БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Кафедра информатики и компьютерных систем

А. Н. Дрозд

АРХИТЕКТУРА КОМПЬЮТЕРОВ

Методические указания к лабораторному практикуму

Минск
2025

Описывается базовая архитектура процессоров x86-64. Излагаются основы языка ассемблера MASM. Приводятся четыре лабораторные работы посвященные практическому изучению системы команд x86-64 посредством разработки программ на ассемблере.

Для студентов специальностей 6-05-0533-05 «Радиофизика и информационные технологии», 6-05-0533-11 «Прикладная информатика», 6-05-0533-12 «Кибербезопасность».

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Знакомство с языком ассемблера традиционно является неотъемлемой частью изучения архитектуры компьютеров, поскольку позволяет непосредственно на практике ощутить особенности конкретной архитектуры набора команд (x86, ARM, RISC-V и др.), которые скрыты от программиста на языке высокого уровня.

Несмотря на то, что современные компиляторы и среды разработки обеспечивают удобство программирования, абстрагируя детали аппаратной платформы и минимизируя потребность в ручных оптимизациях, знание ассемблера остается востребованным, а зачастую критически важным в ряде прикладных областей — от системного программирования и оптимизации кода для научных вычислений до информационной безопасности и встраиваемых систем.

Выбор архитектуры x86-64 обусловлен доступностью — практически каждый настольный компьютер в компьютерном классе или ноутбук студента использует процессор с данной системой команд. К тому же все современные процессоры x86-64 содержат векторные расширения базовой системы команд. Наконец, современные процессоры x86-64 имеют многоядерную организацию. Таким образом, на одном персональном компьютере студент может знакомиться со всеми основными классами вычислительных систем по классификации Флинна (SISD, SIMD и MIMD).

1 АРХИТЕКТУРА НАБОРА КОМАНД x86-64

Архитектура набора команд (Instruction Set Architecture, ISA) (или просто архитектура) — это интерфейс между программными и аппаратными средствами (процессором) компьютера. Это спецификация работы процессора с точки зрения его использования программистом. Интерфейс ISA скрывает детали реализации процессора — то, что называется микроархитектурой (или организацией) процессора.

Архитектура набора команд в основном включает в себя описание множества команд и их семантику, а также описание архитектурного состояния, которое эти команды могут изменять.

1.1 Архитектурное состояние x86-64

Основные элементы архитектурного состояния x86-64, которые представляют интерес для прикладного программиста на языке ассемблера, показаны на рисунке 1.1. К ним относятся: адресное пространство; регистры общего назначения; указатель команд; регистр флагов. Группы регистров XMM, YMM, а также регистр управления и статуса MXCSR для операций над числами с плавающей запятой относятся к расширениям SIMD (Single Instruction Multiple Data) системы команд x86, таким как SSE и AVX.

Адресное пространство. Программе доступно линейное адресное пространство, т. е. сплошная последовательность байтов с адресами от 0 до $2^{64} – 1$ (хотя максимальный размер физического адресного пространства ограничен $2^{52} – 1$ байтами). Код программы, данные и стек размещаются в этом едином адресном пространстве. При размещении данных в памяти младший байт располагается по меньшему адресу.

Регистры общего назначения (РОН). Архитектура x86-64 предлагает программисту 16 регистров общего назначения разрядностью 64 бита. Эти регистры используются для хранения целочисленных значений и адресов. Имена 64-битных регистров начинаются с буквы R и приведены в первом столбце таблицы 1.1. Разнобой в названиях регистров связан с долгой историей развития данной архитектуры. Родоначальник семейства x86, процессор Intel 8086, появился в 1978 году и имел восемь 16-разрядных регистров общего назначения. Строго говоря, они еще не были регистрами общего назначения, т. к. каждый из них использовался для особых целей, что отражено в их названиях:

- AX — аккумулятор (**A**ccumulator);
- BX — база (**B**ase), указатель на начало данных в сегменте данных;
- CX — счетчик (**C**ounter);
- DX — регистр данных (**D**ata);
- SI — индекс источника (**S**ource **I**ndex);
- DI — индекс приемника (**D**estination **I**ndex);
- SP — указатель стека (**S**tack **P**ointer);
- BP — указатель базы (**B**ase **P**ointer), указатель на начало данных в стеке.

Для каждого из первых четырех регистров допускалось обращаться отдельно к младшему (Low) или старшему (High) байту, для чего в названии регистра буква X заменяется на L или H соответственно.

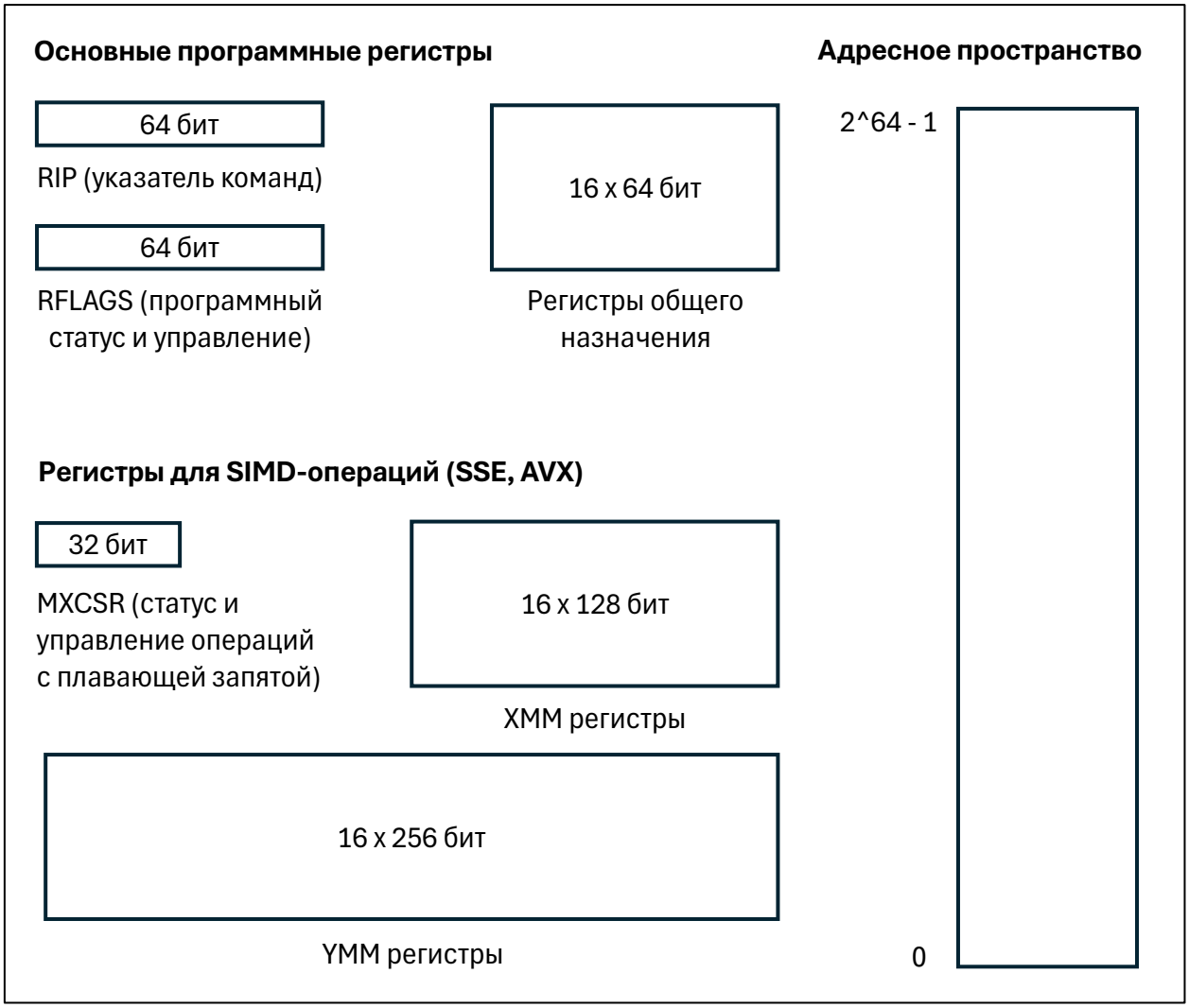

Рисунок 1.1. — Архитектурное состояние x86-64

Начиная с процессора Intel 80386 (1985 год) разрядность регистров была повышена до 32 бит и архитектура стала называться IA-32. Имена расширенных регистров начинаются с буквы E. При этом сохранилась возможность обращаться к старым регистрам меньшей разрядности. Затем в архитектуре x86-64 (2003 год) разрядность регистров общего назначения повысили до 64 бит. Кроме того, добавили еще восемь 64-разрядных регистров с именами от R8 до R15. Для доступа к подрегистрам новых регистров R8–R15 применяется соответствующий суффикс:

- B (**B**yte) —младшие 8 бит регистра, например, R12B;
- W(**W**ord) — младшие 16 бит регистра, например, R12W;
- D (**D**oubleword) — младшие 32 бит регистра, например, R12D.

Следует иметь в виду, что новые и старые регистры не являются независимыми. Например, регистр EAX представляет собой младшие 4 байта регистра RAX. Регистр AX — это младшие два байта регистра EAX. Таким образом модификация, например, регистра EAX может изменить значения регистров AL, AH, AX и RAX. В таблице 1.1 показана взаимосвязь новых и старых регистров общего назначения, а также соглашения по их использованию в контексте вызовов процедур в Windows.

Таблица 1.1. — Регистры общего назначения x86-64

Регистр	**Традиционное использование**	**Биты**			
		0–31	**0–15**	**8–15**	**0–7**
RAX	Возвращаемое значение	EAX	AX	AH	AL
RBX	Сохраняется вызываемым	EBX	BX	BH	BL
RCX	1-й целочисленный аргумент	ECX	CX	CH	CL
RDX	2-й целочисленный аргумент	EDX	DX	DH	DL
RSI	Сохраняется вызываемым	ESI	SI		SIL
RDI	Сохраняется вызываемым	EDI	DI		DIL
RBP	Указатель кадра стека	EBP	BP		BPL
RSP	Указатель стека	ESP	SP		SPL
R8	3-й целочисленный аргумент	R8D	R8W		R8B
R9	4-й целочисленный аргумент	R9D	R9W		R9B
R10	Временные значения	R10D	R10W		R10B
R11	Временные значения	R11D	R11W		R11B
R12	Сохраняется вызываемым	R12D	R12W		R12B
R13	Сохраняется вызываемым	R13D	R13W		R13B
R14	Сохраняется вызываемым	R14D	R14W		R14B
R15	Сохраняется вызываемым	R15D	R15W		R15B

Регистр флагов RFLAGS. Регистр RFLAGS — это 64-разрядный регистр, содержащий группу битов (или флагов) состояния, управления и группу системных флагов. Для прикладного программиста представляют интерес семь флагов. Это флаги состояния (биты 0, 2, 4, 6, 7 и 11), которые фиксируют результат арифметической или логической операции, а также управляющий флаг направления (бит 10):

- CF (бит 0) — флаг переноса (**C**arry **F**lag): устанавливается в 1, если в результате арифметической операции происходит перенос из старшего разряда в следующий (или заем при вычитании). Обозначает переполнение в арифметике без знака.
- PF (бит 2) — флаг четности (**P**arity **F**lag): устанавливается, если младшие 8 битов результата содержат четное число единиц.
- AF (бит 4) — вспомогательный флаг переноса (**A**uxiliary carry **F**lag): указывает на наличие переноса или заёма из 3-го в 4-й бит в результате

арифметической операции. Он используется в двоично-десятичной (BCD) арифметике для коррекции результатов.

- ZF (бит 6) — флаг нуля (**Z**ero **F**lag): устанавливается, если результат последней операции был равен нулю.
- SF (бит 7) — флаг знака (**S**ign **F**lag): результат последней операции был отрицательным (старший бит результата равен 1).
- DF (бит 10) — флаг направления (**D**irection **F**lag): управляет направлением обработки строк данных. Когда DF = 0, операции со строками выполняются в прямом порядке (от младших адресов к старшим), а когда DF = 1 — в обратном (от старших адресов к младшим).
- OF (бит 11) — флаг переполнения (**O**verflow **F**lag): устанавливается в результате переполнения при вычислениях со знаковыми целыми.

Особенно полезны четыре флага: флаг нуля, переполнения, переноса и знака. Вместе они называются *кодами условий*, поскольку значения этих флагов позволяют проверять разные условия, например равны ли два значения между собой или одно больше другого и т. п.

Указатель команд RIP. Регистр указателя команд (RIP) содержит логический адрес следующей инструкции, подлежащей выполнению. Значение в регистре RIP автоматически обновляется во время выполнения каждой инструкции. Оно также неявно изменяется при выполнении инструкций передачи управления, а также прерываниями и исключениями. Например, инструкция CALL (вызов процедуры) помещает содержимое регистра RIP в стек и передаёт управление программой по адресу, указанному в операнде. Инструкция RET (возврат из процедуры) передаёт управление, извлекая восемь верхних байт из стека и загружая их в регистр RIP. Инструкции JMP (переход) и Jcc (условный переход) также позволяют изменить порядок выполнения инструкций, изменяя содержимое регистра RIP.

Регистр RIP также используется для адресации операндов в памяти, когда эффективный адрес определяется путем добавления смещения к содержимому регистра RIP. Значение регистра RIP нельзя получить напрямую из программы. Единственный способ прочитать значение регистра RIP — выполнить инструкцию CALL, а затем извлечь адрес возврата из стека процедуры. Запись в регистр RIP возможна косвенно — путём модификации адреса возврата в стеке и выполнением инструкции возврата (RET).

Регистры для векторных операций. Процессоры x86-64, поддерживающие AVX и AVX2, включают 16 регистров шириной 256 бит с именами YMM0–YMM15. К младшим 128 битам каждого регистра YMM можно обращаться по именам XMM0–XMM15 соответственно, как к отдельным 128-битным регистрам. Регистры YMM и XMM могут использоваться в качестве операндов во многих инструкциях AVX и AVX2 для выполнения операций с упакованными целыми числами или числами с плавающей точкой.

Набор регистров XMM также поддерживает скалярные операции с числами с плавающей точкой, включая базовую арифметику, сравнения и преоб-

разования типов. Процессор использует младшие 32 бита регистра XMM для выполнения скалярных операций с числами с плавающей точкой одинарной точности. Для операций с двойной точностью используются младшие 64 бита регистра XMM. При выполнении таких скалярных операций процессор напрямую не задействует остальные старшие биты регистра XMM. Однако некоторые скалярные инструкции AVX модифицируют эти биты в целевом операнде.

1.2 Фундаментальные типы данных x86-64

Элементарные порции данных, с которыми работает процессор x86-64 во время выполнения программы, документация Intel называет фундаментальными типами данных. К ним относятся: *байт* (8 бит), *слово* (16 бит), *двойное слово* (32 бита), *четверное слово* (64 бита) и *двойное четверное слово* (128 бит). На рисунке 1.2 показан порядок байтов каждого из фундаментальных типов данных при их расположении в памяти. Младший байт располагается по меньшему адресу. Этот адрес считается адресом операнда.

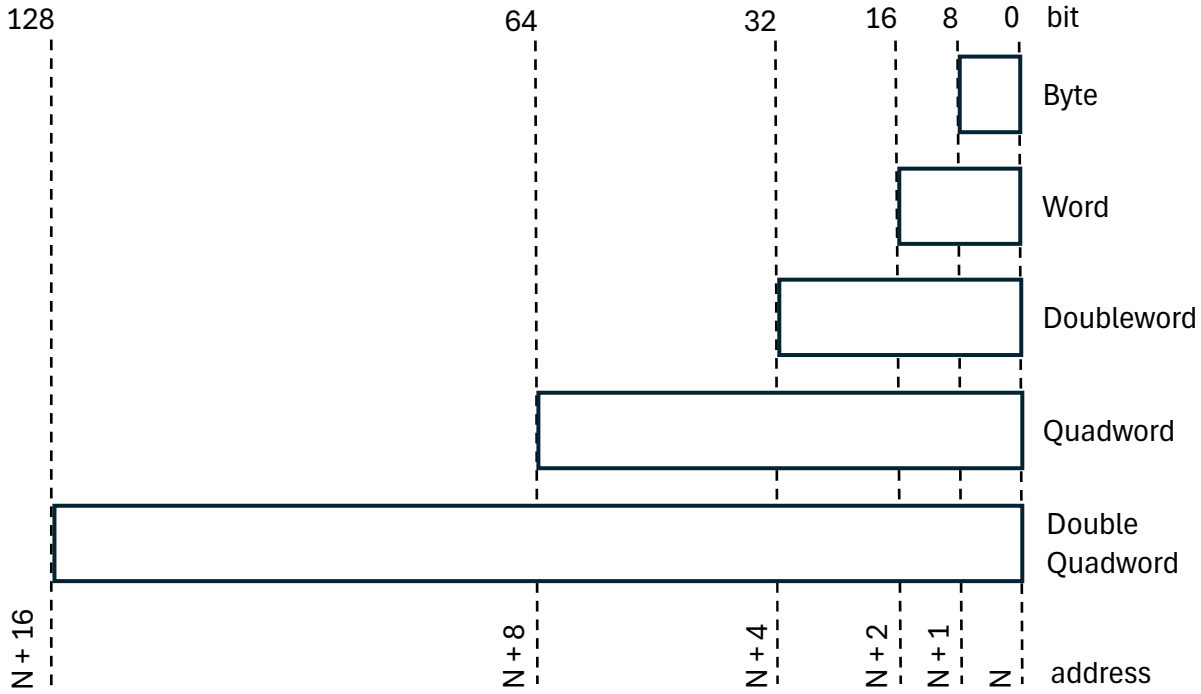

Рисунок 1.2. — Фундаментальные типы данных x86-64

Данные фундаментальных типов не обязаны быть *выровнены* в памяти по естественным границам. *Естественные границы данных* — это адреса, кратные их размеру в байтах. Тем не менее, чтобы повысить производительность программ, структуры данных (особенно стек) следует выравнивать по естественным границам, когда это возможно. Дело в том, что процессору требуется два обращения к памяти для доступа к невыровненному операнду, вместо одного обращения.

