МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(Самарский университет)

Борисов А.Н.

ВЫЧИСЛЕНИЯ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Методические указания к лабораторной работе 2

Самара, 2025

СОДЕРЖАНИЕ

Цели и задачи лабораторной работы

Цель лабораторной работы: изучение способов реализации вычислений с плавающей запятой на языке ассемблера NASM для архитектуры x86-64.

Задание на лабораторную работу: реализовать набор программ, решающих указанные математические задачи в 2-х вариантах: с помощью набора инструкций x87 и с помощью набора инструкций SSE.

1 Введение

1.1 Исторические сведения

Процессор Intel 8087, вышедший в 1978 г., мог осуществлять только целочисленные вычисления. Вычисления с плавающей запятой приходилось эмулировать с помощью целочисленных инструкций, что, очевидно, сказывалось на производительности.

В 1980 г. был выпущен математический сопроцессор (Floating Point Unit, FPU) Intel 8087, предназначенный для выполнения операций вещественной арифметики. Данный сопроцессор был выпущен в виде физически отдельного устройства, который работал параллельно с центральным процессором. Для работы с данным сопроцессором используются инструкции из набора, получившего название x87. Начиная с процессора Intel 80486 (1989 г.), сопроцессор был интегрирован в кристалл центрального процессора и перестал выпускаться в качестве отдельного устройства.

Сопроцессор выступал в качестве единственного средства выполнения вещественных операций до 1999 г., в котором было выпущено расширение набора инструкций x86 под названием SSE (Streaming SIMD Extensions). Данный набор инструкций привнес 8 новых регистров XMM0-XMM7, которые позволяли выполнять арифметические операции над числами одинарной точности. В наборе инструкций SSE2 (2002 г.) добавились инструкции для работы с числами двойной точности, а число регистров XMM увеличилось до 16. В архитектуре x86-64 расширение SSE2 (изначально необязательное для реализации) стало частью базового набора инструкций, что позволило сделать инструкции и регистры из SSE основным средством выполнения вычислений с плавающей запятой.

1.2 Стандарт IEEE-754

1.1.1 Представление чисел с плавающей запятой

Стандарт IEEE-754, принятый в 1985 г. регламентирует представление чисел с плавающей запятой, вычисления над ними и реакцию на некоторые особые случаи, возникающие при вычислениях.

Согласно данному стандарту, числа с плавающей запятой могут кодироваться 32 (числа одинарной точности) либо 64 битами (числа двойной точности). В дальнейшем в стандарт были включены также числа четверной и половинной точности, но в процессорах x86-64 аппаратно они не поддерживаются.

На аппаратном уровне числа представляются в виде $(-1)^s \cdot m \cdot 2^{exp-bias}$, где s – знак числа (0 – положительное, 1 - отрицательное), m – мантисса, exp – экспонента. Мантисса и экспонента кодируются в виде *положительных* чисел, при этом значения $exp = 0$ и $exp = 11 \ldots 11_2$ является зарезервированным.

Поскольку необходимо предоставить возможность представления отрицательных степеней, при вычислениях из экспоненты неявно вычитается постоянное смещение $bias$, значение которого зависит от типа числа (т.е. итоговое представление экспоненты отличается от стандартного представления чисел).

На битовом уровне число кодируется, как показано на рисунке 1.1 с пояснениями в таблице 1.1.

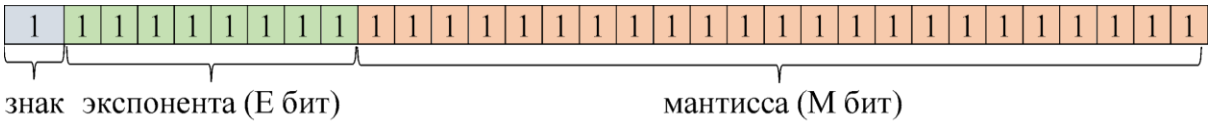

Рисунок 1.1 – Битовое представление вещественного числа

Таблица 1.1. Параметры представления вещественных чисел

Точность числа	**M**	**E**	**bias**
Одинарная	24	8	127
Двойная	53	11	1023

Фактическая точность мантиссы равна (M+1) бит, поскольку для всех *нормализованных* чисел первый бит равен 1 и не хранится.

1.1.2 Бесконечность, денормализованные числа и не-числа

Число является *нормализованным*, если оно представимо в виде $\pm 1, m_0 m_1 \ldots \times 2^{exp-bias}$, где $m_0 m_1 \ldots$ - биты мантиссы, и $exp > 0$. Поскольку битовое представление мантиссы всегда начинается с 1 для нормализованных чисел, первый бит не хранится в представлении числа.

Особыми случаями являются денормализованные числа, бесконечность и не-числа.

Денормализованные числа идентифицируются по значению $exp = 0$. Для денормализованных чисел первый (неявный) бит равен 0. Итоговое значение денормализованного числа представимо в виде $\pm 0, m_0 m_1 \ldots \times 2^{-bias+1}$ (отсюда следует, что ±0 – особое денормализованное число), где $m_0 m_1 \ldots$ - биты мантиссы. Модуль денормализованного числа меньше модуля любого нормализованного числа.

В зависимости от настроек, при получении денормализованного числа в результате операции FPU может просто вернуть результат, округлить его до 0, или выдать аппаратное исключение (Underflow Exception, исключение потери точности).

Остальные особые случаи идентифицируются значением $exp = 2^E - 1$ (т.е. все поле экспоненты заполнено 1).

Бесконечность (как положительная, так и отрицательная), идентифицируется по мантиссе = 0.

Не-числа (Not-a-Number, NaN) идентифицируются по мантиссе ≠ 0. Не-числа являются результатом запрещенной операции *(*$ln\,(-1)$, $\frac{\infty}{\infty}$, $0 \times \infty$, $\mp\infty + \pm\infty$, $\pm\infty - \pm\infty$ и др.). Если хоть один из операндов является NaN, результатом любой операции также является NaN. *NaN не равно никакому другому числу (включая само NaN)*.

Сигнальные NaN, или sNaN, (signaling NaN, мантисса в виде $\mathbf{0}\ldots_2 \neq 0$) заставляют FPU выпустить аппаратное исключение при выполнении операции над ними (при условии, что исключения не маскированы).

«Тихие» NaN, или qNaN, (quiet NaN, мантисса в виде $\mathbf{1}\ldots_2$) не заставляют FPU выпускать исключения. Обычно qNaN являются результатом работы FPU при отключенных аппаратных исключениях.

Для обоих типов NaN оставшаяся часть мантиссы (все биты, кроме первого) может содержать любое значение, и может использоваться для передачи дополнительной информации (payload).

Итоговый вид битовых представлений различных типов вещественных чисел приведен в таблице 1.2 (знаковый бит опущен, т.к. не учитывается).

Таблица 1.2. Битовые представления типов вещественных чисел.

Тип числа	**Экспонента**	**Мантисса**
Нормализованное	$00\ldots01_2$ – $11\ldots10_2$	$00\ldots00_2$ – $11\ldots11_2$
Денормализованное	$00\ldots00_2$	$00\ldots00_2$ – $11\ldots11_2$

Бесконечность	$11\ldots11_2$	$00\ldots00_2$
sNaN	$11\ldots11_2$	$00\ldots01_2 – 01\ldots11_2$
qNaN	$11\ldots11_2$	$10\ldots00_2 – 11\ldots11_2$

1.1.3 Режимы округления

Поскольку в ходе вычислений с плавающей запятой неизбежно возникают округления в младшем разряде, в стандарте IEEE-754 определены 4 возможных режима округления:

1) округление к ближайшему целому;
2) округление вверх (к $+\infty$);
3) округление вниз (к $-\infty$);
4) округление в сторону 0 (к минимальному модулю).

Данные режимы в том числе применяются при преобразовании вещественных чисел в целые.

