pmovzx*` используются для преобразования типов.

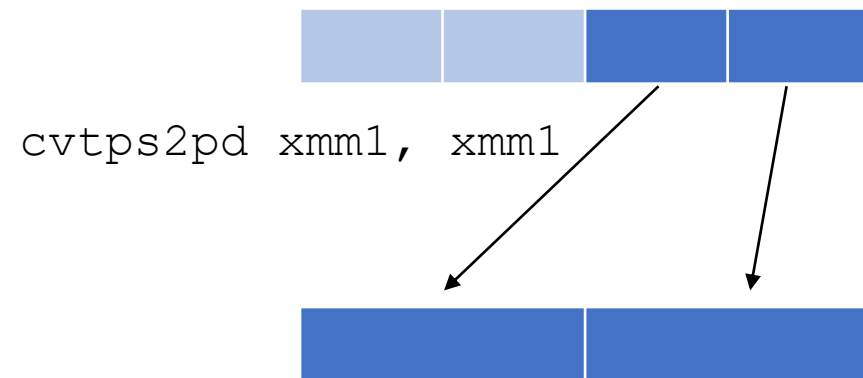
Инструкции `CVT*` проводят преобразования `int<->float`, `int<->double`, `float<->double`.

Инструкции `pmovsx*` и `pmovzx*` используются для расширения целочисленных типов.

Если число элементов в регистре после преобразования меньше, чем до преобразования, то преобразуются *только младшие элементы*.

```
cvt dq2ps  xmm0, xmm1  
cvt dq2pd  xmm1, xmm2
```

```
pmovsxbw   xmm0, xmm1  
pmovzxwd   xmm1, xmm2  
pmovsxdq   xmm2, xmm3
```

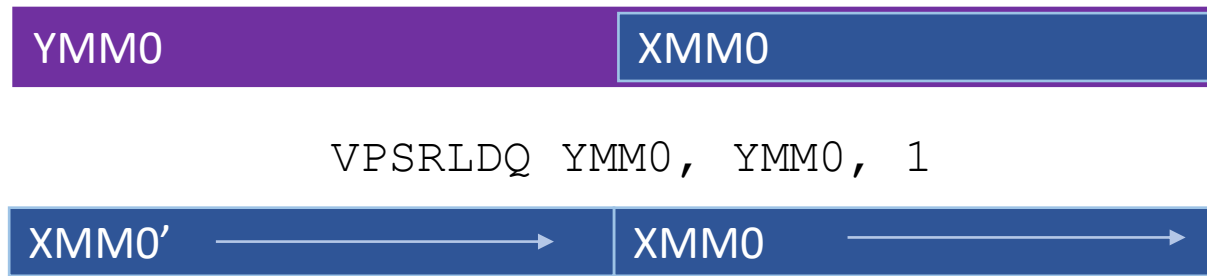


AVX

Набор AVX принес 3 главных новшества – **расширенные векторные регистры**, **новый трехоперандный синтаксис** и **снятие ограничения на выравнивание**.

XMM-регистры являются младшими частями новых YMM-регистров.

При этом, YMM-регистр скорее стоит рассматривать, как 2 почти независимых 128-битных регистра. Например, нельзя сдвинуть весь YMM-регистр – можно только одновременно сдвинуть его старшую/младшую половины или поменять местами половинки целиком. Сделано это для упрощения схемотехники ЦП.



AVX

В AVX были перенесены все инструкции набора SSE, при этом к инструкции добавился **префикс v**:
ADDPS → VADDPS.

При этом обновленные инструкции используют трехоперандный синтаксис ($a = b + c$) и при работе с XMM-регистрами обнуляют старшую половину YMM регистра-приемника.

Кроме того, в новых инструкциях арифметики убрано ограничение на выравнивание адреса (если один из аргументов является адресным выражением).

$$a[:4] = c[:4] - d[:4]; \quad b[:4] = c[:4] + f[:4]$$

; если a, b, c, d не выровнены

```
movups xmm0, [c]
movups xmm1, [d]
movups xmm2, [f]
addps  xmm2, xmm0
subps  xmm0, xmm1
movups [a], xmm0
movups [b], xmm2
```

; если выровнены

```
movaps xmm0, [c]
movps  xmm1, xmm0
subps  xmm0, [d]
addps  xmm1, [f]
movaps [a], xmm0
movaps [b], xmm1
```

; в AVX – без разницы

```
vmovups xmm0, [c]
vsubps  xmm1, xmm0, [d]
vaddps  xmm2, xmm0, [f]
vmovups [a], xmm1
vmovups [b], xmm2
```

Компиляторы и векторные инструкции (пример)

Современные компиляторы по умолчанию не используют векторные инструкции при вещественных вычислениях, т.к. вещественные операции *неассоциативны*. Для того, чтобы разрешить компилятору векторизацию вещественной арифметики, приходится указывать специальные флаги (*--ffast-math* для GCC/Clang, */fp:fast* для MSVC).

Кроме того, по умолчанию компиляторы при сборке используют только SSE-инструкции (т.к. они гарантированно поддерживаются всеми ЦП x86-64). При сборке необходимо явно указывать, что инструкции более новых наборов использовать допустимо (например, *-mavx2* для GCC, */arch:AVX2* для MSVC).

Гарантированным способом задействования векторных инструкций является задействование т.н. intrinsic-функций из заголовочного файла `<immintrin.h>`. Такие функции компилируются в 1 конкретную инструкцию, компилятор только занимается выбором регистров и загрузкой/выгрузкой данных.

[Список intrinsics и соответствующих им инструкций с описанием.](#)