Некоторые инструкции поддерживают дополнительные интерпретации фундаментальных типов данных, позволяя выполнять операции с числовыми

типами данных (знаковыми и беззнаковыми целыми числами, а также числами с плавающей точкой) и строками.

Строки — это непрерывные последовательности битов, байтов, слов или двойных слов. Битовые строки могут начинаться с любого бита в памяти и могут содержать до $2^{32} – 1$ битов. Байтовые строки могут содержать байты, слова или двойные слова и могут иметь размер до $2^{32} – 1$ байт (4 ГБ).

1.3 Система команд x86-64

Архитектура x86-64 обладает сложной и обширной системой команд. Коды команд могут иметь разную длину от 1 до 15 байт. Формат кодов инструкций подробно описан в руководствах Intel и AMD. В целом, можно положиться на ассемблер, который генерирует наиболее компактное представление инструкции.

Большинство инструкций x86-64 имеют один или более операндов, задающих исходные значения, над которыми будет выполнена соответствующая этой инструкции операция, и место для записи результата. Операндами могут быть непосредственные значения, содержимое регистров и значения в памяти.

Непосредственное значение — это константа, содержащаяся в самой инструкции в качестве операнда-источника. Любые регистры общего назначения или SIMD-регистры могут использоваться в качестве операндов-источников и операндов-приемников. В таблице 1.2 приведены различные виды операндов на примере инструкции ADD, которая выполняет операцию эквивалентную «+=» в языке программирования C++. Так в первом примере инструкция ADD добавляет непосредственное значение 42 к содержимому регистра RAX.

Таблица 1.2. — Виды операндов на примере инструкции ADD

Источник/ приёмник	**Источник**	**Пример**	**Аналог в C++**
Регистр	Непосредственный операнд	ADD RAX, 42	RAX += 42
Регистр	Регистр	ADD RAX, RBX	RAX += RBX
Регистр	Память	ADD RAX, MyVal	RAX += MyVal
Память	Непосредственный операнд	ADD MyVal, 42	MyVal += 42
Память	Регистр	ADD MyVal, RAX	MyVal += RAX

В примерах из таблицы предполагается, что MyVal — это имя переменной, размещенной в памяти. Следует иметь в виду, что в команде можно задать операнд, располагающийся в памяти, который может быть либо источником, либо приемником данных, но не тем и другим вместе.

Адресация операндов в памяти. Адрес операнда в памяти можно указать прямо в виде статического значения или вычислить как

Address = BaseReg + IndexReg * ScaleFactor + Disp.

Компонент BaseReg (базовый регистр) может быть любым 64-разрядным регистром общего назначения; IndexReg (индексный регистр) может быть

любым 64-разрядным регистром общего назначения кроме RSP; ScaleFactor (масштабный множитель) может принимать значения 1, 2, 4 или 8; Disp — это постоянное смещение со знаком (8, 16 или 32 бит) заданное непосредственно в инструкции. Длина результирующего адреса — 64 бита. Не обязательно использовать все компоненты данной формулы.

В таблице 1.3 показаны возможные формы адресации операнда в памяти на примере инструкции перемещения MOV. Во всех примерах в регистр RAX загружается восьмибайтное значение, способ вычисления адреса которого задается в квадратных скобках.

Таблица 1.3. — Способы адресации операнда в памяти

Форма адресации	**Пример**
Direct	MOV RAX, [10400h]
RIP + Disp	MOV RAX, [MyVal]
BaseReg	MOV RAX, [RBX]
BaseReg + Disp	MOV RAX, [RBX + 16]
IndexReg * SF + Disp	MOV RAX, [R15 * 8 + 48]
BaseReg + IndexReg	MOV RAX, [RBX + R15]
BaseReg + IndexReg + Disp	MOV RAX, [RBX + R15 + 32]
BaseReg + IndexReg * SF	MOV RAX, [RBX + R15 * 8]
BaseReg + IndexReg * SF + Disp	MOV RAX, [RBX + R15 * 8 + 64]

В первой строке таблицы приведен пример *прямой* (*абсолютной*) *адресации*. Он применяется редко, т. к. делает код менее переносимым. Более распространена *адресация относительно указателя команд* RIP, когда адрес операнда вычисляется путем добавления 32-битного смещения к текущему значению RIP. Это делает код программы позиционно независимым. Ассемблер MASM неявно использует этот режим адресации при обращении к переменным, как показано во втором примере из таблицы 1.3. MASM помещает в машинную инструкцию не адрес переменной MyVal, а смещение от адреса начала следующей инструкции до адреса переменной MyVal.

Форма адресации, использующая только базовый регистр аналогична использованию указателя в языке C/C++. С помощью базового регистра и смещения можно осуществлять доступ к полям структуры. Формы адресации, использующие индексный регистр, полезны для доступа к элементам массива. Масштабный множитель удобен при доступе к элементам массива стандартных типов (размером 1, 2, 4 или 8 байт). Самая общая форма адресации может быть полезна для доступа к массиву элементов более сложных типов.

Многие команды x86-64 изменяют один или несколько флагов состояния в регистре RFLAGS. Семейства команд Jcc (условный переход), CMOVcc (условное перемещение) и SETcc (установка байта по условию) используют так называемые *коды условий* вместо шаблонного суффикса «cc» (condition code), чтобы задать проверяемое условие. Истинность условия выясняется

неявно через проверку флагов состояния — как по отдельности, так и в различных комбинациях.

Таблица 1.4 содержит список кодов условий (суффиксов «cc») их расшифровку и соответствующе им значения флагов состояния. В последнем столбце таблицы используются операторы C++ (==, !=, && и ||) для обозначения, соответственно, равенства, неравенства, логического «И» и логического «ИЛИ».

Таблица 1.4. — Коды условий и флаги состояния

Код условия (cc)	**Проверяемое условие**	**Значения флагов состояния**
A	(**A**bove) больше	CF == 0 && ZF == 0
NBE	(**N**either **B**elow or **E**qual) не меньше и не равно	
AE	(**A**bove or **E**qual) больше или равно	CF == 0
NB	(**N**ot **B**elow) не меньше	
B	(**B**elow) меньше	CF == 1
NAE	(**N**either **A**bove nor **E**qual) не больше и не равно	
BE	(**B**elow or **E**qual) меньше или равно	CF == 1 || ZF == 1
NA	(**N**ot **A**bove) не больше	
E	(**E**qual) равно	ZF == 1
Z	(**Z**ero) нуль	
NE	(**N**ot **E**qual) не равно	ZF == 0
NZ	(**N**ot **Z**ero) не нуль	
G	(**G**reater) больше	ZF == 0 && SF == OF
NLE	(**N**either **L**ess nor **E**qual) не меньше и не равно	
GE	(**G**reater or **E**qual) больше или равно	SF == OF
NL	(**N**ot **L**ess) не меньше	
L	(**L**ess) меньше	SF != OF
NGE	(**N**either **G**reater nor **E**qual) не больше и не равно	
LE	(**L**ess or **E**qual) меньше или равно	ZF == 1 || SF != OF
NG	(**N**ot **G**reater) не больше	
S	(**S**ign) знак	SF == 1
NS	(**N**ot **S**ign) нет знака	SF == 0
C	(**C**arry) перенос	CF == 1
NC	(**N**ot **C**arry) нет переноса	CF == 0
O	(**O**verflow) переполнение	OF == 1
NO	(**N**ot **O**verflow) нет переполнения	OF == 0
P	(**P**arity) четный	PF == 1
PE	(**P**arity **E**ven) четный	
NP	(**N**ot **P**arity) нечетный	PF == 0
PO	(**P**arity **O**dd) нечетный	

Коды условий, содержащие слова «above» и «below» используются для беззнаковых целочисленных операндов, тогда как слова «greater» и «less» используются для целых чисел со знаком.

Далее представлен обзор основных групп команд общего назначения x86-64. Эти команды выполняют базовые операции перемещения данных, арифметики, логики, управления потоком выполнения программы и работы со строками, которые программисты обычно используют при написании прикладных программ для процессоров x86-64.

Команды перемещения данных. Эта группа команд включает инструкции, которые копируют или обменивают данные между двумя регистрами общего назначения либо между регистром общего назначения и памятью. Поддерживаются как условные, так и безусловные перемещения данных. В эту группу также входят инструкции, которые помещают данные в стек или извлекают их из стека. В таблице 1.5 приведены инструкции этой группы.

Таблица 1.5. — Команды перемещения данных

Мнемоника	**Описание**
MOV	Копирует данные из РОН или памяти в РОН или память. Также может использоваться для копирования непосредственного значения в РОН или память.
MOVSX / MOVSXD	Копирует значение из источника в приемник выполняя знаковое расширение (**S**ign e**X**tension), если операнд-источник меньшего размера, чем операнд-приемник.
MOVZX	Копирует значение из источника в приемник. Если операнд-источник меньшего размера, чем операнд-приемник, расширяет значение источника, дополняя старшие биты нулями (**Z**ero e**X**tension).
CMOVcc	Семейство команд условного перемещения данных. Суффикс «cc» обозначает один из кодов условий из таблицы 1.4.
PUSH	Помещает операнд (РОН, значение из памяти или непосредственное значение) в стек
POP	Убирает значение с вершины стека, сохраняя его в указанном регистре или ячейке памяти.
XCHG	Обменивает значения двух РОН или РОН и ячейки памяти.
BSWAP	Обращает порядок байтов в 32- или 64-разрядном РОН.
C[WD|DQ|QO]	Удваивает длину неявного операнда в регистре AX, EAX или RAX с учетом знака и сохраняет результат в паре регистров DX:AX, EDX:EAX или RDX:RAX соответственно.

В последней строке таблицы 1.5 с помощью квадратных скобок указаны возможные варианты суффиксов в названии команды преобразования типа данных. Например, команда CWD (**C**onvert **W**ord to **D**ouble) преобразует слово из регистра AX в двойное слово, которое сохраняется в паре регистров DX:AX. Преобразование выполняется путем *расширения знака* (sign extension) исходного операнда, т. е. старший бит (бит знака) AX копируется во все разряды регистра DX. Для команд C[WD|DQ|QO] не требуется явных операндов.

Арифметические команды. В таблице 1.6 перечислены команды, выполняющие базовые арифметические операции над двоичными целыми числами, расположенными в памяти и/или в регистрах общего назначения.

Таблица 1.6. — Команды двоичной арифметики

Мнемоника	**Описание**
ADD	Целочисленное сложение
ADC	Сложение с переносом. К сумме операндов добавляет значение флага переноса CF.
SUB	Вычитание
SBB	Вычитание с заёмом. Отнимает сумму операнда-источника и флага CF от операнда-приемника.
IMUL	Знаковое умножение
MUL	Беззнаковое умножение
IDIV	Знаковое деление.
DIV	Беззнаковое деление.
INC	Инкремент.
DEC	Декремент.
NEG	Изменяет знак операнда.
CMP	Сравнение. Устанавливает флаги состояния по результату вычитания операндов.

Логические команды. В системе команд x86-64 имеется группа команд выполняющих базовые логические операции над двоичными разрядами целых чисел. Эти команды перечислены в таблице 1.7.

Таблица 1.7. — Логические команды

Мнемоника	**Описание**
AND	Побитовое И
OR	Побитовое ИЛИ
XOR	Побитовое исключающее ИЛИ
NOT	Побитовое НЕ
TEST	Логическое сравнение. Устанавливает флаги состояния по результату применения побитового И к операндам.

Команды сдвига. В таблице 1.8 перечислены команды сдвига, которые выполняют сдвиги битов (в том числе циклические) в своих операндах.

Таблица 1.8. — Команды сдвига

Мнемоника	**Описание**
SHR	Логический сдвиг вправо.
SAR	Арифметический сдвиг вправо.
SHL	Логический сдвиг влево.

SAL	Арифметический сдвиг влево.
ROR	Циклический сдвиг вправо.
ROL	Циклический сдвиг влево.
RCR	Циклический сдвиг вправо через бит переноса CF.
RCL	Циклический сдвиг влево через бит переноса CF.

Команды для работы с битами и байтами. Команды для проверки и модификации отдельных битов своих операндов приведены в таблице 1.9. Сюда же включено семейство команд SETcc, которые устанавливают однобайтный операнд, чтобы отобразить состояние флагов в регистре RFLAGS.

Таблица 1.9. — Битовые и байтовые команды

Мнемоника	**Описание**
BT	Проверка бита
BTS	Проверить бит и установить.
BTR	Проверить бит и сбросить.
BTC	Проверить бит и инвертировать.
BSF	Поиск первого установленного бита начиная с младшего разряда.
BSR	Поиск первого установленного бита начиная со старшего разряда.
SETcc	Установка байта по условию. Суффикс cc обозначает один из кодов условий из таблицы 1.4.

Команды передачи управления. В таблице 1.10 собраны команды, позволяющие управлять порядком выполнения команд в программе.

Таблица 1.10. — Команды передачи управления

Мнемоника	**Описание**
JMP	Безусловный переход.
Jcc	Семейство команд условного перехода. Суффикс cc обозначает один из кодов условий из таблицы 1.4.
CALL	Сохраняет содержимое RIP в стеке и делает безусловный переход на адрес, заданный операндом.
RET	Извлекает адрес из стека и выполняет переход на этот адрес.
LOOP	Цикл со счетчиком в RCX. Отнимает 1 от RCX и делает переход при условии RCX != 0.
LOOPE LOOPZ	Отнимает 1 от RCX и делает переход при условии RCX != 0 && ZF == 1.
LOOPNE LOOPNZ	Отнимает 1 от RCX и делает переход при условии RCX != 0 && ZF == 0.

Команды для работы со строками. В таблице 1.11 показана группа команд, которые действуют на последовательности байтов (слов, двойных слов или четверных слов). Все строковые команды используют регистры RSI и RDI как указатели на источник и приемник соответственно. Строковые команды

также инкрементируют или декрементируют эти регистры в зависимости от значения флага направления DF. С помощью префиксов REP, REPE/REPZ или REPNE/REPNZ возможно повторять исполнение строковых команд с использованием регистра RCX в качестве счетчика.

Таблица 1.11. — Команды, манипулирующие строками

Мнемоника	**Описание**
MOVS[B|W|D|Q]	Копирует значение ячейки памяти, на которую указывает RSI в ячейку, на которую указывает RDI.
SCAS[B|W|D|Q]	Сравнивает значение в памяти по адресу RDI со значением в регистре AL, AX, EAX или RAX аналогично CMP.
CMPS[B|W|D|Q]	Сравнивает два значения в памяти по адресу RSI и RDI аналогично CMP.
LODS[B|W|D|Q]	Загружает значение из ячейки памяти, на которую указывает RSI в регистры AL, AX, EAX или RAX.
STOS[B|W|D|Q]	Сохраняет содержимое регистров AL, AX, EAX или RAX в ячейку памяти, на которую указывает RDI.
REP	Повторить пока RCX != 0.
REPE/REPZ	Повторить пока RCX != 0 && ZF == 1.
REPNE/REPNZ	Повторить пока RCX != 0 && ZF == 0.

Команды управления флагами. В таблице 1.12 перечислены команды, явно манипулирующие флагами в регистре RFLAGS.

Таблица 1.12. — Команды управления флагами

Мнемоника	**Описание**
CLD	Сбрасывает флаг направления DF.
STD	Устанавливает флаг направления DF.
CLC	Сбрасывает флаг переноса CF.
STC	Устанавливает флаг переноса CF.
CMC	Инвертирует флаг переноса CF.
LAHF	Загружает младшие 8 битов RFLAGS в регистр AH.
SAHF	Сохраняет регистр AH в младшие 8 битов RFLAGS.
PUSHFD	Сохраняет RFLAGS в стек.
POPFD	Восстанавливает RFLAGS из стека.

Прочие команды. В таблице 1.13 показаны некоторые другие команды общего назначения. Наиболее полезная из них — команда LEA (**L**oad **E**ffective **A**ddress) для загрузки в регистр общего назначения адреса операнда из памяти. Важно, что эта команда не делает обращение в память для чтения данных. Она только вычисляет адрес данных на основе предоставленного режима адресации. Команда NOP (**N**o **OP**eration) не выполняет никакой операции, кроме перемещения указателя команд RIP на следующую инструкцию. Она может

использоваться для выравнивания команд в памяти или для введения контролируемых временных задержек в процессе выполнения программы.

Таблица 1.13. — Разные команды

Мнемоника	**Описание**
LEA	Загрузить эффективный адрес источника в регистр-приемник.
NOP	Перемещает RIP на следующую команду.
CPUID	Получение информации о процессоре.
XLAT/XLATB	Преобразует значение в регистре AL в другое значение в соответствии с таблицей, на которую указывает RBX.

Лабораторная работа № 1

2 ОСНОВЫ ЯЗЫКА АССЕМБЛЕРА MASM

Цель работы: научиться создавать и настраивать проект Visual Studio 2022 для работы с ассемблером MASM для архитектуры x86-64; научиться писать и отлаживать простые линейные программы на языке ассемблера с использованием основных арифметических, логических и побитовых операций.