Результаты округления числа 0,3:

1) к ближайшему целому: 0,0
2) округление вверх (к $+\infty$): 1,0
3) округление вниз (к $-\infty$): 0,0
4) округление в сторону 0: 0,0

Результаты округления числа -0,7:

1) к ближайшему целому: -1,0
2) округление вверх (к $+\infty$): -0,0
3) округление вниз (к $-\infty$): -1,0
4) округление в сторону 0: -0,0

1.3 Ассемблер NASM и вещественные числа

В языке ассемблера NASM вещественные числа кодируются на уровне их байтовых представлений.

В секциях `.data/.rodata` объявить вещественное число можно следующим образом:

```
section .data
    some_float: dd 0.0
    some_double: dq -1.0
    some_long_double: dt 255.21
```

Вещественная константа будет закодирована согласно размеру ячейки (`dd` – одинарная точность, `dq` – двойная точность, `dt` – расширенная точность). Псевдоинструкция `dt` выделяет 10 байт.

Для помещения специальных значений в NASM определены константы `__Infinity__`, `__QNaN__` и `__SNaN__`:

```
float_inf: dd __Infinity__
double_qnan: dq __QNaN__
long_double_snan: dt __SNaN__
```

Для записи в память или в регистр общего назначения вещественного числа используются макросы `__float32__` и `__float64__`:

```
mov dword[x], __float32__(-1.0)
mov rax, __float64__(0.1)
```

Данные макросы вычисляют битовое представление вещественного числа, а затем подставляют его в виде целого числа в инструкцию. Реально исполняются следующие инструкции:

```
mov dword[x], 0xbf800000
mov rax, 0x3fb999999999999a
```

2 Сопроцессор x87

2.1 Общая схема сопроцессора

Сопроцессор x87 реализован по схеме стековой машины. Отличительной особенностью сопроцессора является то, что по умолчанию он работает не по стандарту IEEE-754. Операции в сопроцессоре выполняются с точностью **80-бит** (тип long double в языке C, некоторые компиляторы, в частности MSVC, считают long double = double).

Общая схема сопроцессора приведена на рисунке 2.1. Сопроцессор имеет 8 регистров данных (R0-R7 на рисунке), организованных в виде стека, управляющий регистр (Control Register на рисунке), регистры состояния и тэгов (Status Register и Tag Register на рисунке соответственно), а также регистры последней инструкции, операндов и опкода.

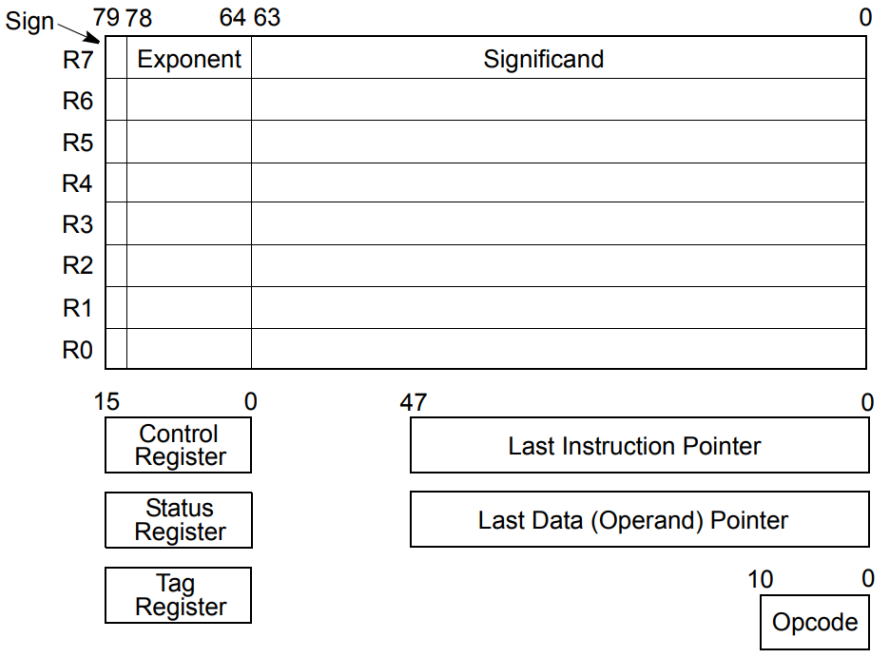

Рисунок 2.1 – Общая схема сопроцессора x87

2.2 Стек сопроцессора. Регистр тегов. Поле TOP регистра состояния

В отличие от регистров общего назначения, регистры x87 не доступны по своим исходным именам. Регистры сопроцессора организованы в стек. В рамках данного стека регистры могут быть либо заняты, либо пустыми. Чтение или запись в пустую ячейку приводят к аппаратному исключению.

Индекс регистра, являющегося вершиной текущего стека, хранится в поле `TOP` регистра состояния. Регистры адресуются относительно текущей

вершины. К примеру, регистр `st0` соответствует текущей вершине стека, регистр `st2` – третьему относительно вершины регистру. Аналогично стеку вызовов, стек сопроцессора растет вниз.

Текущее состояние каждого регистра хранится в регистре тегов, структура которого приведена на рисунке 2.2.

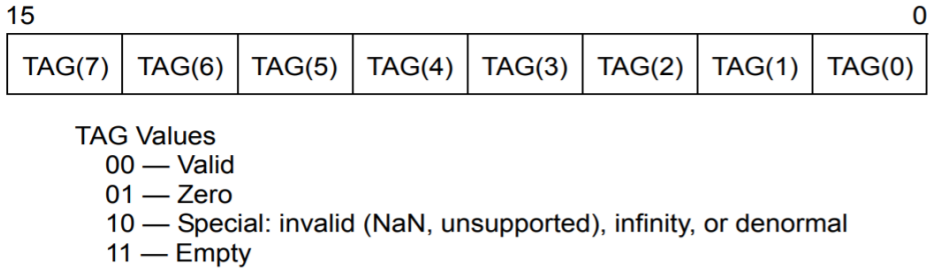

Рисунок 2.2 – Регистр тегов

Регистр тэгов разделен на 8 частей по 2 бита, каждая из которых хранит состояние того или иного регистра. Значение 11 в поле тега означает, что регистр не содержит значения (находится вне стека), чтение из этого регистра приведет к ошибке. Остальные значения означают, что регистр является частью текущего стека, и содержит либо нормализованное число (тег 00), либо 0 (тег 01), либо иное ненормализованное число (тег 10).

Пример состояния стека с 3 числами приведен на рисунке 2.3.

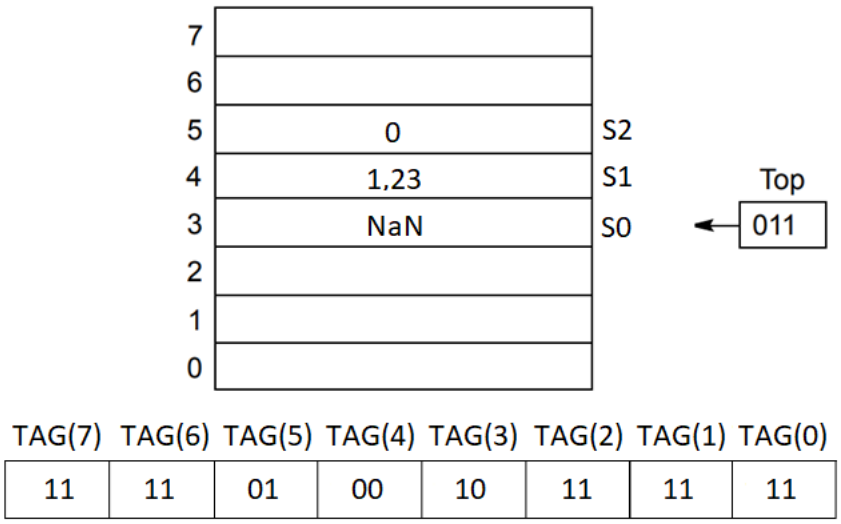

Рисунок 2.3 – Пример стека FPU с 3 элементами

2.3 ПОЛИЗ

Ответ на вопрос о том, почему регистры сопроцессора организованы в стек, можно легко получить, если вспомнить, способы представления последовательности операций математического выражения.