Язык ассемблера (или просто ассемблер) — это язык программирования низкого уровня, который в основном представляет собой символическую форму машинного кода. Вместо числовых кодов машинных инструкций, ассемблер использует *мнемоники* (удобные для запоминания сокращения), что делает код более читаемым. Он служит связующим звеном между высокоуровневыми языками программирования и аппаратными средствами компьютера. *Ассемблером* также называется программа-транслятор, преобразующая код на языке ассемблера в машинный код.

MASM (Microsoft Macro Assembler) — это популярный ассемблер для систем на базе Windows, поддерживающий архитектуру x86-64. Ассемблер MASM входит в состав среды разработки Visual Studio. Чтобы добавить поддержку MASM в процессе установки Visual Studio необходимо выбрать рабочую нагрузку «Разработка классических приложений на C++». Visual Studio включает как 32-разрядную (*ml.exe*), так и 64-разрядную версию MASM (*ml64.exe*).

2.1 Создание и настройка проекта в Visual Studio 2022

Система проектов Visual Studio поддерживает файлы на языке ассемблера MASM в проектах на C++. MASM собирает исходный код на языке ассемблера в объектный файл, который затем можно связать с объектными файлами, скомпилированными из исходного кода на C++. Далее описаны шаги для создания и настройки проекта.

Создание проекта C++:

1. Запустите Visual Studio 2022.
2. Выберите **Создание проекта**.
3. Из предложенных шаблонов выберите **Пустой проект C++** и нажмите **Далее**.
4. Задайте имя проекта (например, AsmLab1) и расположение, затем нажмите **Создать**.

Настройка проекта для работы с MASM:

1. В окне «Обозреватель решений» щёлкните правой кнопкой мыши по имени проекта и выберите **Зависимости сборки | Настройки сборки**.
2. В открывшемся окне поставьте галочку напротив **masm(.targets, .props)** и нажмите **ОК**.

3. Снова щёлкните правой кнопкой мыши по проекту и выберите **Свойства**.
4. Вверху окна свойств убедитесь, что выбраны **Конфигурация: Все** конфигурации и **Платформа: x64**.
5. Перейдите в раздел **Компоновщик | Система**. Убедитесь, что в поле **Подсистема** выбрано **Консоль (/SUBSYSTEM:CONSOLE)**.
6. В разделе **Компоновщик** перейдите во вкладку **Дополнительно**. Найдите поле **Точка входа** и напишите в нём **main**.
7. Нажмите **Применить** и **ОК**.

Добавление файла с исходным кодом на языке ассемблера:

1. В окне **Обозреватель решений** щёлкните правой кнопкой мыши по папке **Исходные файлы**, выберите **Добавить | Создать элемент**.
2. Выберите **Файл C++ (.cpp)**, но в поле **Имя** укажите имя файла с расширением **.asm** (например, *main.asm*). Нажмите **Добавить**.
3. Щёлкните правой кнопкой мыши по созданному файлу main.asm в обозревателе решений и выберите **Свойства**.
4. Убедитесь, что в разделе **Общие** в поле **Тип элемента** выбран **Microsoft Macro Assembler**.

Вставьте в файл *main.asm* следующий код:

```
; Программа, складывающая два числа
     .CODE
main PROC
     mov rax, 5
     add rax, 6
     ret
main ENDP
     END
```

Чтобы собрать и запустить программу, нажмите Ctrl+F5 или выберите в меню **Отладка | Запуск без отладки**. Если все настроено правильно, программа должна выполниться и завершиться.

При желании можно воспользоваться инструментами командной строки, которые предлагает Visual Studio. В меню **Пуск** на панели задач Windows найдите в списке установленных приложений папку **Visual Studio 2022** и в ней выберите **x64 Native Tools Command Prompt for VS 2022**. Откроется окно интерпретатора командной строки с уже настроенным окружением для работы с 64-разрядным ассемблером. Чтобы скомпилировать (ассемблировать) файл *main.asm*, в командной строке перейдите в папку, в которой расположен файл. Затем выполните следующую команду:

```
ml64 main.asm /link /subsystem:console /entry:main
```

В результате ассемблер *ml64.exe* сгенерирует объектный файл *main.obj*. Флаг `/link` указывает MASM вызвать компоновщик (редактор связей) *link.exe*. Все последующие параметры передаются компоновщику. Параметр `/entry:main` передает компоновщику точку входа в программу, т. е. имя функции, с которой начинается выполнение программы после загрузки исполняемого файла в память. Параметр `/subsystem:console` указывает, что создаваемый исполняемый файл является консольным приложением. То есть он будет запускаться в окне командной строки и взаимодействовать со стандартными потоками ввода и вывода. В итоге в каталоге с файлом *main.asm* появятся еще два файла: объектный файл *main.obj* и исполняемый файл программы *main.exe*.

2.2 Структура программы на языке ассемблера MASM

Программа на ассемблере MASM состоит из секций, которые определяют структуру и организацию кода и данных в памяти (символом точка с запятой (`;`) начинается комментарий, действующий до конца строки):

```
; Структура программы на MASM x64

     .CONST

; Здесь определяются константы

     .DATA

; Здесь определяются статические данные

     .DATA?

; Здесь определяются неинициализированные статические данные

     .CODE

; Здесь помещаются исполняемые процессором инструкции

main PROC  ; начало процедуры main (точка входа в программу)

     ; Здесь полезный код (тело процедуры main)

     ret   ; выход из процедуры
main ENDP  ; окончание процедуры main

     END   ; окончание программы

Здесь можно писать что угодно.
Этот текст MASM игнорирует
```

Секции программы соответствуют особым сегментам адресного пространства доступного исполняющейся программе (процессу) как показано на рисунке

2.1. Каждый сегмент предназначен для хранения различных типов данных и инструкций. Начало каждой секции программы обозначается с помощью следующих директив MASM.

`.CODE` — секция кода. В ней размещаются исполняемые инструкции программы. При загрузке исполняемой программы в оперативную память, результат ассемблирования секции кода будет помещен в соответствующий сегмент адресного пространства, который операционная система (ОС) делает защищенной от записи. Код программы обычно состоит из нескольких процедур (функций). Как минимум требуется наличие функции `main`, которая является точкой входа в программу.

`.DATA` — секция данных. Используется для объявления статических инициализированных переменных. Директива сообщает ассемблеру, что объявленные в этой секции данные должны быть загружены в соответствующий сегмент оперативной памяти, с которой разрешено производить операции чтения и записи.

`.DATA?` — секция неинициализированных статических данных. Соответствующий сегмент памяти ОС обычно заполняет нулями.

`.CONST` — секция констант. Секция предназначена для констант, таблиц и других данных, которые программа не должна изменять в процессе выполнения. Эти значения будут помещены в сегмент памяти, защищенный от записи.

Названные секции могут встречаться в программе любое количество раз в произвольном порядке.

Рисунок 2.1. — Сегменты в адресном пространстве программы

Секции стека и кучи как правило не объявляют в программе на языке ассемблера. Их предоставляет ОС.

2.3 Определение данных в MASM

Для выделения места в памяти для хранения данных в MASM используется следующий синтаксис (в квадратные скобки взяты необязательные элементы):

```
[label] directive initializer [, initializer ...]
```

Здесь `label` — *метка*, т. е. идентификатор для обращения к выделенному элементу данных; `directive` — одна из директив, приведенных в таблице 2.1, которые задают размер и тип данных; `initializer` — значение для инициализации выделенного места в памяти. Инициализатор может быть литералом (числа, символы, строки); выражением, которое вычисляется в значение на этапе ассемблирования; оператором `DUP` (для повторения указанного значения указанное число раз).

Идентификаторы MASM могут начинаться с символа доллара (`$`), подчёркивания (`_`) или буквы латинского алфавита, после которых могут следовать ноль или более буквенно-цифровых символов, знаков доллара или подчёркиваний. Идентификатор не может состоять только из одного символа `$`, т. к. он имеет специальное значение в MASM. По умолчанию MASM нечувствителен к регистру символов, т. е. myVar, MyVar и MYVAR будут считаться одним и тем же идентификатором.

Таблица 2.1. — Директивы объявления данных MASM

Директива	**Описание**
`BYTE` (или `DB`)	Байт. 8-битное значение без знака
`SBYTE`	8-битное значение со знаком (S = signed)
`WORD` (или `DW`)	Слово. 16-битное значение без знака
`SWORD`	16-битное значение со знаком
`DWORD` (или `DD`)	Двойное слово. 32-битное значение без знака (D = double)
`SDWORD`	32-битное значение со знаком (SD = signed double)
`QWORD` (или `DQ`)	Четверное слово. 64-битное значение без знака (Q = quad)
`SQWORD`	64-битное значение со знаком (SQ = signed quad)
`OWORD`	Восьмерное слово.128-битное значение без знака (O = octal)
`REAL4`	32-битное значение с плавающей точкой одинарной точности
`REAL8`	64-битное значение с плавающей точкой двойной точности
`XMMWORD`	Вектор размером 128 байт. Для работы с регистрами XMM
`YMMWORD`	Вектор размером 256 байт. Для работы с регистрами YMM

Примеры объявления данных:

```
  .DATA

b1 BYTE    42
b2 BYTE    1010b     ; b = binary
w  WORD    0BEAFh    ; h = hex
c  BYTE    'x'       ; символ ASCII
sd SDWORD -12
q  QWORD   ?         ; неинициализированная переменная (0)
pi REAL4   3.14159
```

Вопросительный знак (`?`) в качестве инициализатора означает объявление неинициализированной переменной (по умолчанию инициализируется нулем).

В одной инструкции объявления данных можно использовать несколько инициализирующих значений, разделенных запятыми. При этом метка будет представлять собой адрес первого инициализатора. Все значения будут расположены в памяти подряд. Таким образом определяются массивы:

```
     .DATA

; Массив байтов:
list1 BYTE 1,2,3,4

; Массив двухбайтных слов
list2 WORD 10,20,30,40

; можно продолжить:
      WORD 50,60,70,80
```

Аналогично можно определить строки символов в стиле языка C:

```
         .DATA

; Строка символов, завершающаяся нулевым байтом:
greeting  BYTE ‘H’,’e’,’l’,’l’,’o’,’!’,0

; Проще так
greeting1 BYTE ‘Hello!’,0

; или так
greeting2 BYTE “Hello, World!”,0

; Большая строка:
greeting3 BYTE “Welcome to program ”
          BYTE “created by Somebody Hacker.”,0dh,0ah
          BYTE “If you wish to modify this program, please ”
          BYTE “send me a copy.”,0dh,0ah,0
```

Шестнадцатеричные коды `0Dh` и `0Ah` соответствуют символам возврата каретки (CR) и перевода строки (LF) соответственно. Это стандартная последовательность для обозначения перевода строки в Windows.

Выделить память для нескольких одинаковых элементов данных можно с помощью оператора дупликации `DUP`:

```
BYTE 20 DUP(1)             ; выделить 20 байт, все равны 1
WORD 10 DUP(?)             ; выделить 10 неинициализированных слов
BYTE 2  DUP(‘tick’,‘tock’) ; “ticktockticktock”
```

Для неинициализированных данных лучше использовать секцию `.DATA?`. Дело в том, что переменные, объявленные в секции `.DATA?`, могут занимать меньше места на диске в исполняемом файле программы, поскольку не нужно хранить начальные значения таких переменных. Однако это поведение зависит от версии операционной системы и формата объектного модуля.

Данные, значения которых не должны изменяться во время выполнения программы, следует объявлять в секции `.CONST`.

2.4 Объявление именованных констант

Язык ассемблера MASM позволяет определять символические имена для целочисленных значений и выражений, а также для текстовых строк. Символические константы не резервируют место в памяти! Они используются только на этапе ассемблирования. Такие константы можно объявлять в любой секции программы.

Директива, обозначаемая знаком равно (`=`), ассоциирует имя с целочисленным выражением. В следующем примере

```
N = 100
```

определяется константа `N` равная 100. Теперь каждое вхождение этого имени в текст программы ассемблер будет заменять его значением. Имя, объявленное с помощью директивы `=` может быть переопределено.

Директива, `EQU`, позволяет связать символическое имя не только с целочисленным значением, но и с произвольным текстом. В последнем случае, текст помещается в угловые скобки (`<>`). Например, следующий фрагмент программы

```
matrix_sz_1 EQU 10 * 10
matrix_sz_2 EQU <10 * 10>

sz1 WORD matrix_sz_1
sz2 WORD matrix_sz_2
```

заменится ассемблером на

```
sz1 WORD 100
sz2 WORD 10 * 10
```

Имя, определенное с помощью `EQU`, не может быть переопределено.

2.5 Счетчик текущей позиции

С каждой секцией программы (`.code`, `.data`, `.const` и `.data?`) MASM связывает счётчик текущей позиции. Изначально значение счётчика в каждой из этих секций равно 0. Когда MASM встречает в одной из секций объявление переменной (или машинную команду), MASM ассоциирует с этой переменной (или командой) значение счётчика текущей позиции в этой секции. Затем MASM увеличивает значение счётчика на размер объявленного объекта. Этот процесс иллюстрируется следующим примером:

```
 .data
b byte  ?  ; текущая позиция = 0,  размер = 1
w word  ?  ; текущая позиция = 1,  размер = 2
```

```
d dword ?  ; текущая позиция = 3,  размер = 4
q qword ?  ; текущая позиция = 7,  размер = 8
           ; текущая позиция = 15
```

Как следует из комментариев, MASM назначит переменным `b`, `w`, `d` и `q` последовательные «адреса»: 0, 1, 3 и 7 соответственно. Однако это не настоящие адреса переменных во время выполнения программы, а всего лишь смещения относительно начала сегмента данных. Во время выполнения программы ОС разместит каждую секцию в память по определенному базовому адресу. Реальный адрес переменной в памяти получается путем сложения базового адреса сегмента и смещения.

Узнать текущую позицию в секции программы можно с помощью оператора `$`. Например объявление

```
selfPtr QWORD $
```

инициализирует переменную `selfPtr` собственным адресом. В следующем примере

```
list     BYTE 1,2,3,4
ListSize = $ - list
```

константное выражение `$ - list` вычисляет разность текущего смещения и смещения первого элемента массива `list`. В данном случае эта разность равна 4. Таким образом константа `ListSize` получит значение 4. При вычислении количества элементов в массиве, нужно учитывать размер элементов, например:

```
wList     WORD 1000h,2000h,3000h,4000h
wListSize = ($ - wList) / 2

dlist     DWORD 10000000h,20000000h,30000000h,40000000h
dListSize = ($ - dList) / 4
```

В MASM имеется также оператор `this`, который подобно `$` возвращает текущее смещение, но позволяет указать для него тип данных.

2.6 Примеры арифметических выражений в MASM

Следующие примеры иллюстрируют как на языке ассемблера MASM реализовать простейшие инструкции языка высокого уровня C/C++. Например, для того, чтобы присвоить переменной значение константы, в MASM, как и в C++ достаточно одной инструкции:

```
; C++: variable = 47;
mov variable, 47
```

Здесь и далее в примерах подразумевается, что переменные были где-то объявлены. Для того, чтобы присвоить одной переменной значение другой пере-

менной на языке ассемблера уже требуется две инструкции, поскольку в одной команде x86 нельзя задать два операнда в памяти:

```
; C++: var1 = var2; // long long int
mov rax,var2
mov var1,rax
```

Операнды команды `mov` должны быть одного размера, поэтому переменные `var1` и `var2` обязаны иметь такой же размер, как и регистр `rax`, т. е. 64 бита. То же самое присваивание можно выполнить, используя операции со стеком:

```
; C++: var1 = var2;
; альтернативный способ (медленнее)
push var2
pop  var1
```

Команды сложения и вычитания в MASM эквивалентны по смыслу, соответственно, операторам `+=` и `-=` в C++:

```
; C++: x = y + z; // int
mov eax,y
add eax,z
mov x,eax

; C++: k = l - m; // long long int
mov rax,l
sub rax,m
mov k,rax
```

Для умножения в MASM имеется две команды. Для беззнаковых целых чисел используется команда `MUL`, для целых чисел со знаком — `IMUL`. Команда MUL принимает один операнд (регистр или значение в памяти). В качестве второго операнда неявно подразумевается регистр `AL`, `AX`, `EAX` или `RAX`, в зависимости от размера первого операнда. Результат перемножения операндов всегда имеет в два раза больший размер:

```
mul r/m8   ; AX = AL * r/m8
mul r/m16  ; DX:AX = AX * r/m16
mul r/m32  ; EDX:EAX = EAX * r/m32
mul r/m64  ; RDX:RAX = RAX * r/m64
```

Здесь `r` означает один из регистров общего назначения, `m` — операнд в памяти. Числа обозначают разрядность операнда. Символ двоеточие (`:`) означает, что пара регистров, между которыми он расположен, рассматривается как один составной регистр, так что биты правого регистра образуют младшую половину пары, а биты левого регистра — старшую. Если старшая половина результата не ноль, устанавливаются флаги переноса (CF) и переполнения (OF).

Команда знакового умножения IMUL имеет три формы. Кроме одноопернадной формы, такой же как для команды MUL, имеется также двух- и трехоперандная форма.

В форме с двумя операндами первый операнд (приемник) умножается на второй операнд (источник). Первый операнд должен быть регистром общего назначения (16, 32 или 64 бит), а второй — константой (до 32 бит), регистром или ячейкой памяти такого же размера, как первый операнд. Промежуточный результат умножения (вдвое больше по размеру, чем входной операнд) усекается и сохраняется в регистр назначения.

Форма с тремя операндами требует один операнд приемник (первый) и два операнда-источника (второй и третий). Первый операнд-источник (может быть регистром общего назначения или ячейкой памяти) умножается на второй операнд-источник (непосредственное значение). Промежуточный результат (вдвое больше по размеру, чем первый операнд-источник) усекается и сохраняется в операнд приемник (регистр общего назначения). Для двух- и трехоперандных форм IMUL устанавливаются флаги CF и OF, если значащие биты результата теряются при усечении. Некоторые формы умножения показаны в следующем примере:

```
; C++: r = a * b; // unsigned int
mov eax, a
mul b           ; результат в EDX:EAX
mov r, eax

; C++: x = y * z * 47; // signed int
mov  eax, y
imul eax, z           ; eax = y * z
imul eax, eax, 47     ; eax = y * z * 47
mov  x, eax
```

Целочисленное деление осуществляется командами DIV (деление без знака) и IDIV (деление со знаком). Обе команды требуют единственный операнд, который используется в качестве делителя. В качестве делимого используется регистр AX, DX:AX, EDX:EAX или RDX:RAX, в зависимости от размера первого операнда. Команда IDIV может принимать следующие формы:

```
idiv r/m8  ; Делит AX на r/m8. Частное в AL, остаток в AH
idiv r/m16 ; Делит DX:AX на r/m16. Частное в AX, остаток в DX
idiv r/m32 ; Делит EDX:EAX на r/m32. Частное в EAX, остаток в EDX
idiv r/m64 ; Делит RDX:RAX на r/m64. Частное в RAX, остаток в RDX
```

Такие же формы принимает команда DIV.

Перед выполнением деления важно правильно подготовить делимое. Например, при делении 32-разрядных чисел, делимое необходимо сначала занести в регистр EAX. Затем необходимо расширить разрядность этого значения до 64 бит, так чтобы делимое занимало пару регистров EDX:EAX. Для этого в случае беззнакового деления, достаточно обнулить регистр EDX:

```
; C++: x = y / z; // unsigned int
mov eax,y
mov edx,0       ; расширение EAX путем заполнения нулями EDX
div z           ; деление EDX:EAX на z
mov x,eax       ; сохраняем частное
```

При делении со знаком необходимо выполнить знаковое расширение (sign extension) значения `EAX`. Для этого применяется команда `CDQ` (последняя строка в таблице 1.5). Эта команда без аргументов. Она копирует бит знака `EAX` во все разряды `EDX`:

```
; C++: x = y / z; // signed int
mov  eax,y
cdq             ; знаковое расширение EAX в EDX
idiv z          ; деление EDX:EAX на z
mov  x,eax      ; сохраняем частное
```

Примеры вычисления остатка от деления:

```
; C++: r = y % z; // unsigned short
mov ax,y
mov dx,0
div z
mov r,dx

; C++: x = y % z; // signed short
mov  ax,y
cwd             ; знаковое расширение AX в DX
idiv z
mov  x,dx       ; забираем остаток из DX
```

При кодировании сложных арифметических выражений на языке ассемблера, их приходится разбивать на элементарные подвыражения и вычислять последовательно в порядке приоритета. Промежуточные значения желательно сохранять в регистрах общего назначения для эффективности:

```
; C++: r = (a + b) * (y + z); // int
;
; Разбить на простые части:
; tmp1 = a + b;
; tmp2 = y + z;
; r = tmp1 * tmp2;
; По возможности сохранять промежуточные значения tmp1 и tmp2
; в свободных регистрах

mov  eax,a
add  eax,b   ; eax = a + b
mov  ebx,y
add  ebx,z   ; ebx = y + z;
imul eax,ebx     ; eax = eax * ebx
```

```
mov  r,eax


; C++: w = (y + z) * (a - b) / 10; // long long
;
; Разбить на простые части:
; tmp1 = y + z
; tmp2 = a - b
; tmp1 = tmp1 * tmp2
; w = tmp1 / 10

mov  rax,y
add  rax,z    ; rax = y + z
mov  rcx,a
sub  rcx,b    ; rcx = a - b
imul rcx      ; rdx:rax = rax * rcx
mov  rcx,10
idiv rcx      ; rax = rdx:rax / 10
mov  w,rax
```

2.7 Примеры логических выражений в MASM

Несмотря на то, что для представления логических значений достаточно одного бита, в языке ассемблера для этих целей обычно используют один из стандартных типов (например, байт или слово). Для представления логического значения `false` можно использовать число `0`, а для `true` — все ненулевые числа. Другой способ — представлять `true` и `false` числами `1` и `0` соответственно.

Последний способ имеет ряд преимуществ. Во-первых, несмотря на то, что логические команды x86-64 приведенные в таблице 1.7 (`AND`, `OR`, `XOR`, `NOT`) являются поразрядными (побитовыми), при работе с числами `0` и `1` они дают правильный результат, который ожидается для булевых операций (за исключением `NOT`).

Пусть логические переменные `a` и `b` представляются вторым способом, тогда выполнение базовых логических операций над ними реализуется следующим образом:

```
; C++: y = a && b; // bool
mov al,a
and al,b
mov y,al

; C++: y = a || b; // bool
mov al,a
or  al,b
mov y,al

; В C++ нет булевого оператора “Исключающее ИЛИ”
```

```
; Нелегальный C++: y = a xor b; // bool
mov al,a
xor al,b
mov y,al

; C++: b = !a; // bool
mov al,a  ; Команда NOT сама по себе не даст
not al    ; правильный ответ, т.к. (not 0) не равно 1.
and al,1  ; Команда AND исправляет проблему
mov b,al

; C++: b = !a; // bool
; Способ лучше
mov al,a
xor al,1  ; Инвертирует нулевой бит
mov b,al
```

Еще одно преимущество кодирования логических значений числами `0` и `1` состоит в том, что в x86 имеется семейство команд `SETcc` (установить байт по условию «`cc`»), с помощью которых удобно преобразовывать результаты сравнений в булевские значения. Как и все условные команды, `SETcc` обычно применяются в паре с `CMP` (сравнить левый операнд с правым операндом). Сначала команда `CMP L, R` вычисляет разность `L - R` только для того, чтобы в соответствии с полученным результатом изменились флаги состояний в регистре `RFLAGS`. Затем команда `SETcc B` проверяет флаги состояний, чтобы выяснить истинность условия. Проверяемое условие задается суффиксом «`cc`» (смотри таблицу 1.4). Если условие верно, в однобайтный операнд `B` записывается `1`, иначе — `0`. Следующий пример иллюстрирует этот подход:

```
; C++:
; bool b;
; int  a, d, x, y, z;
; b = ((x == y) && (a <= d)) || ((z - a) != 5);

mov   eax, x
cmp   eax, y
sete  al      ; AL = (x == y) ? 1 : 0

mov   ebx, a
cmp   ebx, d
setle bl      ; BL = (a <= d) ? 1 : 0
and   bl, al  ; BL = BL && AL

mov   eax, z
sub   eax, a
cmp   eax, 5
setne al      ; AL = (EAX != 5) ? 1 : 0
or    al, bl  ; AL = ((x == y) && (a <= d)) || ((z - a) != 5)
mov   b, al
```

В группу логических команд x86 (таблица 1.7) входит также команда `TEST`. Она применяет команду `AND` к двум своим операндам, чтобы повлиять на флаги кодов условий. В отличие от `AND` она не записывает результат в первый операнд. С помощью `TEST` можно проверить значение выбранного бита в числе. Например `TEST AL, 1` приведет к установке флага нуля (ZF), если регистр `AL` содержит `0` в младшем разряде. Проверку флага нуля можно осуществить командами `SETZ` или `SETNZ`. Аналогично можно проверить значение любого бита в первом аргументе, задавая соответствующую битовую маску в качестве второго аргумента.

Также с помощью `TEST` можно выяснить равны ли нулю все биты из заданного набора. Например, инструкция `TEST AL, 0Fh` устанавливает флаг нуля, если все 4 младших бита в `AL` равны 0.

Команда `TEST` часто используется для того, чтобы проверить регистр на равенство нулю:

```
test eax, eax
setz bl          ; BL = (EAX == 0 ? 1 : 0)
```

Задания для самостоятельной и индивидуальной работы

1.1. Вычислить значение выражения:

1. $y = (b^2 - (a + 1) \cdot c) / b$
2. $y = a / c - b + (d + 1) \cdot 5$
3. $y = a \cdot b - b^2 / (c + 2)$
4. $y = a \cdot (a + b / 4) / (c - 1)$
5. $y = 3 \cdot a \cdot c / (5 \cdot (b - 5))$
6. $y = a \cdot c - 3 \cdot (b + 3 / c)$
7. $y = a^3 / 3 - c \cdot (b + 3)$
8. $y = (b - 5)^2 / 4 + 2 \cdot b$
9. $y = a \cdot c / 2 - (a + b) / 2$
10. $y = (b^2 - 2 \cdot b) / (3 \cdot a + b)$
11. $y = (a^2 - b^2) / 2 + a \cdot (c + 1)$
12. $y = (a - c)^2 + 2 \cdot a \cdot c / b$
13. $y = (a^3 - 1) / (b - 4) - 5$
14. $y = b^2 \cdot (a + c) + (d - 1) / c$
15. $y = b^3 - 2 \cdot a \cdot b + a^2 / b$
16. $y = b^2 / 3 - a \cdot c + 5$
17. $y = a \cdot c^2 - b \cdot a / c + a / b$

18. $y = a \cdot c \cdot (b - a) / 4 + a^2 - 2$

19. $y = a \cdot c^2 - b \cdot c / a + d / (b + a)$

20. $y = (1 - a)^2 / c + b - 1 + c / 2$

21. $y = (a - b^2) / (c - a) + a^2 - c$

22. $y = (b - 5) \cdot (b + 2) + b + a / 2$

23. $y = (a + b) / c - c^2 \cdot a - b$

24. $y = b \cdot (c - a) - c / (d - 1)$

1.2. Вычислить результат логического выражения $T == S$ для заданных значений логических переменных a, b, c.

+ — логическое сложение (логическое «или»)
· — логическое умножение (логическое «и»)
¯ —логическое отрицание (логическое «не»)

1. $T = a \cdot \overline{b \cdot \overline{c}}, \quad S = a \cdot \overline{b} + a \cdot c$

2. $T = a + \overline{b} \cdot \overline{c}, \quad S = a + \overline{b + c}$

3. $T = (a + b) \cdot \overline{c} \cdot \overline{d}, \quad S = a \cdot \overline{c + d} + b \cdot \overline{c + d}$

4. $T = a \cdot \overline{b} + a \cdot \overline{c}, \quad S = a \cdot \overline{b \cdot c}$

5. $T = \overline{b + c} \cdot d, \quad S = \overline{b} \cdot \overline{c} \cdot d$

6. $T = \left(\overline{b} + \overline{c}\right) \cdot \overline{d}, \quad S = \overline{b + d} + \overline{c + d}$

7. $T = (a + b) \cdot \left(\overline{c} + \overline{d}\right), S = a \cdot \overline{c \cdot d} + b \cdot \overline{c \cdot d}$

8. $T = \overline{(a + b) \cdot \left(\overline{c} + \overline{d}\right)}, S = \overline{a + b} + c \cdot d$

9. $T = \overline{\overline{a} \cdot b + c \cdot \overline{d}}, \quad S = \left(a + \overline{b}\right) \cdot (\overline{c} + d)$

10. $T = \overline{(\overline{a} + b + c) \cdot \overline{d}}, \quad S = a \cdot \overline{b + c} + d$

11. $T = \overline{(a + b + \overline{c}) \cdot d}, \quad S = \overline{a + b} \cdot c + \overline{d}$

12. $T = \overline{\left(a + \overline{b} + c\right) \cdot d}, \quad S = \overline{a + c} \cdot b + \overline{d}$

1.3. Побитовые операции.

1. Задано число n в формате `char`. Сбросить биты 1, 3.
2. Задано число n в формате `unsigned char`. Установить биты 2, 4.
3. Задано число n в формате `signed char`. Инвертировать биты 3, 6.
4. Задано число n в формате `short int`. Сбросить биты 1, 7.
5. Задано число n в формате `unsigned short int`. Установить биты 7, 8.

6. Задано число *n* в формате `signed short int`. Инвертировать биты 1, 15.
7. Задано число *n* в формате `int`. Сбросить биты 7, 15.
8. Задано число *n* в формате `unsigned int`. Установить биты 7, 15.
9. Задано число *n* в формате `signed int`. Инвертировать биты 1, 7.
10. Задано число *n* в формате `unsigned long`. Сбросить биты 1, 10.
11. Задано число *n* в формате `signed long`. Установить биты 0, 1.
12. Задано число *n* формате `long long`. Инвертировать биты 2, 15.

Лабораторная работа № 2

3 УПРАВЛЯЮЩИЕ СТРУКТУРЫ В MASM

Цель работы: научиться реализовывать на языке ассемблера основные управляющие конструкции структурных языков программирования.

Архитектура x86 по умолчанию предполагает последовательное выполнение команд, в порядке их следования в программе. Изменить порядок выполнения можно с помощью команд передачи управления (таблица 1.10). В качестве операндов в таких командах часто используются метки.

Метки применяются для указания местоположения команд или данных. *Метка* — это имя, заменяющее адрес команды или данных. Метки данных — то же, что имена переменных. Чтобы создать метку команд, необходимо поместить имя (метку), за которым следует символ двоеточие (`:`) перед командой. Такие метки используются как цель перехода в командах передачи управления. Например, следующая команда `JMP` передает управление на команду `MOV`, помеченную меткой `target`, образуя цикл:

```
target:
        mov ax,bx
        ...
        jmp target
```

Метку можно располагать в одной строке с командой или в отдельной строке. Директивы ассемблера `PROC` и `ENDP` отмечающие начало и конец процедуры образуют локальную область видимости для меток команд.

Адрес, скрываемый меткой, можно получить двумя способами (с помощью оператора `OFFSET` или используя команду `LEA` (таблица 1.13)):

```
Label1:
        ...
        mov rax, offset Label2
        ...
        lea rbx, Label1
        ...
Label2:
```

Здесь используются 64-битные регистры, поскольку адреса представляют собой 64-битные значения. Команда `LEA` (load effective address) использует 32-битное относительное смещение от текущей команды, поэтому её код значительно короче, чем у команды `MOV`, которая встраивает полный 8-байтный адрес в свой код.

Далее рассматриваются примеры реализации на языке ассемблера традиционных для языков высокого уровня управляющих структур. Реализация простого ветвления (`if`) в MASM:

```
; C++:
```

```
; if(i < 10) ++i;

          cmp i, 10     ; Если !(i < 10)
          jnl EndOfIf   ; перейти на метку EndOfIf
          inc i         ; ++i
EndOfIf:
```

Здесь после команды `CMP`, устанавливающей флаги состояния, используется одна из команд условного перехода `Jcc`. Она проверяет условие, заданное суффиксом «`cc`» (таблица 1.4), и делает переход на метку, указанную в качестве операнда, если условие верно.

Конструкция `if-else` реализуется следующим образом:

```
; C++:
; if(i < 10) ++i;
; else --i;

          cmp i, 10     ; Если !(i < 10)
          jnl ElsePart  ; перейти на метку ElsePart
          inc i         ; ++i
          jmp EndOfIf   ; Перейти на метку EndOfIf
ElsePart:
          dec i         ; --i
EndOfIf:
```

В следующем примере сложное условие `if` вычисляется по сокращенной схеме. Для этого `if` со сложным условием преобразован в цепочку вложенных операторов `if` с простыми условиями:

```
; C++:
; if( (( x > y ) && ( z < t )) || ( a != b ) )  f = d;
;
; Лучше представить в виде
; if(a != b) f = d;
; else if(x > y)
;        if(z < t)
;          f = d;

          mov eax, a
          cmp eax, b
          jne DoIf
          mov eax, x
          cmp eax, y
          jng EndOfIf
          mov eax, z
          cmp eax, t
          jnl EndOfIf
DoIf:
          mov eax, d
```

```
          mov f, eax
EndOfIf:
```

Оператор множественного выбора `switch` в языке C++ семантически эквивалентен цепочке вложенных `if-else`, т. е. следующие два фрагмента кода эквивалентны:

```
switch(i)
{
    case 0: действие_0; break;
    case 1: действие_1; break;
    case 2: действие_2; break;
}

if(i == 0)      действие_0;
else if(i == 1) действие_1;
else if(i == 2) действие_2;
```

Таким образом оператор `switch` можно реализовать на языке ассемблера как цепочку `if-else`. Такая реализация имеет сложность $O(N)$, где N — количество ветвей `case`. Если N велико, а значения констант, среди которых производится поиск, образуют плотную последовательность целых чисел, эффективнее использовать таблицу косвенных переходов. Следующая реализация имеет сложность $O(1)$:

```
          mov eax,i
          lea rbx, jmp_table
          jmp qword ptr [rbx + rax * 8]

jmp_table  qword case_0, case_1, case_2

case_0:    ; действие_0
           jmp end_switch

case_1:    ; действие_1
           jmp end_switch

case_2:    ; действие_2
           jmp end_switch

end_switch:
```

В этом примере создается массив меток (т. е. таблица адресов переходов) `jmp_table`. В полноценной программе этот массив лучше разместить в секции `.CONST`. Команда `lea` загружает в регистр `rbx` адрес начала таблицы переходов. Искомое значение `i`, записанное в регистр `rax` используется как индекс в таблице переходов, чтобы сразу обратиться к нужному элементу. В данном примере не показана проверка на выход за пределы таблицы.