Стандартный способ записи математических выражений (инфиксная запись) обладает неоднозначностью, которую приходится решать с помощью управляющих символов (скобок). К примеру, `a+b/c` и `(a+b)/c` являются совершенно разными выражениями

Префиксная и постфиксная запись не обладают данными недостатками. Префиксная запись была изобретена польским математиком Яном Лукасевичем в 1920 г, и получила название польской нотации или польской записи. Постфиксная запись получила в информатике большее распространение, и в настоящее время широко известна под названиями польской обратной записи, польской обратной нотации, польской инверсной записи или ПОЛИЗ.

В ПОЛИЗ оператор записывается после операндов. К примеру, `a+b` будет записано, как `a b +`. Выражения `a+b/c` и `(a+b)/c` будут записаны, как `a b c / +` и `a b + c /`.

Замечательными свойствами данной нотации являются отсутствие неоднозначностей и простота реализации вычислений на основе стековой машины по следующему правилу:

1) выражение читается слева направо;

2) если текущий элемент – операнд, поместить в стек;

3) если текущий элемент – оператор, вытолкнуть из стека требуемое количество операндов, обратить их порядок, выполнить операцию, записать результат в стек;

4) если вся строка прочитана, то результат – на вершине стека.

К примеру, для `3 16 8 / +`, на момент выполнения операции `/` стек будет иметь вид `[8 16 3]`, из стека будут взяты `8 16` (в таком порядке), операция будет выполнена, как `16/8=2` (отметьте изменение порядка), после операции стек будет содержать `[2 3]`. Процесс выполнения `+` тривиален, после операции стек будет содержать `[5]`.

Для `3 16 + 8 /` , на момент выполнения операции `+` стек будет иметь вид `[3 16]`, из стека будут взяты `3 16`, операция будет выполнена, как `16+3=19`, после операции стек будет иметь вид `[19]`.На момент выполнения операции `/` стек будет иметь вид `[8 19]`, из стека будут взяты `8 19`, операция будет выполнена, как `19/8=2,375`, после операции стек будет иметь вид `[2,375]`.

Во время компиляции математические выражения преобразуются в ПОЛИЗ-запись, которая потом отлично транслируется в инструкции сопроцессора. Алгоритм Дейкстры для преобразования инфиксной записи в постфиксную можно легко найти в открытых источниках.

2.4 Регистр состояния. Управляющий регистр. Исключения

В процессе выполнения арифметических операций могут возникнуть различного рода ошибки: деление на 0, потеря точности и пр.

Аналогично регистру флагов RFLAGS, результат последней операции отражается в регистре состояния сопроцессора. Структура регистра приведена на рисунке 2.4.

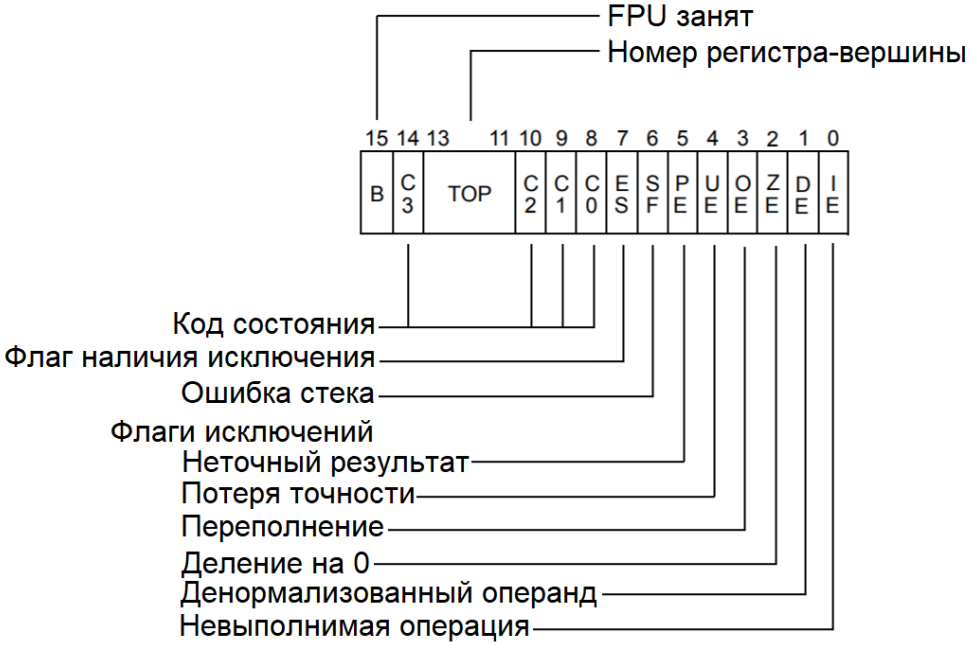

Рисунок 2.4 – Регистр состояния FPU

Код состояния определяет результат последней операции. Ранее код состояния использовался при реализации сравнений, однако начиная с процессоров архитектуры P6 данные коды более могут не использоваться.

Флаг ошибки стека выставляется при переполнении стека (попытке добавить элемент в уже полный стек) либо при попытке удаления элемента из пустого стека.

Флаг наличия исключения равен 1, если равен 1 любой из флагов исключений. Флаги исключений сигнализируют об ошибке, произошедшей в результате последней операции.

Ошибка неточного результата возникает, если результат операции не может быть представлен в точности (например, бесконечная дробь 1/3 не может быть представлена точно).

Ошибка потери точности возникает, если результатом операции является денормализованное число.

Ошибка переполнения возникает, если результатом операции является бесконечность.

Ошибка денормализованного операнда возникает, если один из операндов является денормализованным числом.

Ошибка невыполнимой операции возникает, если запрошенная операция не может быть выполнена корректно (например, ln -1). Это же исключение возникает при попытки загрузки сигнального NaN в FPU.

Сохранение регистра состояния производится инструкциями `FSTSW/FNSTSW` (FPU STore Status Word). Инструкция `FSTSW` проверяет наличие немаскированных исключений перед сохранением слова состояния в памяти, инструкция `FNSTSW` сохраняет слово состояния без проверок. Сохранение слова состояния может потребоваться для проверки кода состояния или его отдельных флагов `C0-C3`.

За управление поведением FPU отвечает управляющий регистр FPU (рисунок 2.5). В частности, управляющий регистр управляет генерацией аппаратных исключений через т.н. *маску исключений*: ошибка приводит к аппаратному исключению, если соответствующий типу ошибки флаг в маске исключений не равен 1.