Для реализации циклов на языке ассемблера также используются команды условных и безусловных переходов. Цикл `while` можно реализовать как оператор `if` с командой безусловного перехода в конце:

```
; C++:
; i = 0;
; while( i < 10 )
; {
;    ++i;
; }

            mov i, 0
BeginWhile:
            cmp i, 10       ; Если !(i < 10)
            jnl EndWhile    ; выйти из цикла
            inc i
            jmp BeginWhile ; Повторить
EndWhile:
```

Цикл `do-while` на языке ассемблера реализуется еще проще:

```
; C++:
; i = 0;
; do
; {
;    ++i;
; } while( i < 10 );

            mov i, 0
do_while:
            inc i
            cmp i, 10     ; Если i < 10
            jl  do_while  ; повторить
```

Здесь на каждом шаге цикла вместо двух команд перехода выполняется одна. Поскольку разные виды циклических конструкций (`while`, `for`, `do-while`) взаимозаменяемы, то с точки зрения эффективности в программах на языке ассемблера выгодно использовать циклы с проверкой условия в конце.

Если тело цикла нужно повторить N раз, то эффективнее будет изменять счетчик цикла не от 0 до $N-1$, а в обратном порядке. Допустим в предыдущем примере переменная `i` нужна только для того, чтобы контролировать число повторений цикла. Тогда подобный цикл можно реализовать следующим образом:

```
            mov ecx, 10     ; Счетчик цикла в ecx
loop_start:
            ; Здесь полезная работа в цикле
            dec ecx         ; Если ecx != 0
            jnz loop_start  ; повторить
```

Такой подход эффективнее, поскольку не требует выполнения команды `CMP` на каждом шаге цикла. В системе команд x86-64 даже есть команда `LOOP` (таблица 1.10), которая семантически эквивалентна последовательности команд `DEC ECX` / `JNZ`. Однако современные компиляторы обычно не используют команду `LOOP`, т. к. она работает медленнее на многих современных процессорах x86, чем последовательность `DEC`/`JNZ`.

Задания для самостоятельной и индивидуальной работы

2.1. Обменять значения целочисленных переменных

1. `x`, `y`, `z` так, чтобы `x >= y >= z`.
2. `x`, `y`, `z` так, чтобы `x < y < z`.
3. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 < x2 < x3 < x4`.
4. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 > x2 > x3 > x4`.
5. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 >= x2 < x3 >= x4`.
6. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 < x2 > x3 < x4`.
7. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 >= x2 >= x3 >= x4`.
8. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 < x2 > x3 > x4`.
9. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 > x2 > x3 < x4`.
10. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 > x2 < x3 < x4`.
11. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 >= x2 < x3 <= x4`.
12. `x1`, `x2`, `x3`, `x4` так, чтобы `x1 > x2 < x3 > x4`.

2.2. Поле шахматной доски определяется парой натуральных чисел, каждое из которых не превосходит 8: первое — номер вертикали, второе — номер горизонтали. Заданы натуральные числа *k, l, m, n*.

1. Определить одного ли цвета поля (*k*, *l*) и (*m*, *n*).
2. На поле (*k*, *l*) расположен слон. Угрожает ли он полю (*m*, *n*)?
3. На поле (*k*, *l*) расположен ферзь. Угрожает ли он полю (*m*, *n*)?
4. На поле (*k*, *l*) расположен конь. Угрожает ли он полю (*m*, *n*)?
5. Можно ли с поля (*k*, *l*) одним ходом ладьи попасть на поле (*m*, *n*)?
6. На поле (*k*, *l*) стоит ладья, на поле (*m*, *n*) — слон. Определить, бьет ли ладья слона, слон — ладью или фигуры не угрожают друг другу.
7. На поле (*k*, *l*) стоит ладья, на поле (*m*, *n*) — ферзь. Определить, бьет ли ладья ферзя, ферзь — ладью или фигуры не угрожают друг другу.
8. На поле (*k*, *l*) стоит ладья, на поле (*m*, *n*) — конь. Определить, бьет ли ладья коня, конь — ладью или фигуры не угрожают друг другу.

9. На поле (k, l) стоит ладья, на поле (m, n) — пешка. Определить, бьет ли ладья пешку, пешка — ладью или фигуры не угрожают друг другу.
10. На поле (k, l) стоит ферзь, на поле (m, n) — слон. Определить, бьет ли ферзь слона, слон — ферзя или фигуры не угрожают друг другу.
11. На поле (k, l) стоит слон, на поле (m, n) — конь. Определить, бьет ли слон коня, конь — слона или фигуры не угрожают друг другу.
12. На поле (k, l) стоит ферзь, на поле (m, n) — конь. Определить, бьет ли ферзь коня, конь — ферзя или фигуры не угрожают друг другу.

2.3. Выполнить задание с циклами.

1. Найдите сумму первых n натуральных чисел, которые являются степенью числа 5.
2. Найдите сумму первых n натуральных чисел, которые делятся на 3.
3. Найти сумму первых n членов геометрической прогрессии с первым членом 1 и знаменателем –2.
4. Найдите сумму первых n натуральных чисел, которые являются полными квадратами.
5. Найдите сумму первых n натуральных чисел, которые являются степенью числа 3.
6. Найдите сумму первых n натуральных чисел, которые делятся на 5.
7. Найдите сумму первых n натуральных чисел, которые делятся на 6.
8. Найдите сумму первых n натуральных чисел, которые делятся на 9.
9. Найдите сумму первых n натуральных чисел, которые делятся на 3 и 5.
10. Найдите сумму первых n натуральных чисел, которые делятся на 3 и 10.
11. Найдите сумму первых n натуральных чисел, которые делятся на 2 или 5.
12. Найдите сумму первых n натуральных чисел, которые делятся на 3 или 5.

2.4. Выполнить задание с циклами.

1. Дано натуральное n. Верно ли, что это число содержит только две одинаковые цифры?
2. Дано натуральное n. Верно ли, что это число содержит ровно три одинаковые цифры?
3. Дано натуральное n. Определить, является ли это число палиндромом.
4. Дано натуральное n. Верно ли, что все цифры числа различны?
5. Дано натуральное n. Верно ли, что это число содержит ровно k одинаковых цифр?
6. Дано натуральное n. Верно ли, что все цифры числа различны?

7. Дано натуральное n. Верно ли, что это число содержит ровно 2 одинаковые цифры?
8. Дано натуральное n. Верно ли, что это число содержит более k одинаковых цифр?
9. Определить, равна ли сумма k первых цифр заданного натурального числа, сумме k его последних цифр.
10. Дано натуральное n. Верно ли, что это число содержит более k одинаковых цифр?
11. Дано натуральное n. Верно ли, что это число содержит k цифр, значения которых меньше заданного m?
12. Дано натуральное n. Верно ли, что это число содержит более k цифр больших суммы первых 2 цифр.

Лабораторная работа № 3

4 МАССИВЫ. СТРОКИ

Цель работы: изучить последовательную обработку массивов данных на языке ассемблера.

Создание массива в MASM заключается в размещении в памяти нескольких значений данных одного типа подряд. Примеры были показаны в разделе «Определение данных в MASM». Для доступа к элементам массива нужно задать его адрес. Считая, что индексы элементов начинаются с нуля (как в C++), адрес элемента массива с индексом `index` есть

```
element_address = base_address + index * element_size
```

Здесь `base_address` — это базовый адрес, т. е. адрес первого элемента массива; `element_size` — размер каждого элемента в байтах.

Пример обращения к последнему элементу массива:

```
      .data
array dword 1,2,3,4,5,6,7
el_sz = TYPE array                ; = 4
      .code
      lea rbx, array              ; базовый адрес
      mov rsi, LENGTHOF array - 1 ; индекс последнего элемента
      mov eax, [rbx + rsi * el_sz]
```

Оператор `TYPE` возвращает размер элемента данных в байтах. Адрес начала массива загружается в регистр `RBX`. В регистр `RSI` загружается индекс элемента, к которому нужно обратиться (в данном случае — последний элемент). Оператор `LENGTHOF` возвращает количество элементов в массиве. В последней команде используется режим адресации «база плюс индекс с масштабированием» (таблица 1.3).

Пример перебора элементов массива при подсчете их суммы:

```
       .data
array  dword 1,2,3,4,5,6,7
       .code
       lea rsi, array          ; указатель на начало массива
       mov rcx, lengthof array ; длина массива
       xor eax, eax            ; обнулить eax
L1:
       add eax, [rsi]          ; добавить элемент к сумме
       add rsi, type dword     ; смещение указателя
       dec rcx
       jnz L1
                               ; сумма в eax
```

Здесь регистр `RSI` играет роль указателя, который последовательно смещается на следующий элемент путем прибавления размера элемента в байтах. В этом

примере используется часто встречающийся быстрый способ обнуления регистра при помощи XOR.

Пример простой сортировки массива:

```
; Пузырьковая сортировка массива
      .data
arr   DWORD 5, 4, 3, 2, 1
Count = ($ - arr) / TYPE arr

      .code
main  PROC
      mov rcx, Count
      dec rcx
L1:   push rcx           ; сохранить счетчик внешнего цикла
      lea rsi, arr       ; ESI указывает на первый элемент
L2:   mov eax, [rsi]     ; взять число из массива
      cmp [rsi+4], eax   ; сравнить со следующим
      jg L3              ; if ([ESI+4] > [ESI]) не обменивать
      xchg eax,[rsi+4]   ; обменять значения соседних
      mov [rsi], eax     ; элементов массива
L3:   add rsi,4          ; сдвинуть указатель на следующий элемент
      loop L2            ; внутренний цикл
      pop rcx            ; восстановить счетчик внешнего цикла
      loop L1            ; повторить внешний цикл
L4:
      ret
main  ENDP
      END
```

Это также пример цикла в цикле. Здесь использована команда LOOP: во-первых, для краткости, во-вторых, чтобы продемонстрировать технику сохранения и восстановления регистра в стеке, если один и тот же регистр управляет внешним и внутренним циклом.

Двумерный массив может располагаться в памяти по строкам или по столбцам. В первом случае формула вычисления адреса элемента будет следующей:

```
elem_addr = base_addr + (row_idx * col_size + col_idx) *elem_size
```

Здесь `base_addr` — это адрес начала массива; `col_size` — количество столбцов (элементов в строке); `elem_size` — размер элементов в байтах; `row_idx` — индекс строки; `col_idx` — индекс столбца.

В следующем примере показано обращение к заданному элементу двумерного массива:

```
        .data
; Пусть имеется таблица чисел из 3 строк и 5 столбцов.
; Расположим элементы таблицы в памяти построчно.
arr2D   dword  1,  2,  3,  4,  5
RowSize = ($ - arr2D) ; размер строки (в байтах)
```

```
        dword  6,  7,  8,  9, 10
        dword 11, 12, 13, 14, 15

; Допустим, нужно обратиться к элементу arr2D[1][2]
i = 1
j = 2

        .code
        lea rbx, arr2D  ; адрес начала таблицы
        add rbx, RowSize * i ; адрес начала i-й строки
        mov rsi, j
        mov eax, [rbx + rsi * TYPE arr2D]  ; EAX = 8
```

В системе команд x86-64 имеется группа команд для работы со строками (таблица 1.11). В терминологии x86-64 под строками понимаются непрерывные последовательности битов, байтов, слов, двойных или четверных слов. Таким образом это не обязательно символьные строки, но произвольные массивы данных. Использование строковых команд может значительно ускорить программы, работающие с массивами.

Далее демонстрируются примеры использования строковых команд: `MOVSx`, `CMPSx`, `SCASx`, `LODSx` и `STOSx`, где суффикс `x` означает `B` (байт), `W` (слово), `D` (двойное слово) или `Q` (четверное слово). Эти команды производят определенные действия с элементами массивов, но не имеют явных операндов. Они неявно используют определенные регистры в качестве операндов:

- `RSI` — индекс источника;
- `RDI` — индекс приемника;
- `RCX` — счетчик;
- регистры `AL`, `AX`, `EAX` и `RAX`;
- флаг направления (DF) из регистра RFLAGS.

Например, команда `MOVSB` (**M**ove **S**tring of **B**ytes) копирует байт из ячейки памяти по адресу `RSI` в ячейку по адресу `RDI`, после чего изменяет `RSI` и `RDI` на 1. Направление изменения контролируется флагом DF. Если флаг сброшен, `RSI` и `RDI` инкрементируются, если установлен — декрементируются. Если перед командой `MOVSB` присутствует префикс `REP` (repeat) описанные действия выполняются, только если `RCX` не равен `0` и далее повторяются пока `RCX` не станет равен `0`.

Аналогично работает команда `MOVSW` (`MOVSD` или `MOVSQ`), только копирует не байт, а слово (двойное слово или четверное слово) и регистры `RSI` и `RDI` изменяются на 2 (4 или 8).

Пример копирования 100 байт из `byteArray1` в `byteArray2`:

```
byteArray1 byte 100 dup (?)
byteArray2 byte 100 dup (?)
```

```
...

cld                             ; направление = вперед
lea rsi, byteArray1             ; указатель на источник
lea rdi, byteArray2             ; указатель на приемник
mov rcx, LENGTHOF byteArray1 ; счетчик rep = 100
rep movsb                       ; копировать байты
```

Команда `CLD` сбрасывает флаг направления (DF).

Пример копирования 100 двойных слов из `byteArray1` в `byteArray2` в обратном порядке:

```
dwordArray1 dword 100 dup (?)
arr_size = $ - dwordArray1
dwordArray2 dword 100 dup (?)

...

std         ; направление = назад
lea rsi, dwordArray1[arr_size - 4]
lea rdi, dwordArray2[arr_size - 4]
mov rcx, lengthof dwordArray1
rep movsd  ; копировать двойные слова
cld         ; вернуть направление по умолчанию
```

Команда `STD` устанавливает флаг направления (DF). Когда работа выполнена рекомендуется очистить флаг направления.

Команды `CMPSx` (compare string) работают подобно `MOVSx`, но вместо копирования элементов строк, сравнивают их (как `CMP`). При использовании префикса `REPE` (`REPNE`) сравнения элементов строк будут производиться пока они равны (не равны) и `RCX > 0`.

Команда `STOSx` (store string) сохраняет в ячейку памяти с адресом `RDI` значение регистра-аккумулятора. Так, например, можно заполнить массив постоянным значением 0:

```
cld
lea rdi, wordArr
mov rcx, arrLen
xor rax, rax        ; обнулить RAX
rep stosw
```

Команда `LODSx` (load string) наоборот загружает значение по адресу `RSI` в аккумулятор.

Следующий пример показывает, как с помощью `LODSx` и `STOSx` выполнить преобразование массива, применяя произвольную функцию f к каждому элементу:

```
array dword 1,2,3,4,5,6,7,8,9,10

      ...

      cld
      lea rsi, array          ; Загрузить адрес массива в RSI
      mov rdi, rsi            ; и в RDI
      mov rcx, lengthof array ; В RCX – количество элементов

L1:   lodsd                   ; Загрузить в EAX следующий элемент
      ; EAX = f(EAX)          ; Преобразовать
      stosd                   ; Сохранить обратно в массив
      dec rcx
      jnz L1
```

Задания для самостоятельной и индивидуальной работы

3.1. Одномерные массивы.

1. Найти количество тех элементов массива, которые встречаются в нем ровно два раза.
2. Найти сумму элементов массива, расположенных до минимального.
3. Найти сумму тех элементов массива, которые являются степенью числа 5.
4. Найти сумму тех элементов массива, которые встречаются по одному разу.
5. Найти сумму элементов массива, расположенных до максимального.
6. Найти сумму тех элементов массива, индексы которых делятся на 2 или 3.
7. Найти сумму элементов массива, расположенных между минимальным и максимальным значениями.
8. Найти сумму тех элементов массива, которые делятся на 3 и 5.
9. Найти сумму 3 наибольших элементов массива.
10. Найти сумму тех элементов массива, которые делятся на 3.
11. Найти сумму 5 наименьших элементов массива.
12. Найти сумму элементов массива, расположенных после максимального.
13. Найти сумму тех элементов массива, которые делятся на 3 или 5.
14. Найти сумму элементов массива, расположенных после минимального.
15. Найти сумму тех элементов массива, которые являются полными квадратами.
16. Найти сумму каждого третьего элемента массива.
17. Найти сумму тех элементов массива, которые делятся на 5.

18. Найти сумму элементов массива, расположенных до максимального из отрицательных элементов.
19. Найти количество тех элементов массива, которые встречаются в массиве более одного раза.
20. Найти сумму элементов массива, расположенных после минимального из положительных элементов.
21. Найти наибольшее из чисел массива, встречающихся в нем ровно один раз.
22. Найти сумму чисел, расположенных между минимальным положительным и максимальным отрицательным элементами.
23. Найти сумму тех элементов массива, которые делятся на 3 и 10.
24. Найти наименьшее из чисел массива, встречающихся в нем более одного раза.

3.2. Одномерные массивы.

1. Получить за один просмотр массив $C(K)$, упорядоченный по возрастанию, путем слияния массивов $A(N)$ и B(M), упорядоченных по возрастанию ($K = N + M$).
2. Вычислить скалярное произведение заданных векторов a и b длины n.
3. Дан массив целых чисел, содержащий n элементов. Получить массив, в котором записаны сначала все отрицательные числа и нули, затем все положительные числа, сохраняя порядок следования.
4. Отсортировать массив методом выбора.
5. Заданы два одномерных массива $X(n)$, $Y(m)$, причем $0 \le Y_i < n$; $m \le n$, $Y_i \ne Y_j$. Вычислить сумму тех элементов массива X, индексы которых совпадают со значениями элементов массива Y.
6. Отсортировать массив методом вставки.
7. Из двух массивов $A(N)$ и $B(M)$, упорядоченных по возрастанию, получить за один просмотр массив $C(K)$, также упорядоченный по возрастанию, в который включить пересечение элементов двух исходных массивов.
8. Найти наименьшее среди чисел первой последовательности, не входящих во вторую.
9. Дан массив целых чисел, содержащий n элементов. Получить массив, в котором записаны сначала все положительные числа, затем все отрицательные числа и нули, сохраняя порядок следования.
10. Из двух массивов $A(N)$ и $B(M)$, упорядоченных по возрастанию, получить за один просмотр массив $C(K)$, также упорядоченный по возрастанию, в

который включить элементы первого массива, исключив из них элементы второго массива.

11. Найти наибольшее среди чисел первой последовательности, входящих во вторую.
12. Заданы два одномерных массива $X(n)$, $Y(m)$. Причем $0 \le Y_i < n$; $m \le n$, $Y_i \ne Y_j$. Вычислить сумму тех элементов массива X, индексы которых не совпадают со значениями элементов массива Y.

3.3. Двумерные массивы.

1. Транспонировать заданную матрицу.
2. Вычислить сумму произведений элементов строк заданной матрицы.
3. Для заданной матрицы A получить матрицу $A\,A^T$.
4. Упорядочить строки матрицы по убыванию их первых элементов.
5. Поменять местами строку, содержащую элемент с наибольшим значением в матрице, со строкой, содержащей элемент с наименьшим значением.
6. Вычислить сумму произведений элементов столбцов заданной матрицы.
7. Для заданных матриц A, B получить матрицу $C = A\,B - B\,A$.
8. Создать массив из минимальных значений столбцов заданной матрицы.
9. Упорядочить строки матрицы по возрастанию их последних элементов.
10. Найти максимальный и минимальный элементы среди стоящих на главной и побочной диагоналях матрицы и обменять их местами.
11. Вычислить произведение сумм элементов строк заданной матрицы.
12. Дана матрица A размерности $n \times n$. Получить матрицу $B = E - A + A^2 / 2$.
13. Создать массив из максимальных значений строк заданной матрицы.
14. Упорядочить столбцы матрицы по убыванию их первых элементов.
15. Среди строк заданной матрицы, содержащих только нечетные элементы, найти строку с максимальной суммой элементов.
16. Вычислить произведение сумм элементов столбцов заданной матрицы.
17. Дана матрица A размерности $n \times n$. Получить матрицу $B = E - A^2 / 2$.
18. Упорядочить столбцы матрицы по возрастанию их последних элементов.
19. Найти сумму элементов того столбца, в котором находится наименьший элемент матрицы.
20. Создать массив из минимальных значений строк заданной матрицы.
21. Дана матрица A размерности $n \times n$. Получить матрицу $B = A - A^3 / 6$.
22. Создать массив из максимальных значений столбцов заданной матрицы.

23. Для заданных матрицы A и вектора b, получить вектор $c = A^2 b$.
24. Для заданных матрицы A и вектора x получить число $x^T A x$.

Лабораторная работа № 4

5 ПРОЦЕДУРЫ В MASM

Цель работы: изучить механизм вызова процедур, передачу параметров и использование стека; научиться интегрировать низкоуровневый ассемблерный код в высокоуровневые программы на C/C++.

Процедура (подпрограмма, функция) в MASM определяется с помощью директив `PROC` и `ENDP`:

```
proc_name PROC
    ; ... инструкции ...
        ret
proc_name ENDP
```

Здесь `proc_name` — имя процедуры (любой допустимый и уникальный идентификатор MASM). Инструкция `ret` выполняет выход из процедуры в точку вызова. Вызов процедуры осуществляется командой `call` (таблица 1.10).

Команда `call` сохраняет в стек значение регистра `RIP` (т. е. адрес следующей команды) и записывает в `RIP` адрес указанной в операнде процедуры (т. е. передает ей управление). Команда `ret` забирает с вершины стека 8-байтное значение, которое использует как адрес возврата, на который выполняет переход.

Передачу аргументов в языке ассемблера удобно производить через регистры общего назначения. В следующем примере объявлено две процедуры. Процедура `main` помещает адрес массива в регистр `rsi` и количество элементов — в регистр `rcx`. Затем вызывает функцию `sum_arr`, которая вычисляет сумму элементов массива, оставляет результат в `eax` и возвращает управление функции `main`. После возврата, сумма копируется из регистра `eax` в переменную `sum`.

```
        .data

array   dword 1,2,3,4,5
sum     dword ?

        .code

; sum_arr
; Вычисляет сумму 32-битных элементов массива
; IN : RSI - указатель на начало массива
;      RCX - количество элементов
; OUT: EAX - сумма

sum_arr PROC
        xor eax,eax    ; обнулить eax
L1:     add eax,[rsi]
        add rsi, 4
```

```
        dec rcx
        jnz L1
        ret
sum_arr ENDP

main    PROC
        lea  rsi, array           ; аргумент
        mov  rcx, lengthof array  ; аргумент
        call sum_arr              ; вычислить сумму
        mov  sum,eax              ; сохранить результат
        ret
main    ENDP

        END
```

Хорошей практикой считается сохранение изменяемых функцией регистров (за исключением регистра, в который записывается возвращаемое значение), так чтобы вызывающая функция могла продолжить свое выполнение с прежними значениями регистров. Для этого удобно использовать стек. Так процедуру суммирования элементов массива следовало бы изменить следующим образом:

```
sum_arr PROC
        push rsi
        push rcx

        xor eax,eax
L1:     add eax,[rsi]
        add rsi, 4
        dec rcx
        jnz L1

        pop rcx
        pop rsi
        ret
sum_arr ENDP
```

Важный момент: сохраненные значения регистров извлекаются из стека в порядке обратном добавлению в стек. Тот же результат можно получить с помощью оператора `USES` после директивы `PROC`:

```
sum_arr PROC USES rsi rcx
        xor eax,eax
L1:     add eax,[rsi]
        add rsi, 4
        dec rcx
        jnz L1
        ret
sum_arr ENDP
```

5.1 Активационная запись. Соглашения о вызове процедур

С вызовом процедуры сопряжена определенная совокупность данных: передаваемые в процедуру аргументы, адрес возврата, сохраняемые значения регистров, локальные переменные. Структура данных, которая хранит перечисленную информацию об отдельном вызове процедуры, называется *активационная запись*. Программа создает ее при каждом вызове (активации) процедуры и уничтожает при возврате. Активационные записи обычно создаются в стеке. Часть стека, которую занимает одна активационная запись, называется *кадр стека*.

Контракт между вызывающей и вызываемой процедурой, регламентирующий порядок создания и уничтожения активационной записи, называется *соглашение о вызовах*. В самодостаточной программе на языке ассемблера программист может устанавливать свои соглашения о вызовах. Однако, если программа должна взаимодействовать со сторонними библиотеками, приходится соблюдать соглашения, принятые компиляторами соответствующих языков и конкретной ОС.

Microsoft использует единообразную схему передачи параметров и вызова процедур в 64-разрядных программах, известную как соглашение о вызовах Microsoft x64 (Microsoft x64 Calling Convention). Это соглашение применяется компиляторами C/C++, а также интерфейсом прикладного программирования Windows (API). Основные детали этого соглашения:

1. Первые четыре параметра, передаваемые процедуре, размещаются в регистрах RCX, RDX, R8 и R9 — именно в таком порядке. Если передаётся только один параметр, он помещается в RCX. Второй — в RDX, и так далее. Дополнительные параметры помещаются в стек в порядке справа налево.
2. Вызывающий код обязан выделить на стеке как минимум 32 байта. Это так называемое теневое или домашнее пространство (shadow/home space) может использоваться вызываемой процедурой для сохранения параметров из регистров.
3. Возвращаемое значение (целочисленное) вызываемая процедура сохраняет в регистре RAX (или его подрегистрах).
4. Непосредственно перед вызовом подпрограммы (CALL) указатель стека (RSP) должен быть выровнен по границе в 16 байт (т. е. адрес в RSP должен быть кратен 16).

5.2 Вызов функции C/C++ из программы MASM

Для вывода на экран из программы на ассемблере можно воспользоваться стандартной функцией C/C++ `printf`. Следующий код иллюстрирует передачу пяти параметров в функцию `printf` (строки форматирования и четырех целочисленных значений разных размеров) в соответствии с соглашениями о вызо-

вах Microsoft x64. Первые четыре аргумента передаются через соответствующие регистры. Пятый аргумент помещается в стек.

```
includelib ucrt.lib
includelib legacy_stdio_definitions.lib

        .data

fmtStr  byte  “Arg2: %d.”,10
        byte  “Arg3: %d.”,10
        byte  “Arg4: %d.”,10
        byte  “Arg5: %d.”,10,0

a2b     byte  2
a3w     word  3
a4d     dword 4
a5q     qword 5

        .code

externdef printf:PROC

main    PROC
; Сейчас RSP = xxx8h
; Если число аргументов четное:
;       sub   rsp, 8  ; раскомментировать для выравнивания

; Поместить в стек аргументы сверх первых 4-х (справа налево):
        push  a5q         ; 5-й аргумент

; Выделить теневое пространство для первых 4-х аргументов:
        sub   rsp,32

; Поместить первые 4 аргумента в регистры:
        mov   r9d, a4d     ; 4-й аргумент
        movsx r8,  a3w     ; 3-й аргумент
        movsx rdx, a2b     ; 2-й аргумент
        lea   rcx, fmtStr  ; 1-й аргумент (строка форматирования)

; Перед вызовом процедуры указатель стека
; должен быть выровнен по адресам кратным 16
        call  printf

; Восстановить стек:
        add   rsp, 32      ; очистить теневое пр-во
        add   rsp, 8       ; удалить 5-й аргумент
; Восстановить выравнивание стека если необходимо:
;       add   rsp, 8

        xor   rax,rax
```

```
        ret
main    ENDP
        END
```

Необходимые библиотеки подключаются с помощью директивы ассемблера `includelib`. Библиотека *ucrt.lib* — это библиотека Universal C Runtime, содержащая реализацию стандартных функций C, включая `printf`, `malloc`, `fopen` и т. д. Дополнительная библиотека *legacy_stdio_definitions.lib* предоставляет определения устаревших имен функций, которые использовались в более ранних версиях CRT и всё ещё требуются при линковке из MASM или других низкоуровневых языков.

Определенные во внешних модулях идентификаторы объявляются с помощью директивы `externdef` в следующей форме:

```
externdef имя:тип
```

Здесь `имя` — идентификатор, а `тип` задает его тип данных, например, `BYTE`, `WORD`, `DWORD`, `PROC` (для процедур).

Замечания по поводу выравнивания стека. Непосредственно перед вызовом процедуры `main` стек был выровнен. Сразу после входа в `main`, указатель стека кратен 8, но не 16, т. к. в стек был помещен адрес возврата. Выделение в стеке места для пяти аргументов (5 * 8 байт) вновь сделает стек выровненным по границе в 16 байт непосредственно перед командой `call printf`. При передаче, например, шести аргументов, для выравнивания стека перед вызовом `printf` потребуется дополнительно уменьшить `rsp` на 8, а после возврата соответственно увеличить на 8.

5.3 Вызов ассемблерной процедуры из программы на C/C++

Более распространена обратная ситуация, когда основная программа пишется на C++, а отдельные ее части могут быть реализованы на ассемблере (например, для оптимизации производительности).

В следующем примере показан код программы на C++, которая вызывает ассемблерную функцию `calc_asm` и выводит на экран результат.

```
// main.cpp
// Простая программа, которая
// вызывает ассемблерную функцию

#include <iostream>
#include <cstdint>  // для имен типов int8_t и т.д.

extern "C" {
    int32_t calc_asm(
        int8_t a1, int16_t a2, int32_t a3, int64_t a4,
        int8_t b1, int16_t b2, int32_t b3, int64_t b4,
        int64_t& quo, int64_t& rem );
}
```

```
int main()
{
    int8_t  a1 = -12;
    int16_t a2 = 1000;
    int32_t a3 = -631;
    int64_t a4 = 1000000000;
    int8_t  b1 = 1;
    int16_t b2 = 2;
    int32_t b3 = -3;
    int64_t b4 = 56;
    int64_t quotient_1, reminder_1;
    int64_t quotient_2, reminder_2;

    if (calc_asm(a1, a2, a3, a4,
                 b1, b2, b3, b4,
                 quotient_2, reminder_2))
    {
        quotient_1 = ((int64_t)a1 + a2 + a3 + a4)
                   / ((int64_t)b1 + b2 + b3 + b4);
        reminder_1 = ((int64_t)a1 + a2 + a3 + a4)
                   % ((int64_t)b1 + b2 + b3 + b4);

        std::cout <<   "a1 = " << (int)a1
                  << ", a2 = " <<      a2
                  << ", a3 = " <<      a3
                  << ", a4 = " <<      a4 << "\n";
        std::cout <<   "b1 = " << (int)b1
                  << ", b2 = " <<      b2
                  << ", b3 = " <<      b3
                  << ", b4 = " <<      b4 << "\n";
        std::cout <<   "quotient_1 = " << quotient_1
                  << ", reminder_1 = " << reminder_1 << "\n";
        std::cout <<   "quotient_2 = " << quotient_2
                  << ", reminder_2 = " << reminder_2 << "\n";
    }
    return 0;
}
```

Объявление функции `calc_asm` заключено в блок `extern "C"` для того, чтобы предотвратить *декорирование имен* компилятором C++. Дело в том, что компилятор C++ в отличие от компилятора C модифицирует имена функций, добавляя в них информацию о типах параметров (для реализации перегрузки функций).

Функция `calc_asm` реализована на ассемблере в отдельном файле:

```
; calc.asm
; int32_t calc_asm (
;   int8_t a1, int16_t a2, int32_t a3, int64_t a4,
;   int8_t b1, int16_t b2, int32_t b3, int64_t b4,
```

```
;   int64_t &quo, int64_t &rem
; );

option casemap:none

        .code

calc_asm PROC
 ; num = a1 + a2 + a3 + a4
         movsx  rax,cl
         movsx  rdx,dx
         add    rax,rdx              ; rax = a1 + a2
         movsxd r8, r8d
         add    r9, r8               ; r9 = a3 + a4
         add    rax,r9               ; rax = a1 + a2 + a3 + a4

 ; den = b1 + b2 + b3 + b4
         movsx  r8,byte ptr [rsp+40]
         movsx  r9,word ptr [rsp+48]
         add    r8,r9                ; r8 = b1 + b2
         movsxd r9,dword ptr [rsp+56]
         add    r9,[rsp+64]          ; r9 = b3 + b4
         add    r8,r9                ; r8 = b1 + b2 + b3 + b4
         jz     DivBy0

       ; num / den
         cqo                         ; rdx:rax = a1 + a2 + a3 + a4
         idiv   r8
         mov    rcx,[rsp+72]
         mov    [rcx],rax            ; &quo = num / den
         mov    rcx,[rsp+80]
         mov    [rcx],rdx            ; &rem = num % den

         mov    eax,1
         ret                 ; Вернуть 1 (деление успешно)

DivBy0:  xor    eax,eax
         ret                 ; Вернуть код ошибки 0 (деление на 0)
calc_asm ENDP

         END
```

По умолчанию MASM преобразует все идентификаторы к верхнему регистру. Директива `option casemap:none` отключает это поведение, делая MASM чувствительным к регистру символов. В данном примере это необходимо, т. к. компилятор C++ различает регистр символов.

На рисунке 5.1 показано содержимое стека сразу после входа в функцию `calc_asm`. Функция вычисляет частное и остаток от деления суммы первых четырех аргументов на сумму следующих четырех аргументов. Перед суммиро-

ванием аргументы короче 64 бит расширяются по знаку с помощью инструкций `movsx` или `movsxd` (таблица 1.5).

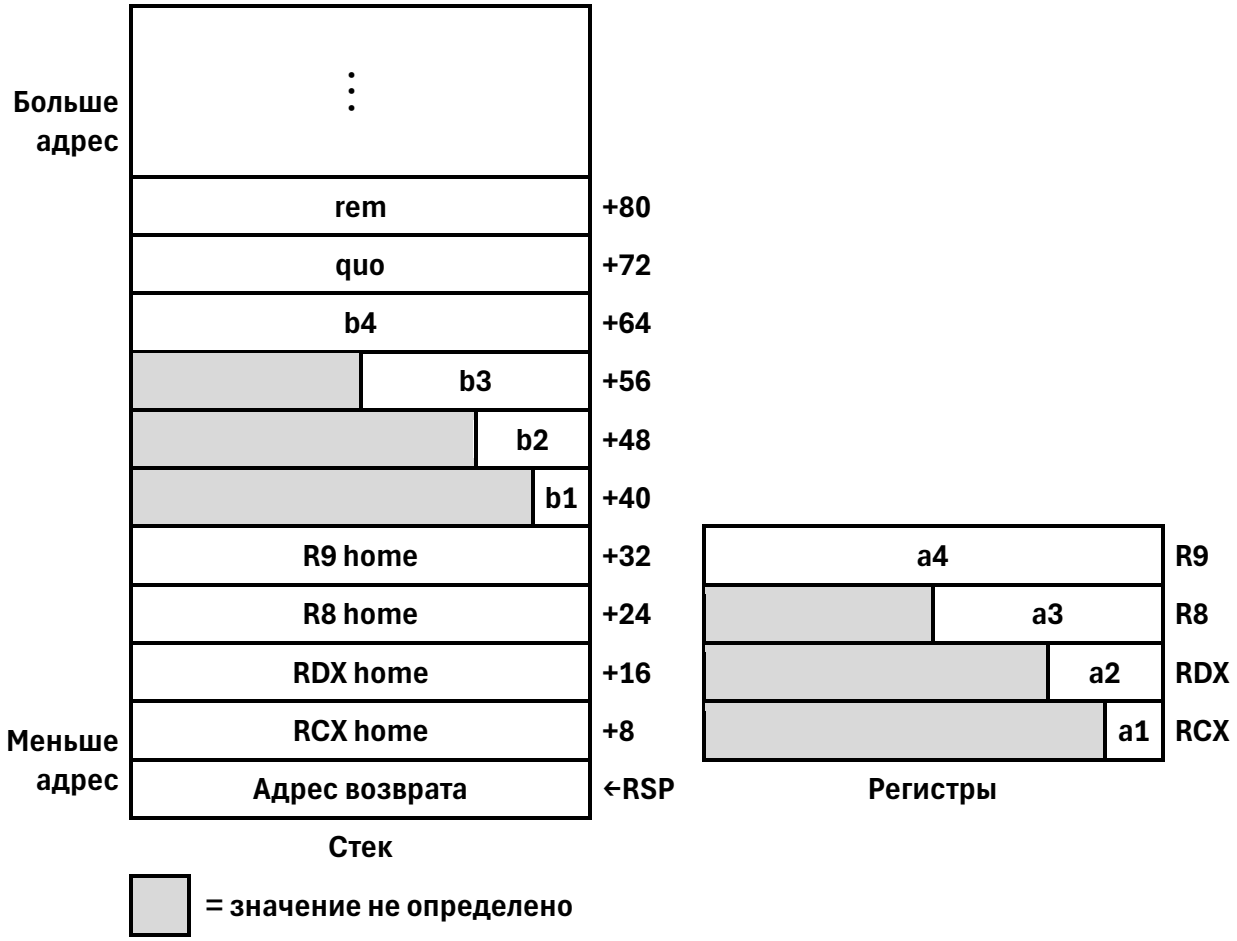

Рисунок 5.1. — Регистры с аргументами и кадр стека при входе в calc_asm()

Соглашение о вызовах Microsoft x64 различает *волатильные* и *неволатильные* регистры. Регистры RAX, RCX, RDX, R8, R9, R10 и R11 считаются *волатильными*. Вызываемая процедура не обязана сохранять их значения. Регистры RBX, RBP, RDI, RSI, RSP, R12–R15 считаются *неволатильными* и должны сохраняться вызываемой подпрограммой (таблица 1.1). Кроме того, подпрограмма может создавать на стеке свои локальные переменные. Сохраненные неволатильные регистры и локальные переменные должны располагаться ниже адреса возврата в каре стека (рисунок 5.1).