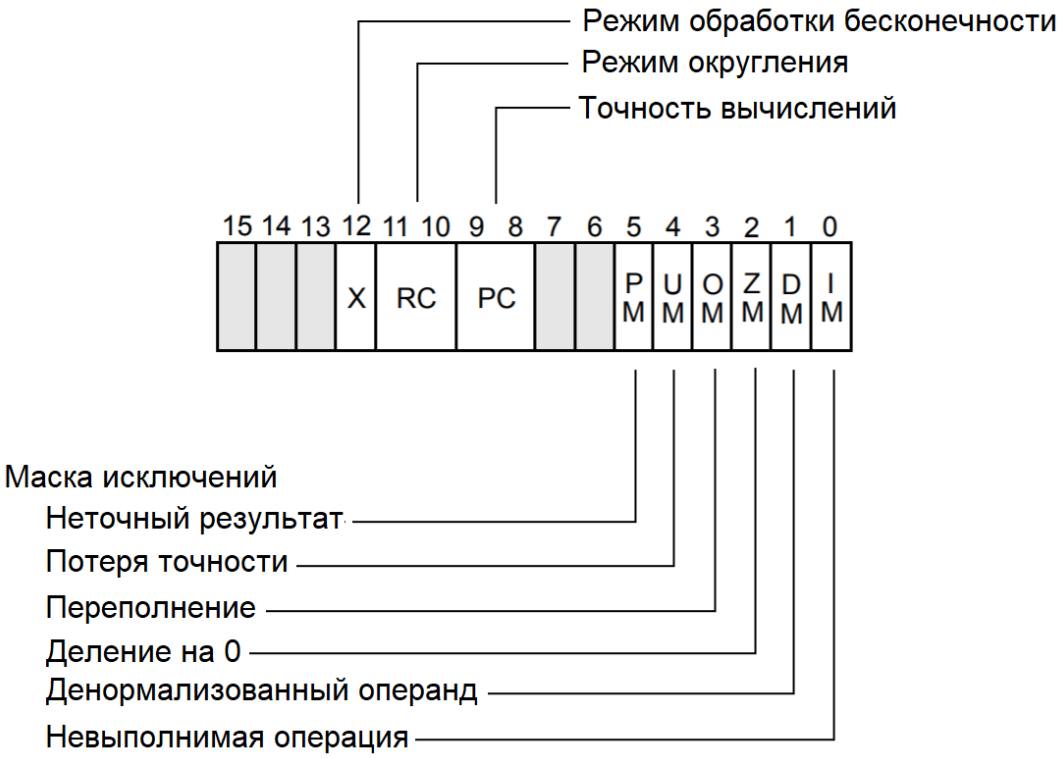

Рисунок 2.5 – Управляющий регистр FPU

Помимо маски исключений, управляющий регистр также содержит 2 бита, указывающие режим округления (см. таблицу 2.1) и 2 бита, указывающие точность вычислений (в пределах л/р не используются).

Таблица 2.1. Коды режимов округления

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

Сохранение управляющего регистра производится инструкциями `FSTCW/FNSTCW` (FPU STore Control Word). Инструкция `FSTCW` проверяет наличие немаскированных исключений перед сохранением состояния регистра в памяти, инструкция `FNSTCW` сохраняет значение управляющего регистра без проверок. Загрузка управляющего регистра производится инструкцией `FLDCW` (FPU LoaD Control Word). Пример изменения состояния управляющего регистра приведен в приложении А.

Отключить возникновение аппаратных исключений можно последовательностью инструкций

```
FNSTCW  [addr]
MOV byte[addr], 0x3F
FLDCW   [addr]
```

Хотя ошибки детектируются FPU немедленно, аппаратные исключения, по историческим причинам, бросаются только во время выполнения *следующей* инструкции FPU (т.е., ошибка деления на 0 при `FDIV` может быть обнаружена только во время выполнения следующей инструкции, например `FADD`).

Для немедленной проверки на наличие немаскированных исключений используется инструкция `FWAIT`. В то время, когда сопроцессор был отдельным устройством, данная инструкция использовалась для ожидания окончания выполнения FPU текущей операции. В настоящее время ее единственная функция – проверка на исключения.

2.5 Инструкции перемещения данных

Поскольку регистры сопроцессора организованы в виде стека, загрузка данных в стек и выгрузка данных из него возможны только в/из вершины стека.

Поместить новое вещественное число в стек можно инструкцией FLD. При этом стек увеличивается на 1. Данная инструкция принимает

только 1 аргумент – источник, в роли которого может выступать другой регистр сопроцессора или память.

Примеры:

`FLD ST5` ; поместить содержимое регистра ST5 в стек (ST5 станет при этом ST6)

`FLD dword[rax]` ; поместить число *одинарной* точности в стек.

`FLD qword[rbx]` ; поместить число *двойной* точности в стек.

`FLD tbyte[rcx]` ; поместить число расширенной точности в стек

При загрузке данных в сопроцессор вещественное число автоматически приводится к 80-битному формату.

Выгрузка вещественных чисел осуществляется инструкциями `FST` и `FSTP`. Инструкция `FST` сохраняет значение на вершине стека. Инструкция `FSTP` дополнительно выталкивает значение из вершины стека (т.е. размер стека уменьшается).

Примеры:

`FST ST5` ; ST5=ST0

`FSTP ST5` ; ST5=ST0, ST0 выталкивается, ST5 становится ST4.

`FST dword[rbx]` ; сохранить в память число одинарной точности

`FSTP qword[rbx]` ; сохранить в память число двойной точности, вытолкнуть значение из вершины стека

Загрузка знаковых целых чисел с последующим преобразованием в вещественное число производится инструкцией `FILD`. Данная инструкция принимает единственный аргумент, который должен быть адресом (не целочисленным регистром!). При этом указатель на `byte` не поддерживается, только `word/dword/qword`.

Примеры:

`FILD word[rax]` ; преобразование short -> long double.

`FILD dword[rax]` ; преобразование int -> long double.

`FILD qword[rax]` ; преобразование long long -> long double.

Преобразование вещественных чисел в целое производится инструкциями `FIST` и `FISTP`. `FISTP` дополнительно выталкивает значение из вершины стека. Преобразование в `qword` поддерживается только инструкцией `FISTP`.

Примеры:

`FIST word[rax]` ; преобразование long double->short.

`FISTP qword[rbx]` ; преобразование long double->int с выталкиванием из вершины.

Обмен значений между вершиной стека и другим регистром производится инструкцией `FXCH`. Данная инструкция принимает 1 операнд – регистр, с которым вершина стека обменивается значением.

Пример: `FXCH ST2 ; ST2⇔ST0`

Для загрузки констант в вершину стека используется ряд специальных инструкций:

`FLD1` ; поместить 1 в стек.

`FLDZ` ; поместить 0 в стек.

`FLDPI` ; поместить π в стек.

`FLDL2T` ; поместить $\log_2 10$ в стек.

`FLDL2E` ; поместить $\log_2 e$ в стек.

`FLDLG2` ; поместить $\log_{10} 2$ в стек.

`FLDLN2` ; поместить $\ln 2$ в стек.

2.6 Инструкции арифметики

Список инструкций, выполняющих стандартные арифметические операции, приведен ниже:

`FADD/FADDP` ; сложение

`FMUL/FMULP` ; умножение

`FSUB/FSUBR/FSUBP/FSUBR` ; вычитание

`FDIV/FDIVR/FDIVP/FDIVR` ; деление

Поскольку арифметические операции выполняются сходным образом, далее будут рассмотрена только операция вычитания.

Инструкция `FSUB` может принимать 0,1 или 2 операнда.

В случае отсутствия операндов инструкции `FSUB` и `FSUBP` являются синонимами – операция выполняется над ST1 и ST0, результат записывается в ST1, значение из вершины стека выталкивается.

В случае одного операнда первым неявным операндом является `ST0`. Значение аргумента вычитается из ST0 без выталкивания.

В случае 2-х операндов один из операндов должен быть регистр `ST0` – вершина стека. Выталкивания значения из вершины не происходит.

Примеры:

```
FSUB             ; ST1-=ST0, вытолкнуть значение из вершины.
FSUB qword[rax]; ST0-=(long double)*RAX.
FSUB st5         ; ST0-=ST5.
FSUB st0, st5    ; ST0-=ST5.
FSUB st5, st0    ; ST5-=ST0.
```

Инструкция FSUBR действует аналогично, но при этом порядок элементов при вычитании меняется.

```
FSUBR          ; ST1=ST0-ST1, вытолкнуть значение из вершины.
FSUBR qword[rax]; ST0=(long double)*RAX-ST0.
FSUBR st5       ; ST0=ST5-ST0.
FSUBR st0, st5  ; ST0=ST5-ST0.
FSUBR st5, st0  ; ST5=ST0-ST5.
```

Инструкции FSUBP и FSUBRP выполняются аналогично своим аналогам без суффикса P, но при этом всегда выталкивают значение с вершины стека.

Приведенные в разделе 2.3 примеры могут быть реализованы следующим образом (считая, что все аргументы – числа одинарной точности).