Функция, которая не изменяет неволатильные регистры называется *листовой функцией*. Код нелистовых функций содержит стандартную входную последовательность команд (*пролог*) и стандартную завершающую последовательность (*эпилог*), для управления кадром стека. Пролог может включать следующие действия:

```
someFun PROC
; Пролог

; Сохранить регистры с аргументами в теневую область (опционально)
```

```
        mov [rsp+8], rcx
        mov [rsp+16], rdx
        mov [rsp+24], r8
        mov [rsp+32], r9

; Сохранить используемые неволатильные регистры,
; например:
        push rbp
        push r15
        push r14

; Выделить место для локальных переменных (опционально)
        sub rsp, fixed_alloc_size

; Инициализировать локальные переменные
        ...
        ...
```

Тогда соответствующий эпилог при выходе из функции будет иметь вид:

```
        ...
; Эпилог

; Удалить локальные переменные
        add rsp, fixed_alloc_size

; Восстановить использованные неволатильные регистры,
; например:
        pop r14
        pop r15
        pop rbp

        ret
someFun ENDP
```

Доступ к локальным переменным или аргументам в данном случае производится относительно RSP. Однако в качестве базового адреса можно выбрать любую удобную позицию в кадре стека. Адрес этой позиции сохраняется обычно в RBP. Часто в качестве базового адреса кадра выбирают адрес сохраненного старого значения RBP.

Задания для самостоятельной и индивидуальной работы

4.1 Оформить одну программу из лабораторных работ 1–3 с использованием простых процедур.

4.2 Добавить в одну программу из лабораторных работ 1–3 вывод результата на экран с помощью функции printf.

4.3 Оформить одну программу из лабораторных работ 1–3 в виде ассемблерных функции, вызываемой из программы на C++. Ввод исходных данных и вывод результатов выполнить из программы на C++.

Лабораторная работа № 5

6 АРИФМЕТИКА С ПЛАВАЮЩЕЙ ТОЧКОЙ

Цель работы: научиться выполнять вычисления с плавающей точкой с использованием скалярных операций систем команд SSE и AVX.

Традиционно вычисления с плавающей точкой на платформе x86 обеспечивала система команд x87, которая изначально реализовывалась отдельными сопроцессорами (например, Intel 8087), а затем — блоком вычислений с плавающей точкой (FPU, Floating-Point Unit) внутри самого центрального процессора (начиная с модели Intel 486DX). Архитектура x87 предоставляет 8 регистров разрядности 80 бит, доступ к которым организован по принципу стека.

Стековая архитектура x87 усложняет распределение регистров, ограничивает параллелизм и мешает эффективному внеочередному выполнению инструкций в современных процессорах.

Современной более эффективной альтернативой x87 являются наборы команд SSE (Streaming SIMD Extensions) и AVX (Advanced Vector Extensions). Несмотря на то, что они в первую очередь предназначены для векторной обработки (т. е. для параллельных вычислений класса SIMD, Single Instruction, Multiple Data — один поток команд, много потоков данных), они также предоставляют инструкции для скалярных операций с плавающей точкой. Использование данных инструкций является стандартом для современных платформ x86-64. При этом система команд x87 все еще поддерживается в процессорах x86 в целях обратной совместимости со старым ПО.

Регистры SSE/AVX. Архитектура SSE определяет 16 регистров XMM0–XMM15 длиной 128 бит и поддерживает два типа данных с плавающей точкой: одинарной точности (сохраняются в младшие 32 бита регистров XMM) и двойной точности (сохраняются в младшие 64 бита регистров XMM).

Архитектура AVX определяет 16 регистров YMM0–YMM15 длиной 256 бит. Регистры XMM — это младшие 128 бит соответствующих регистров YMM. Для скалярных вычислений по-прежнему необходимо обращаться к регистрам XMM, но использование команд AVX может иметь ряд преимуществ.

Соглашения о вызовах подпрограмм. В главе 5 рассматривались соглашения о вызове функций Microsoft x64. Следует дополнить данные соглашения на тот случай, когда аргументы или возвращаемое значение являются числами с плавающей точкой. Согласно требованиям Microsoft x64 ABI (Application Binary Interface — двоичный интерфейс приложений) первые четыре аргумента с плавающей точкой передаются в процедуру через регистры XMM0–XMM3. Остальные помещаются в стек в обратном порядке. Если среди аргументов есть как целочисленные, так и с плавающей точкой, то правила следующие.

Первый параметр помещается в RCX или XMM0, второй параметр — в RDX или XMM1 и т. д. Например, если первый параметр является числом с плавающей точкой, он записывается в XMM0. Если при этом второй параметр

является целым числом, то он записывается в RDX, в то время как RCX не используется. Максимальное количество параметров, которые можно передать в регистрах — четыре. Возвращаемое значение с плавающей точкой функция обязана оставлять в регистре XMM0.

6.1 Основные скалярные инструкции SSE

Названия команд для скалярных операций с плавающей точкой оканчиваются суффиксами SS (**S**calar **S**ingle-precision) для операций над скалярными операндами одинарной точности или SD (**S**calar **D**ouble-precision) для операций над скалярными операндами двойной точности. Это отличает их от аналогичных команд с суффиксами PS (**P**acked **S**ingle-precision) и PD (**P**acked **D**ouble-precision) для векторных операций над упакованными операндами, которые будут рассмотрены в главе 7.

В таблице 6.1 представлены скалярные команды SSE для перемещения и преобразования данных. Все команды двухоперандные: первый операнд — приемник, второй — источник. В таблице использованы стандартные обозначения для разных видов операндов: r — один из регистров общего назначения, m — операнд в памяти; xmm — один из 128-разрядных регистров SSE. Числа обозначают разрядность операнда. Косой чертой обозначается альтернатива. Например, запись r/m32 означает один из 32-разрядных регистров общего назначения или область памяти размером 32 бита.

Таблица 6.1 — Команды перемещения и преобразования данных SSE

Команда	**Описание**

MOVSS xmm, m32
MOVSS m32, xmm
MOVSD xmm, m64
MOVSD m64, xmm
	Копирует значение с плавающей точкой.

MOVD xmm, r/m32
MOVD r/m32, xmm
MOVQ r/m64, xmm
MOVQ xmm, r/m64
MOVQ xmm1, xmm2
	Копирует данные без преобразования формата.

CVTSI2S[S|D] xmm, r/m64	Преобразовать целое в значение с плавающей точкой.
CVTSS2SI r64, xmm/m32 CVTSD2SI r64, xmm/m64	Преобразовать значение с плавающей точкой в целое.
CVTTSS2SI r32/64, xmm/m32 CVTTSD2SI r32/64, xmm/m64	Преобразовать с усечением значение с плавающей точкой в целое.
CVTSS2SD xmm, xmm/m32	Преобразовать float в double.
CVTSD2SS xmm, xmm/m64	Преобразовать double в float.

В таблице 6.2 приведены команды выполняющие арифметические операции над скалярными значениями с плавающей точкой. Все команды использу-

ют два операнда. Для бинарных операций регистр-приемник также является первым операндом-источником:

```
dst = dst <операция> src
```

где `src` — источник, `dst` — приемник.

Таблица 6.2 — Арифметические команды SSE

Команда	**Операция**
ADDSS xmm, xmm/m32 ADDSD xmm, xmm/m64	Сложение.
SUBSS xmm, xmm/m32 SUBSD xmm, xmm/m64	Вычитание.
MULSS xmm, xmm/m32 MULSD xmm, xmm/m64	Умножение.
DIVSS xmm, xmm/m32 DIVSD xmm, xmm/m64	Деление.
MINSS xmm, xmm/m32 MINSD xmm, xmm/m64	Минимум.
MAXSS xmm, xmm/m32 MAXSD xmm, xmm/m64	Максимум.
SQRTSS xmm, xmm/m32 SQRTSD xmm, xmm/m64	Приближенное значение квадратного корня.
RCPSS xmm, xmm/m32	Приближенное обратное значение.
RSQRTSS xmm, xmm/m32	Приближенное обратное значение квадратного корня.

Для сравнения чисел с плавающей точкой расширение SSE предоставляет инструкции CMPS[S|D], которые имеют следующий формат:

```
CMPSS xmm, xmm/m32, imm8
CMPSD xmm, xmm/m64, imm8
```

Они сравнивают значения с плавающей точкой в двух первых операндах. Результат сравнения записывается в младшую часть первого операнда в виде маски из всех битов 1 (true) или всех битов 0 (false). Младшие биты третьего непосредственного операнда `imm8` задают способ сравнения, как показано в таблице 6.3. Там же показаны двухоперандные псевдокоманды сравнения, предлагаемые ассемблером MASM, которые автоматически подставляют подходящий третий операнд. Два значения с плавающей точкой *несравнимы* или *неупорядочены* (unordered), если хотя бы одно из них имеет значение NaN (Not-a-Number).

Таблица 6.3. — Псевдокоманды сравнения чисел с плавающей точкой одинарной точности и их реализация

Псевдокоманда	Реализация	Операция
`CMPEQSS l, r`	`CMPSS l, r, 0`	`l == r`
`CMPLTSS l, r`	`CMPSS l, r, 1`	`l < r`
`CMPLESS l, r`	`CMPSS l, r, 2`	`l <= r`
`CMPUNORDSS l, r`	`CMPSS l, r, 3`	несравнимо
`CMPNEQSS l, r`	`CMPSS l, r, 4`	`l != r`
`CMPNLTSS l, r`	`CMPSS l, r, 5`	`l >= r`
`CMPNLESS l, r`	`CMPSS l, r, 6`	`l > r`
`CMPORDSS l, r`	`CMPSS l, r, 7`	сравнимо

Аналогичные команды сравнения существуют для чисел с плавающей точкой двойной точности (имена команд вместо суффикса SS содержат суффикс SD). Данные команды не устанавливают флаги как целочисленная команда CMP. Вместо этого они записывают маску из всех битов 1 (true) или всех 0 (false) в регистр назначения. Если необходимо выполнить ветвление на основе этих логических значений, можно переместить регистр XMM с сохраненной маской в регистр общего назначения и проверить этот регистр на ноль с помощью команды TEST:

```
cmpeqsd xmm0, xmm1
movd    eax, xmm0   ; Переместить true/false в EAX
test    eax, eax    ; Проверить на true/false
jnz     xmm0EQxmm1  ; Выполнить переход если xmm0 == xmm1
```

Для скалярного кода проще использовать команды сравнения COMIS[S|D] и UCOMIS[S|D], которые изменяют регистр EFLAGS напрямую:

```
[U]COMISS xmm, xmm/m32  ; Unordered Compare Scalar Single
[U]COMISD xmm, xmm/m64  ; Unordered Compare Scalar Double
```

Эти команды сравнивают младшие 32 или 64 бита операндов и устанавливают флаги ZF, PF и CF регистра EFLAGS, которые можно проверить стандартными условными командами (например, JE или JA), как показано в таблице 6.4.

Команды COMIS[S|D] и UCOMIS[S|D] различаются тем, как они обрабатывают случаи с несравнимыми операндами. Если один из операндов имеет значение NaN, команды без префикса U (**U**nordered) могут генерировать исключение Invalid Operation. Команды с префиксом U при наличии операнда NaN не генерируют исключение, а просто устанавливают PF = 1.

Таблица 6.4. — Флаги состояния и коды условий после выполнения команд UCOMIS[S|D]

Условие	Код условия (cc)	Состояние флагов

==	E / Z	ZF == 1
<	B	CF == 1
<=	BE	CF == 1 || ZF == 1
несравнимо	P	PF == 1
!=	NE / NZ	ZF == 0
>=	AE	CF == 0
>	A	CF == 0 && ZF == 0

6.2 Скалярные инструкции AVX

Набор команд AVX содержит аналоги рассмотренных инструкций SSE. Названия соответствующих инструкций AVX отличаются от оригинальных команд SSE наличием префикса V (**V**ector). Для многих операций вводится формат с тремя операндами. В таблице 6.5 приведены аналоги команд из таблицы 6.1 для перемещения и преобразования данных.

Таблица 6.5 — Команды перемещения и преобразования данных AVX

Команда	**Описание**
VMOVSS xmm, m32 VMOVSS m32, xmm VMOVSS xmm1, xmm2, xmm3 VMOVSD xmm, m64 VMOVSD m64, xmm VMOVSD xmm1, xmm2, xmm3	Копирует значение с плавающей точкой.
VMOVD xmm, r/m32 VMOVD r/m32, xmm VMOVQ r/m64, xmm VMOVQ xmm, r/m64 VMOVQ xmm1, xmm2	Копирует данные без преобразования формата.
VCVTSI2SS xmm1, xmm2, r/m64 VCVTSI2SD xmm1, xmm2, r/m64	Преобразовать целое в значение с плавающей точкой.
VCVTSS2SI r64, xmm/m32 VCVTSD2SI r64, xmm/m64	Преобразовать значение с плавающей точкой в целое.
VCVTTSS2SI r64, xmm/m32 VCVTTSD2SI r64, xmm/m64	Преобразовать с усечением значение с плавающей точкой в целое.
VCVTSS2SD xmm1, xmm2, xmm3/m32	Преобразовать float в double.
VCVTSD2SS xmm1, xmm2, xmm3/m64	Преобразовать double в float.

В таблице 6.6 показаны аналоги арифметических команд из таблицы 6.2. Все они принимают по три операнда. Например, команды сложения имеют следующий формат и семантику:

```
VADDSS xmm1, xmm2, xmm3/m32  ; xmm1 = xmm2 + xmm3/m32
```

```
VADDSD xmm1, xmm2, xmm3/m64  ; xmm1 = xmm2 + xmm3/m64
```

Таким образом, операнды-источники не модифицируются. Это исключает необходимость дополнительных перемещений данных между регистрами и может сократить число команд в программе и повысить производительность.

Таблица 6.6 — Арифметические команды AVX

Команда	**Операция**
VADDSS xmm1, xmm2, xmm3/m32 VADDSD xmm1, xmm2, xmm3/m64	Сложение
VSUBSS xmm1, xmm2, xmm3/m32 VSUBSD xmm1, xmm2, xmm3/m64	Вычитание
VMULSS xmm1, xmm2, xmm3/m32 VMULSD xmm1, xmm2, xmm3/m64	Умножение
VDIVSS xmm1, xmm2, xmm3/m32 VDIVSD xmm1, xmm2, xmm3/m64	Деление
VMINSS xmm1, xmm2, xmm3/m32 VMINSD xmm1, xmm2, xmm3/m64	Минимум
VMAXSS xmm1, xmm2, xmm3/m32 VMAXSD xmm1, xmm2, xmm3/m64	Максимум
VSQRTSS xmm1, xmm2, xmm3/m32 VSQRTSD xmm1, xmm2, xmm3/m64	Квадратный корень младшего элемента из третьего операнда сохраняет в xmm1. Старшая часть xmm2 копируется в соответствующую часть xmm1.
VRCPSS xmm1, xmm2, xmm3/m32	Обратное значение элемента float из xmm3/m32 сохраняет в xmm1. Старшая часть xmm2 копируется в соответствующую часть xmm1.
VRSQRTSS xmm1, xmm2, xmm3/m32	Обратное значение квадратного корня элемента float из xmm3/m32 сохраняет в xmm1. Старшая часть xmm2 копируется в соответствующую часть xmm1.

Аналоги команд сравнения COMIS[S|D] и UCOMIS[S|D] в AVX имеют префикс V в названии, но так же, как и оригинальные команды, используют два операнда.