Выражение: x=a+b/c
ПОЛИЗ: a b c / +
Ассемблер:

```
FLD dword[a]
FLD dword[b]
FLD dword[c]
FDIV
FADD
FSTP dword[x]
```

Выражение: x=(a+b)/c
ПОЛИЗ: a b + c /
Ассемблер:

```
FLD dword[a]
FLD dword[b]
FADD
FLD dword[c]
FDIV
FSTP dword[x]
```

2.7 Инструкции сравнения

Сравнение реализуется с помощью инструкций FCOMI и FCOMIP (FPU COMpare and set Integer flags). Данные инструкции принимают регистр FPU в качестве операнда и сравнивают его с вершиной стека. FCOMIP

дополнительно выталкивает значение из вершины стека. Пример: `FCOMI st1`.

Результат сравнения записывается в флаги `ZF`, `CF` и `PF` регистра `RFLAGS` согласно таблице 2.2.

Таблица 2.2. Флаги, изменяемые `FCOMPI(P)`

Результат сравнения	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Нет порядка (NaN)	1	1	1

Поскольку результат сравнения записывается в флаг `CF` (флаг переноса/беззнакового переполнения), следует использовать условные переходы, завязанные на беззнаковое сравнение (`JA`, `JB`)

Если исключение невыполнимой операции не маскировано, то `FCOMI` сгенерирует аппаратное исключение, если хотя бы один из операндов является NaN. Для корректной обработки NaN необходимо либо маскировать исключение, либо использовать инструкции `FUCOMI`/`FUCOMIP`.

2.8 Тригонометрические инструкции

Тригонометрические функции им вычисляются следующими инструкциями (значения интерпретируются в радианах):

`FSIN` ; ST0 = sin(ST0)

`FCOS` ; ST0 = cos(ST0)

`FSINCOS` ; X = sin(ST0), Y = cos(ST0); вытолкнуть из стека вершину, поместить в стек Y и X.

`FPTAN` ; ST0 = tg (ST0), затем поместить в стек 1.0.

`FPATAN` ; ST1 = arctg(ST1/ST0), вытолкнуть из стека вершину.

Модуль экспоненты аргумента должен быть не более 63 (т.е модуль числа должен находиться в диапазоне $(2^{-64}; 2^{64})$). О причинах подобного см. инструкцию `FPREM1`.

Инструкции для прямого вычисления котангенса, арксинуса, арккосинуса и арккотангенса отсутствуют.

2.9 Инструкции остатка, возведения в степень и логарифмирования

Вещественный остаток от деления на число рассчитывается инструкциями `FPREM` и `FPREM1`. `FPREM1` рассчитывает остаток, согласно IEEE-754.

Данные инструкции рассчитывают остаток от деления `ST0` на `ST1`. При этом, если разница в экспоненте между данными числами более 63 (т.е. остаток от деления `ST0` на `ST1` по модулю больше 2^{63}), то инструкция устанавливает флаг `C2=1` в слове состояния и возвращает частичный остаток. Как следствие, если нет гарантии, что `ST0` и `ST1` отличаются менее, чем в 2^{63} раз, инструкции `FPREM/FPREM1` должны выполняться в цикле с выгрузкой слова состояния инструкцией `FSTSW` в память и явной проверкой флага `C2`.

Вычисление квадратного корня выполняется инструкцией `FSQRT`. Данная инструкция вычисляет ST0=sqrt(ST0).

Вычисление логарифма выполняется с помощью инструкции `FYL2X`. Данная инструкция вычисляет ST1=ST1*$\log_2$(ST0) с вытеснением вершины из стека (на вершине остается логарифм). Данная инструкция позволяет считать логарифм по другому основанию по формуле $x = \frac{x}{a}$.

Возведение в степень производится с помощью инструкций `F2XM1`, `FSCALE` и `FPREM`.

Инструкция `F2XM1` вычисляет ST0 = 2^{ST0} -1. *ST0 должно быть по модулю <1*.

Инструкция `FSCALE` вычисляет $ST0 = ST0 * 2^{sgn(ST1)*\lfloor|ST1|\rfloor}$ (возводит 2 степень, равную ST1 с отброшенной дробной частью).

Возведение в степень по основанию, отличному от 2, производится по формуле $a^b = 2^{b \cdot \log_2 a}$ -1+1. Показатель степени в данном выражении можно вычислить инструкцией `FYL2X`, дробная часть показателя извлекается инструкцией `FPREM`, далее целая и дробная части показателя используются инструкциями `FSCALE` и `F2XM1`. Итоговый код на языке ассемблера:

```
FLD dword[b]
FLD dword[a]
FYL2X    ;вычисляем показатель
FLD1     ;загружаем +1.0 в стек
FLD ST1  ;дублируем показатель в стек
FPREM    ;получаем дробную часть
```

```
F2XM1    ;возводим в дробную часть показателя
FADD     ;прибавляем 1 из стека
FSCALE   ;возводим в целую часть и умножаем
FSTP ST1 ; выталкиваем лишнее из стека
```

3 Набор инструкций SSE

3.1 Регистры XMM. Скалярный режим работы.

Набор инструкций SSE2, включенный в x86-64, определяет 16 регистров, обозначаемых XMM0-15. Регистры XMM независимы друг от друга, в отличие от регистров сопроцессора.

XMM-регистр имеет размер 16 байт и может вмещать вектор из 2 или 4 вещественных чисел. В скалярном режиме используется только младшие 4 или 8 байт регистра. Для работы с числами одинарной точности используются инструкции с суффиксом `SS` (Scalar Single, пример – `ADDSS`). Для работы с числами двойной точности используется инструкции с суффиксом `SD` (Scalar Double, пример - `ADDSD`).

3.2 Инструкции перемещения и приведения типов

Перемещение данных производится инструкциями `MOVS*` (где вместо `*` – `S` для чисел одинарной точности, `D` для чисел двойной точности). Инструкции принимают 2 аргумента – источник и приемник. Хотя бы один аргумент должен быть XMM-регистром. Если аргументом является указатель, то размер читаемых/записываемых данных соответствует типу данных инструкции (4 или 8 байт)

Примеры:

```
MOVSS xmm0, [rax]
MOVSD [rbx], xmm1
MOVSS xmm1, xmm0
```

Переместить значение без приведения типов из целочисленного регистра в XMM-регистр можно инструкциями `MOVD/MOVQ`, суффикс указывает на размер данных (4/8 байт). Данные инструкции полезны для загрузки констант при помощи макросов из раздела 1.3. Пример:

```
MOV RAX, __float64__(-1.0)
MOVQ XMM0, RAX ;
MOV RCX, __float32__(128.23)
MOVD XMM1, ECX ;
```

Приведение между числами разной точности производится инструкциями `CVTSS2SD` и `CVTSD2SS`. Инструкция `CVTSS2SD` (ConVerT Scalar Single To Scalar Double) выполняет приведение float=>double (одинарную в двойную точность). Инструкция `CVTSD2SS` (ConVerT Scalar Double To Scalar Single) выполняет приведение double=>float (двойную в

одинарную точность). Аргументы аналогичны инструкциям MOVSS/MOVSD.

Преобразование целых чисел в вещественные производится инструкциями CVTSI2S* и CVTS*2SI (где вместо * – S или D) (ConVerT Scalar Integer To …). Инструкции CVTSI2S* преобразуют целое число в число одинарной/двойной точности. Инструкции CVTS*2SI преобразуют число одинарной/двойной точности в целое. Для инструкций CVTS*2SI приемник должен быть регистром общего назначения. Источник для всех инструкций может быть как регистром, так и указателем. Размер целого числа выводится из размера аргумента.

Примеры:

```
CVTSI2SS xmm1, eax  ; xmm1 = (float)eax
CVTSI2SD xmm2, [rax]; xmm2 = (double)(*rax)
CVTSD2SI eax, xmm3  ; eax=(int)xmm3
CVTSD2SI rax, [rbx] ; rax=(long long)(*rbx)
```

3.3 Инструкции арифметики

Стандартные инструкции арифметики выполняются следующими инструкциями (где * – S или D):

```
ADDS* XMM, ARG   ; XMM+=ARG.
SUBS* XMM, ARG   ; XMM-=ARG.
DIVS* XMM, ARG   ; XMM/=ARG.
MULS* XMM, ARG   ; XMM*=ARG.
SQRTS* XMM, ARG  ; XMM=sqrt(ARG)
RCPSS XMM, ARG   ; XMM=1/ARG
RSQRTSS XMM, ARG ; XMM=1/sqrt(ARG)
```

Все инструкции принимают 2 операнда. Первый операнд – всегда XMM-регистр, второй операнд может быть XMM-регистром или указателем.

Примеры:

```
ADDSS xmm1, xmm2
SUBSD xmm1, [rax]
SQRTSS xmm2, [rbx]
RCPSS xmm2, xmm3
```

3.4 Инструкции сравнения

Если сравнение необходимо *для дальнейшего выполнения условного перехода*, то следует использовать инструкции COMIS* или UCOMIS*. Инструкции сравнивают операнды и выставляют флаги ZF, CF и PF

регистра RFLAGS согласно таблице 3.1. Инструкция UCOMISx используется, если NaN является допустимым значением.

Пример:

```
COMISS xmm1, [rax]
JZ label
```

Для сравнения вещественных чисел одинарной и двойной точности также используются инструкции CMPS* (CoMPare Scalar Single/Double). Это трехоперандные инструкции, третий аргумент которых является целочисленной константой, указывающей тип сравнения. Данные инструкции заполняют регистр-приемник битами 1 в случае, если результатом сравнения является True, и 0 – если False.

Для удобства в языке ассемблера NASM существуют специальные псевдоинструкции, реализующие сравнение чисел с использованием CMPS*:

```
CMPEQS* XMM, ARG   ; XMM = XMM==ARG ? 0xF...F : 0
CMPLTS* XMM, ARG   ; XMM = XMM<ARG ? 0xF...F : 0
CMPLES* XMM, ARG   ;  XMM = XMM<=ARG ? 0xF...F : 0
CMPNEQS* XMM, ARG  ; XMM = XMM!=ARG ? 0xF...F : 0
CMPNLTS* XMM, ARG  ; XMM = XMM>=ARG ? 0xF...F : 0
CMPNLES* XMM, ARG  ; XMM = XMM>ARG ? 0xF...F: 0
CMPUNORDS* XMM, ARG; XMM=(XMM is NaN)||(ARG is NaN)?-1:0
CMPORDS* XMM, ARG; XMM=!((XMM is NaN)||(ARG is NaN)) ?-1:0
```

Если необходимо определить наибольшее из 2-х чисел, то вместо инструкций сравнения следует использовать инструкции MINS*/MAXS*.

```
MINS* XMM, ARG  ;  XMM = XMM<ARG ? XMM : ARG
MAXS* XMM, ARG  ;  XMM = XMM>ARG ? XMM : ARG
```

3.5 Регистр MXCSR. Маска исключений

Аналогично сопроцессору, в наборе инструкций SSE существует регистр для хранения управляющих флагов и слова состояния SSE-FPU. В отличие от сопроцессора x87, вся информация хранится в одном регистре MXCSR (MX Control and Status Register). Схема регистра приведена на рисунке 4.1.

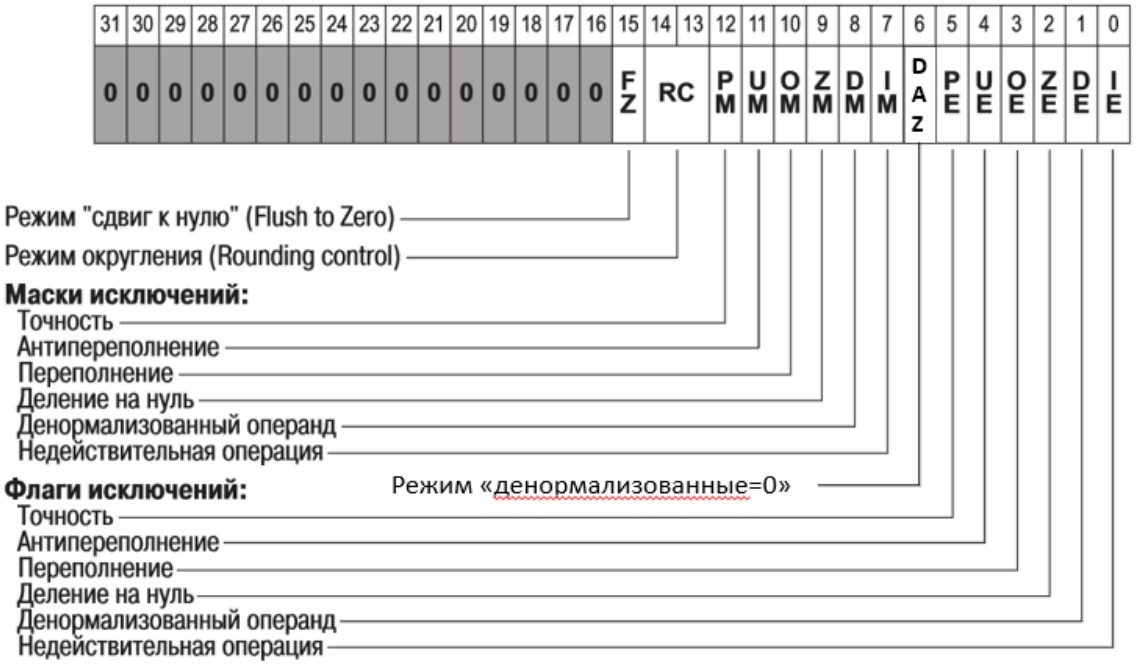

Рисунок 4.1 – Регистр MXCSR

Для сохранения регистра `MXCSR` в память используется инструкция `STMXCSR`. После сохранения регистра его копия в памяти может быть изменена, а затем загружена обратно инструкцией `LDMXCSR`. К примеру, маскировка всех исключений может производится следующими инструкциями:

```
STMXCSR [addr]
OR  word[addr], 0x1F80
LDMXCSR [addr]
```

3.6 Ограничения набора инструкций SSE

Набор SSE не предоставляет инструкций для возведения в степень, логарифмирования и тригонометрических операций.

4 Векторные инструкции

4.1 SIMD. Векторные инструкции SSE

Рассмотренные в предыдущем разделе скалярные инструкции являются лишь частью набора инструкций SSE. Оставшаяся часть предназначена для одновременной обработки нескольких однотипных элементов – вектора. Размер вектора равен размеру XMM-регистра, деленного на размер элемента. В случае чисел одинарной точности XMM-регистр вмещает вектор из 4 чисел одинарной точности.

В целом, одновременное выполнение операций над несколькими однотипными элементами данных является основной идеей парадигмы SIMD (Single Instruction, Multiple Data). Использование данной парадигмы позволяет повысить производительность обработки однотипных данных, поскольку а) одновременная обработка данных сама по себе увеличивает производительность; б) уменьшаются накладные расходы на выполнение (нужно считывать и декодировать меньшее количество инструкций + за счет одновременного считывания/записи групп элементов уменьшается количество отдельных запросов в память).

Векторные инструкции SSE, работающие с вещественными векторами, отличаются от скалярных только суффиксом P (packed): `ADDSS` – скалярная инструкция, `ADDPS` – векторная. Начиная с набора инструкций SSE2, являющегося частью архитектуры x86-64, поддерживается также обработка целочисленных векторов (инструкции начинаются с префикса P, например PADDQ). В рамках данной лабораторной работа с целочисленными векторами не рассматривается.

4.2 Особенности взаимодействия с памятью

Значимым отличием векторных инструкций перемещения данных от скалярных является зависимость от выравнивания данных.

Если точно известно, что считываемые/записываемые данные *выровнены по 16 байтам*, т.е. их адрес делится на 16 без остатка, то для их перемещения используется инструкция `MOVAP*` (MOVe Aligned Packed *). Если гарантии выравнивания данных нет, то следует использовать инструкцию `MOVUP*` (MOVe Unaligned Packed *).

Если адресное выражение является операндом не-MOV инструкции, то итоговый адрес обязан делиться на 16.

В случае, если требование выравнивания данных нарушается, генерируется аппаратное исключение.

Требования к выравниванию данных вытекают во многом из особенностей работы подсистемы памяти: если гарантируется, что данные

выровнены, то доступ к данным можно произвести по «упрощенному» пути, и итоговая операция чтения/записи выполнится быстрее. В некоторых архитектурах невыровненный доступ запрещен в принципе. В x86 исторически сложилось, что невыровненный доступ к данным не запрещен, однако для векторных инструкций, перемещающих 16/32/64 байта за 1 раз, по-видимому, введение данного ограничения имело достаточный смысл.

4.3 Извлечение/вставка отдельных элементов вектора

Для извлечения/вставки младшего элемента вектора можно использовать скалярные инструкции `MOVSS/MOVSD`.

Для извлечения *произвольного* элемента из вектора чисел одинарной точности используется инструкция `EXTRACTPS`, имеющая следующую форму:

```
EXTRACTPS dst, xmm, n
```

где dst – регистр общего назначения либо адресное выражение (без ограничения на выравнивание), xmm – XMM-регистр, n – номер извлекаемого элемента.

Для вставки значения в вектор используется инструкция `INSERTPS`.

Формально для извлечения /вставки элементов могут также использоваться инструкции `PEXTR*/PINSR*`, предназначенные для работы с целочисленными векторами. Данные инструкции не производят никакого преобразования типов, и потому могут применяться к вещественным векторам (а `PEXTRQ/PINSRQ` вовсе являются единственным способом извлечения/вставки элементов векторов двойной точности). Однако по определенным причинам, при работе с векторами одинарной точности лучше использовать специализированные инструкции.

4.4 Инструкции перестановки элементов вектора

Для переупорядочивания элементов вектора без выгрузки вектора в ОЗУ могут использоваться инструкции `PSHUF*` при работе с целочисленными векторами или `SHUFP*` (SHUFfle Packed *) при работе с вещественными векторами. Поскольку в рамках л/р требуется выполнить обработку вещественных векторов одинарной точности, в данном разделе будет рассмотрена только инструкция `SHUFPS`, хотя формально можно использовать целочисленные инструкции для переупорядочивания элементов вещественного вектора.

Инструкция `SHUFPS` имеет следующую форму:

```
SHUFPS <xmmA>, <xmmB>, <n>
```

где xmmA и xmmB – XMM-регистры, n – 8-битное целое число, задающее переупорядочивание.

Третий аргумент рассматривается как 4 2-битных числа, задающих перемещение элементов: $n = I_3^B I_2^B I_1^A I_0^A \mid I_k^l \in \{0,1,2,3\}$. В такой записи верхний индекс указывает, откуда берется элемент вектора – из xmmA или xmmB, значение числа указывает, какой элемент берется, а нижний индекс – в какую позицию выбранный элемент записывается. Например, если xmmA являлся вектором {4, 3, 2, 1}, xmmB – вектором {8,7,6,5}, а $n=1023_4$, то результатом будет вектор {6,5,3,4}. В языке NASM такое значение n можно закодировать бинарной константой `01_00_10_11b` (NASM позволяет вставлять нижнее подчеркивание в качестве разделителя.)

Такая форма инструкции позволяет выполнять как переупорядочивание элементов одного вектора (в этом случае один и тот же регистр указывается в качестве xmmA и xmmB), так и объединение элементов из двух векторов.

4.5 «Горизонтальное сложение»

Сложение элементов вектора выполняется инструкциями PHADD* для целочисленных регистров и HADDP* (Horizontal ADD Packed *) для вещественных регистров. Как и в случае с перестановкой элементов вектора, далее будет рассмотрено только «горизонтальное» сложение вектора вещественных чисел одинарной точности.

Инструкция `HADDPS` имеет следующую форму:

```
HADDPS xmm, arg
```

где xmm – XMM-регистр, arg – другой XMM-регистр либо адресное выражение.

Учитывая, что в данном случае в XMM-регистре хранится 4-элементный вектор, работу инструкции можно выразить следующим образом:

xmm[0] = xmm[0] + xmm[1]
xmm[1] = xmm[2] + xmm[3]
xmm[2] = arg[0] + arg[1]
xmm[3] = arg[2] + arg[3]

Применив эту инструкцию дважды к одному и тому же регистру, можно получить в младшем элементе вектора сумму всех исходных элементов.

4.6 Наборы инструкций AVX и AVX2

Набор инструкций AVX, предложенный Intel в 2008 г. является наследником SSE, привносящим 3 главных новшества.

Первым и главным новшеством является расширение 128-битных XMM-регистров до 256-битных YMM-регистров, при этом XMM-регистры

являются младшими частями соответствующих YMM-регистров (по аналогии с RAX-EAX).

Вторым новшеством является 3-операндный синтаксис. Все инструкции из набора SSE были перенесены в набор AVX с добавлением префикса V, при этом в новых инструкциях на 1 аргумент больше (`ADDPS arg1, arg2 -> VADDPS arg1, arg2, arg3`), при этом первый аргумент в большинстве случаев является только приемником и не участвует в вычислениях (a=b+c вместо a+=b), а в качестве аргументов могут использоваться как XMM, так и YMM-регистры. Если используются XMM-регистры, то старшая часть соответствующего YMM-регистра обнуляется (при использовании SSE-инструкций она остается без изменений). Такое поведение является более выгодным с точки зрения производительности (почему – будет сказано далее по курсу), поэтому, хотя явных препятствий этому нет, лучше использовать либо только SSE-инструкции, либо только AVX-инструкции при работе с XMM-регистрами.

При работе с YMM-регистрами следует учитывать, что они обрабатываются в большинстве случаев не как один 256-битный вектор, а как 2 128-битных вектора. Например, инструкция `VSHUFPS` может переставлять элементы в пределах 128-битных половин, но не между ними, а инструкция `VEXTRACTPS` может извлекать элементы только из младшей половины. Для перестановки 128-битных половин целиком есть инструкция `VPERM2F128`. Такое поведение выбрано с целью упрощения структуры векторного FPU, который в этом случае можно оставить 128-битным (в современных ЦП чаще всего используется полноценный 256-битный FPU).

Третьим (и менее явным) новшеством является ослабление требований к выравниванию данных: в инструкциях, допускающих адресные выражения в качестве аргумента, теперь не требуется, чтобы данные были выровнены по 16/32 байтам (т.е. можно свободно использовать адресные выражения почти во всех AVX-инструкциях). Исключением является `VMOVAP*`, которую можно использовать только для загрузки/выгрузки данных, выровненных по 16(для XMM-регистров) или 32 (для YMM-регистров) байтам.

Набор AVX2, представленный в 2013 г. привнес поддержку обработки целочисленных векторов (которые не поддерживались AVX) и добавление нескольких инструкций, из которых полезными могут показаться `VPERMPS` (перестановка элементов в пределах всего 256-битного вектора) и `VGATHERDPS` (загрузка элементов из памяти, расположенных в памяти непоследовательно).

5 Задание на лабораторную работу

Выполнить 4 задания согласно назначенным вариантам и выбранному уровню сложности.

Задания 1 и 2 на легком и среднем уровнях необходимо выполнить двумя способами:

- с помощью только инструкций сопроцессора x87;

- с помощью инструкций SSE (если в SSE нет инструкции с нужной математической операцией - использовать x87 для выполнения *только этой операции*).

Достаточно, чтобы программа работала для правильных значений переменных (например, вылет или NaN из-за деления на введенный пользователем 0 ошибкой не считается).

Организовывать ввод-вывод не обязательно, за исключением задания 4, где требуется, как минимум, вывести 1(да) или 0(нет). Для демонстрации результатов можно использовать отладчик.

Параметры a, b и c считать константами в пределах одного запуска (т.е. допустимо объявить их в секции. rodata).

В момент выхода из программы стек сопроцессора должен быть пуст (для определения смотрите на регистр тегов в отладке).

Примечание: по определенным причинам стандартные макросы ввода/вывода могут изменять состояние регистров XMM и сопроцессора. Как следствие, ввод лучше производить до выполнения каких-либо вычислений, а вывод – после.

Варианты заданий:

Легкий уровень:

Задание 1: реализовать функцию, округляющую введенное пользователем вещественное число до ближайшего целого:

1) вверх
2) вниз

Задание 2. Вычислить:

1) $y = \frac{a(x+b)}{c}$	4) $y = \frac{(a-x)b}{c}$	7) $y = x^2 + \frac{a}{b}$
2) $y = \frac{a-bx}{c}$	5) $y = \frac{bc}{a+x}$	8) $y = \frac{x^2+a}{b}$
3) $y = \left(\frac{a}{x} + b\right)c$	6) $y = \frac{c}{(x-a)b}$	9) $y = \frac{x^2}{a+b}$

Задание 3. Решить уравнение:

1) $ax^2 = bx + c$	2) $\frac{ax}{b-x} = c$	3) $b(x+a)^2 = c$

Задание 4. Проверить, что заданная точка (x;y) удовлетворяет условию:

1) $y < a\ log_2 x$	3) $y > ln\frac{x}{a}$	5)$y = lg\ ax$
2)$y \geq sin\ (ax)$	$y \leq cos\ (a+x)$	$y\ \neq \frac{si\,n(x)}{a}$

Средний уровень:

Задание 1: см. легкий уровень

Задание 2. Вычислить:

1) $y = a\ sin(bx)\ + cos\ (cx)$	4) $y = e^{ax^2+b}$	7) $y = 2^{ax} + b$
2) $y = tg(ax)\ + ctg(bx)$	5) $y = ln\ (ax)\ + log_2(\frac{x}{b})$	8) $y = ax^b$
3) $y = \frac{sin\ (x)}{ax} + \frac{b\ cos(x)}{x}$	6) $y = \frac{sin(x-a)}{c*\cos\ (b)}$	9) $y = a[x] + b\{cx\}$

Задание 3. Решить уравнение:

1) $cos(lg\,(x-a)\,)=b$	2) $log_2\left(tg\left(\frac{x}{a}\right)\right)=2b$	3) $2^{tg(x)-b}=a$

Задание 4. Проверить, что заданная точка (x, y) удовлетворяет условию:

1) $y<ln^3(sin\,(-x)+a)$	3) $y>sh(x)-a$	5)$y=th\,\left(\frac{a}{x}\right)$
2) $y\geq lg^2(cos\,(ax))$	4) $y\leq ch\,(a-x)$	6) $y\neq cth\,(ax)$

Сложный уровень:

Задание 1:

Реализовать приближенное вычисление инструкциями SSE:

1) косинуса;
2) синуса;
3) экспоненциальной функции;
4) натурального логарифма в диапазоне значений (0; 2).

Задание 2:

Реализовать вычисление косинуса между 2 векторами в двух вариантах:

- с помощью скалярных инструкций SSE;
- с помощью векторных регистров.

Для вычислений использовать одинарную точность. Разрешается использовать любой из доступных векторных наборов инструкций. Полученные результаты сравнить, наблюдаемую разницу значений (если таковая ест) объяснить. Допускается не вводить вектора вручную, а задать их в секции .data/.rodata (можно вынести определения векторов в отдельный файл и использовать директиву `%include`).

Задания 3 и 4: то же, что и в среднем варианте.

Приложение А. Пример изменения маски исключений

```
section .text
global main

enable_zd_exception:
    sub rsp, 8
    fstcw [rsp] ; store the control word
    mov al, [rsp] ; get the lower byte
    and al, 11 ; clear the bit 3 (ZDivision mask bit)
    mov [rsp] , al
    fldcw [rsp] ; load the control word
    add rsp, 8
    ret

main:
    ; uncomment below and see the results
    ; call enable_zd_exception
    fld1
    fldz
    fdiv ; 1/0
    fwait ; check and raise the exception
    xor eax, eax
    ret
```

Приложение Б. Примеры решения задач

Задача 1: Вычислить $y = a\sin^2(x) + tg(bx)$

ПОЛИЗ: $x\ sin\ ^2\ a * b\ x * tg +$

Код для x87:

```
section .rodata
    a: dd 2.0
    b: dd 2.0
    x: dd 4.0

section .bss
    y: resd 1

section .text
global main
main:
    fld dword[x]
    fsin
    fmul st0
    fld dword[a]
    fmul
    fld dword[b]
    fld dword[x]
    fmul
    fptan
    fstp st0 ; pop 1.0 that fptan pushed
    fadd
    fstp dword[y]
    ret
```

Код для SSE:

```
section .rodata
    a: dd 2.0
    b: dd 2.0
    x: dd 4.0

section .bss
    y: resd 1
```

```
section .text
global main
main:
    sub rsp, 8
    fld dword[x]
    fsin
    fstp dword[rsp]
    movss xmm0, [rsp]

    movss xmm1, [x]
    mulss xmm1, [b]
    movss [rsp], xmm1

    fld dword[rsp]
    fptan
    fstp st0
    fstp dword[rsp]

    mulss xmm0, xmm0
    mulss xmm0, [a]
    addss xmm0, [rsp]

    movss [y], xmm0

    add rsp, 8
    ret
```

Задача 2. Решить уравнение $\sqrt{1-\frac{x}{a}} = b$

Замечание: нет решения при $b < 0$.
Решение: $x = a(1-b^2)$.
ПОЛИЗ: $1\ b\ b\ *\ -\ a\ *$

Код для x87:

```
section .rodata
    a: dd 2.0
    b: dd 2.0

section .bss
    x: resd 1

section .text
global main
main:
    fld1
    fld dword[b]
    fld dword[b]
    fmul
    fsub
    fld dword[a]
    fmul
    fstp dword[x]
    ret
```

Код для SSE:

```
section .rodata
    a: dd 2.0
    b: dd 2.0

section .bss
    x: resd 1

section .text
global main
```

```
main:
    movss xmm0, [b]
    mov ecx, 1 ; we can't load constant into xmm
    cvtsi2ss xmm1, ecx ; so we convert it from ECX
    mulss xmm0,xmm0
    subss xmm1, xmm0
    mulss xmm1, [a]
    movss [x], xmm1
    ret
```

Задача 3. Проверить, что заданная точка (x, y) удовлетворяет условию $y < ln\,(a - x)$.

Замечание: правая часть не определена на $\mathbb{R}$ при $x > a$.

ПОЛИЗ правой части: $a\ x\ -\ ln$

ПОЛИЗ логарифма: $ln\ =\ \log_2\ [\log_2 e]\ /$, где $[\log_2 e]$ – константа.

Итоговая ПОЛИЗ: $a\ x\ -\ \log_2\ [\log_2 e]\ /$

Учитывая особенности FYL2X, ПОЛИЗ лучше привести к виду: $1\ [\log_2 e]\ /\ a\ x\ -(fyl2x)$

Код для x87:

```
%include "io64.inc"
section .rodata
    a: dd 4.0
    x: dd 2.0
    y: dd 1.5

section .text
global main
main:
    fld1
    fldl2e
    fdiv ; 1/log2e
    fld dword[a]
    fld dword[x]
    fsub
    fyl2x
    fld dword[y]
    fcomip
    jnb false
    PRINT_DEC 4, 1
    jmp end
false:
    PRINT_DEC 4, 0
end:
    fstp st0 ; clear stack
    xor rax,rax
    ret
```