Кроме того, система команд AVX предлагает также иные команды для скалярных операций, не имеющие аналогов в SSE. Например, в версии AVX2 доступны инструкции FMA (Fused Multiply-Add), которые позволяют вычислять одно из следующие выражений одной командой:

```
a =  (b * c) + d
a =  (b * c) - d
a = -(b * c) + d
a = -(b * c) - d
```

В каждом из этих выражений процессор применяет только одну операцию округления при вычислении конечного результата. Ниже показаны формат и семантика команд данной группы:

```
VFMADD132SS xmm1, xmm2, xmm3/m32 ; xmm1 = xmm1 * xmm3/m32 + xmm2
VFMADD213SS xmm1, xmm2, xmm3/m32 ; xmm1 = xmm2 * xmm1 + xmm3/m32
VFMADD231SS xmm1, xmm2, xmm3/m32 ; xmm1 = xmm2 * xmm3/m32 + xmm1

VFMSUB132SS xmm1, xmm2, xmm3/m32 ; xmm1 = xmm1 * xmm3/m32 - xmm2
VFMSUB213SS xmm1, xmm2, xmm3/m32 ; xmm1 = xmm2 * xmm1 - xmm3/m32
VFMSUB231SS xmm1, xmm2, xmm3/m32 ; xmm1 = xmm2 * xmm3/m32 - xmm1

VFNMADD132SS xmm1, xmm2, xmm3/m32; xmm1 = -(xmm1*xmm3/m32) + xmm2
VFNMADD213SS xmm1, xmm2, xmm3/m32; xmm1 = -(xmm2*xmm1) + xmm3/m32
VFNMADD231SS xmm1, xmm2, xmm3/m32; xmm1 = -(xmm2*xmm3/m32) + xmm1

VFNMSUB132SS xmm1, xmm2, xmm3/m32; xmm1 = -(xmm1*xmm3/m32) - xmm2
VFNMSUB213SS xmm1, xmm2, xmm3/m32; xmm1 = -(xmm2*xmm1) - xmm3/m32
VFNMSUB231SS xmm1, xmm2, xmm3/m32; xmm1 = -(xmm2*xmm3/m32) - xmm1
```

Для задач с длинными цепочками умножений и сложений команды FMA могут уменьшить эффект накопления ошибок и повысить скорость вычислений.

6.3 Пример программы для вычислений с плавающей точкой

Рассмотрим вычисление значения функции $\sin x$ через разложение в ряд Тейлора $\sin x = \sum_{n=0}^{\infty} (-1)^n x^{2n+1}/(2n+1)!$ с заданной точностью:

```
float eps = 1e-6;  // допустимая погрешность
float x = 0.5;
float result = 0.;

float term = x;
float sine = term;
float p    = -x * x;
int   n    = 0;
while (abs(term) > eps) {
    term *= p;
    n += 2;
    term /= n * (n + 1);
    sine += term;
}
result = sine;
```

Программа на языке ассемблера, реализующая данный алгоритм с использованием команд SSE, может выглядеть следующим образом:

```
     .data
eps  real4 1.e-6
x    real4 0.5
```

```
res  real4 ?

     .code
main proc
     movss    xmm2, dword ptr [x]  ; term = x
     movss    xmm0, xmm2           ; sine = term

     movss    xmm3, xmm2
     mulss    xmm3, xmm2           ; x * x
     movd     eax, xmm3
     xor      eax, 80000000h       ; -(x * x)

     movd     xmm3, eax            ; p = -x * x
     xor      ecx, ecx             ; n = 0
L1:
     movd     eax, xmm2
     and      eax, 7fffffffh       ; abs(term)
     movd     xmm4, eax
     ucomiss  xmm4, xmm1
     jbe      L2                   ; if (abs(term) <= eps)

     mulss    xmm2, xmm3           ; term *= p
     add      ecx, 2               ; n += 2
     lea      eax, dword ptr [rcx + 1]  ; eax = ecx + 1 !!!
     imul     eax, ecx
     cvtsi2ss xmm4, eax            ; xmm4 = n * (n + 1)
     divss    xmm2, xmm4           ; term /= n * (n + 1)
     addss    xmm0, xmm2           ; sine += term
     jmp      L1
L2:
     movss    res, xmm0
     ret
main endp
     end
```

Сумма ряда накапливается в младших четырех байтах регистра `xmm0`. Текущий член ряда хранится в регистре `xmm2`. Постоянный множитель `-x * x` хранится в регистре `xmm3`. Изменение знака числа с плавающей точкой в регистре `xmm3` выполнено следующим образом. Командой `movd` младшее двойное слово из регистра `xmm3` переносится в регистр общего назначения `eax` (без преобразования float в int). Затем командой `xor` производится инвертирование самого старшего бита (бит знака) в `eax`. После этого измененное значение возвращается в регистр `xmm3` командой `movd`. Аналогичный прием применяется для вычисления абсолютного значения текущего члена ряда в начале цикла. В данном случае бит знака очищается командой `and`. Целочисленная переменная `n` хранится в регистре `ecx`.

Для вычисления `n + 1` использован трюк с командой `lea`. Обычно команда `lea` используется для загрузки в регистр эффективного адреса области памяти, однако в данном примере она использована для вычисления арифметического

выражения. Одна инструкция `lea eax, dword ptr [rcx + 1]` фактически вычисляет выражение `eax = ecx + 1`, которое иначе нужно было вычислять с помощью двух инструкций: `mov eax, ecx` и `inc eax`.

Задания для самостоятельной и индивидуальной работы

5.1 Используя систему команд SSE реализовать на языке ассемблера MASM математическую функцию с помощью разложения в ряд Тейлора. Сравнить значения реализованной функции со значениями соответствующей математической функции C/C++, объявленной в заголовочном файле *math.h*. Ассемблерная функция должна вызываться из программы C++.

1. $\log(1+x) = \sum_{n=1}^{\infty}(-1)^{n+1}\frac{x^n}{n}, \quad -1 < x \leq 1$
2. $\log(1-x) = -\sum_{n=1}^{\infty}\frac{x^n}{n}, \quad -1 \leq x < 1$
3. $e^x = \sum_{n=0}^{\infty}\frac{x^n}{n!}$
4. $e^{-x} = \sum_{n=0}^{\infty}(-1)^n\frac{x^n}{n!}$
5. $\cos x = \sum_{n=0}^{\infty}\frac{(-1)^n}{(2n)!}x^{2n}$
6. $\sinh x = \sum_{n=0}^{\infty}\frac{x^{2n+1}}{(2n+1)!}$
7. $\cosh x = \sum_{n=0}^{\infty}\frac{x^{2n}}{(2n)!}$
8. $\arctan x = \sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}x^{2n+1}, \quad |x| \leq 1$
9. $(1+x)^{-\frac{1}{2}} = \sum_{n=0}^{\infty}\frac{(-1)^n(2n)!}{4^n(n!)^2}x^n$
10. $(1+x)^{\frac{1}{2}} = \sum_{n=0}^{\infty}\frac{(-1)^{n-1}(2n)!}{4^n(n!)^2(2n-1)}x^n$
11. $\frac{1}{1-x} = \sum_{n=0}^{\infty}x^n, \quad |x| < 1$
12. $\frac{1}{(1-x)^2} = \sum_{n=1}^{\infty}n\,x^{n-1}, \quad |x| < 1$

5.2 Используя систему команд AVX реализовать на языке ассемблера MASM математическую функцию из задания 5.1 с помощью разложения в ряд Тейлора. Сравнить значения реализованной функции со значениями соответствующей математической функции C/C++, объявленной в заголовочном файле *math.h*. Ассемблерная функция должна вызываться из программы C++.

5.3 Используя команды FMA, вычислить значение многочлена для заданного n в точке x по схеме Горнера.

1. $y = 10 + \sum_{k=1}^{n} kx^{2k}$

2. $y = 5 + \sum_{k=1}^{n} kx^{2k-1}$
3. $y = 10 + \sum_{k=1}^{n} 2kx^{2k}$
4. $y = 10 + \sum_{k=1}^{n} (2k-1)x^{2k-1}$
5. $y = 5 + \sum_{k=1}^{n} (-1)^k kx^{2k-1}$
6. $y = 10 + \sum_{k=1}^{n} (-1)^{k+1} 2kx^{2k}$
7. $y = 10 + \sum_{k=1}^{n} (-1)^{k+1} (2k-1)x^{2k-1}$
8. $y = 6 + \sum_{k=1}^{n} (-1)^k kx^{2k}$
9. $y = 10 + \sum_{k=1}^{n} (2k-1)x^{2k}$
10. $y = 5 + \sum_{k=1}^{n} 2kx^{2k-1}$
11. $y = \sum_{k=1}^{n} (-1)^k kx^k$
12. $y = 1 + \sum_{k=1}^{n} (-1)^k 2kx^k$

Лабораторная работа № 6–7

7 ВЕКТОРНЫЕ РАСШИРЕНИЯ СИСТЕМЫ КОМАНД x86-64

Цель работы: ознакомиться с основами векторной обработки данных на примере упакованных операций в системах команд SSE и AVX.

SIMD (Single Instruction, Multiple Data, один поток команд, несколько потоков данных) — это класс параллельных архитектур в таксономии Флинна, которые используют параллелизм на уровне данных: одна инструкция применяет одну и ту же операцию к нескольким значениям данных одновременно. Задачи, легко поддающиеся такому распараллеливанию, часто называют *естественно параллельными* или *чрезвычайно параллельными* (*embarrassingly parallel*). Примеры таких задач: обработка 3D-графики, научные вычисления, цифровая обработка сигналов.

В архитектуре x86-64 имеется ряд расширений системы команд, предоставляющих поддержку вычислений класса SIMD, таких как различные версии SSE (Streaming SIMD Extensions) и AVX (Advanced Vector Extensions).

7.1 Регистры SSE/AVX

Архитектуры SSE/AVX используют специальные наборы регистров:

- Регистры XMM (SSE/SSE2/SSE3/SSE4): 16 регистров XMM0–XMM15, каждый имеет разрядность 128 бит. В один 128-битный регистр может быть «упаковано»:
  - 4 x 32-разрядных числа с плавающей точкой одинарной точности;
  - 2 x 64-разрядные числа с плавающей точкой двойной точности;
  - 16 x 8-битных целых чисел (byte);
  - 8 x 16-битных целых чисел (word);
  - 4 x 32-битных целых числа (dword);
  - 2 x 64-битных целых числа (qword).
- Регистры YMM (AVX/AVX2): 16 регистров YMM0–YMM15 шириной 256 бит. Регистры YMM являются расширениями регистров XMM. Например, младшие 128 бит YMM0 — это регистр XMM0. Один 256-битный регистр может содержать:
  - 8 x 32-разрядных числа с плавающей точкой одинарной точности;
  - 4 x 64-разрядные числа с плавающей точкой двойной точности;
  - 32 x 8-битных целых чисел (byte);
  - 16 x 16-битных целых чисел (word);
  - 8 x 32-битных целых числа (dword);
  - 4 x 64-битных целых числа (qword).

- Регистры ZMM (AVX-512): до 32 регистров разрядностью 512 бит. Первые 16 из них являются расширениями регистров YMM. Система команд AVX-512 как правило встречается в более дорогих процессорах семейства x86-64, предназначенных для серверов и рабочих станций. AVX-512 далее подробно рассматриваться не будет.

MASM использует три типа данных, соответствующих указанным регистрам:

- XMMWORD — 128-битный операнд для инструкций SSE;
- YMMWORD — 256-битный операнд для инструкций AVX/AVX2;
- ZMMWORD — 512-битный операнд для инструкций AVX-512.

7.2 Выравнивание операндов SSE/AVX в памяти

Для достижения максимальной производительности для инструкций SIMD требуется, чтобы данные в памяти были *выровнены*:

- SSE (128 бит) — данные должны располагаться по адресам кратным 16;
- AVX (256 бит) — адреса данных должны быть кратны 32.

Доступ к невыровненным данным возможен с помощью определенных инструкций, но это приводит к значительному снижению производительности. В MASM можно обеспечить выравнивание данных или инструкций с помощью директивы ALIGN:

```
          .data
scalar    byte  ?

          ALIGN 16   ; Выровнять для SSE
sse_arr1  real4 1.0, 2.0, 3.0, 4.0
sse_arr2  dword   1,   2,   3,   4

; Error A2189: invalid combination with segment alignment : 32
          ALIGN 32   ; Выровнять для AVX
avx_array real4 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
```

Начало сегмента `.data` гарантированно выровнено по границе в 16 байт. Если не использовать директиву `ALIGN 16`, массив `sse_arr1` будет начинаться в памяти с нечетного адреса (один байт перед ним занимает переменная `scalar`), т. е. не будет выровнен. Директива `ALIGN 16`, заставляет MASM добавить между `scalar` и `sse_arr1` дополнительные 15 байт.

Следующая директива (`ALIGN 32`) в данном примере, к сожалению, работать не будет. Дело в том, что числовой параметр после ALIGN должен быть степенью двойки и не превышать величину выравнивания для текущего сегмента (для сегмента `.data` — это 16). Чтобы исправить ошибку, необходимо явно определить сегмент (данных), который выровнен как минимум по 32-байтной границе.

Общий синтаксис объявления сегмента в MASM имеет следующий вид:

```
имя_сегмента SEGMENT [READONLY] [alignment] [class]
             инструкции
имя_сегмента ENDS
```

Здесь `имя_сегмента` — это идентификатор. Он не обязательно должен быть уникальным, можно объявить несколько сегментов с одинаковыми именами. MASM их объединит при генерации кода в объектный файл. Следует избегать имен сегментов _TEXT, _DATA, _BSS и _CONST, так как MASM использует эти имена при определении сегментов с помощью директив .CODE, .DATA, .DATA? и .CONST соответственно. В квадратные скобки взяты необязательные атрибуты сегмента. Опция `READONLY` — это подсказка для MASM, что сегмент будет содержать данные только для чтения (константы). Параметр `alignment` позволяет указать способ выравнивания сегмента, как показано в таблице 7.1. Если он не указан, используется выравнивание по параграфам (16 байт). Такое же выравнивание используется для директив упрощенного объявления сегментов (.CODE, .DATA, .DATA?, .CONST). Необязательный атрибут `class` представляет собой строку (заключенную в одинарные кавычки) и обычно имеет значение `'CODE'`, `'DATA'` или `'CONST'`. MASM и компоновщик объединяют сегменты с одинаковыми именами классов, даже если имена сегментов различаются.

Таблица 7.1. — Параметры выравнивания при объявлении сегмента в MASM

Параметр выравнивания	**Начальный адрес**
BYTE	Адрес следующего доступного байта.
WORD	Адрес следующего доступного слова (2 байта).
DWORD	Адрес следующего доступного двойного слова (4 байта).
PARA	Адрес следующего доступного параграфа (16 байт). Используется по умолчанию.
PAGE	Адрес следующей доступной страницы (256 байт).
ALIGN(*n*)	Адрес следующего доступного *n*-го байта. Константа *n* должна быть степенью двойки.

Таким образом, предыдущий пример можно переписать в виде:

```
DataSeg   SEGMENT ALIGN(32)
scalar    byte  ?
          ALIGN 16   ; Выровнять для SSE
sse_arr1  real4 1.0, 2.0, 3.0, 4.0
sse_arr2  dword   1,   2,   3,   4

          ALIGN 32   ; Выровнять для AVX
avx_array real4 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
DataSeg   ENDS
```

Директива `ALIGN 32` заставит MASM выделить дополнительные 16 байт из нулей между `sse_arr2` и `avx_array`.

7.3 Система команд SSE

В этом разделе делается обзор основных групп команд SSE. В названиях команд используется устоявшаяся система сокращений, которая позволяет сразу понять, с какими типами данных работает инструкция и в каком режиме она применяется. Так для обозначения целочисленных типов данных, упакованных в 128-битном регистре XMM или области памяти в названиях команд используются следующие суффиксы:

- B (**B**yte) — 8-битный элемент;
- W (**W**ord) — 16-битный элемент;
- D (**D**oubleword) — 32-битный элемент;
- Q (**Q**uadword) — 64-битный элемент;
- DQ (**D**ouble **Q**uadword) — 128-битный блок (обычно соответствует целому регистру XMM).

Большинство команд с плавающей точкой работают в двух режимах: *упакованном* — когда операция применяется ко всем отдельным элементам данных и *скалярном* — когда операция применяется только к младшим элементам данных. Соответственно в таких командах встречаются следующие суффиксы:

- PS (**P**acked **S**ingle-precision) — упакованные числа с плавающей точкой одинарной точности (32-битные float);
- PD (**P**acked **D**ouble-precision) — упакованные числа с плавающей точкой двойной точности (64-битные double);
- SS (**S**calar **S**ingle-precision) — скалярная операция над одним числом с плавающей точкой одинарной точности;
- SD (**S**calar **D**ouble-precision) — скалярная операция над одним числом с плавающей точкой двойной точности.

Скалярные команды SSE рассматривались в главе 6. В этом разделе основной акцент сделан на командах, параллельно применяющих одну операцию к нескольким упакованным данным (SIMD-обработка).

Команды SSE как правило имеют следующий формат:

```
<мнемоника> xmm, xmm/m128
```

где целевой операнд `xmm` обозначает один из 128-битных регистров, а исходный операнд `xmm/m128` обозначает либо один из 128-битных регистров, либо 128-битную область памяти, определенную любым из допустимых режимов адресации. Некоторые команды используют третий 8-битный непосредственный (immediate) операнд, обозначаемый `imm8`.

7.3.1 Команды перемещения данных

Команды MOVD (**MOV**e **D**word) / MOVQ (**MOV**e **Q**word) перемещают двойное/четверное слово между регистром общего назначения или областью памяти и младшей частью регистра XMM. Если в качестве операнда назначения используется регистр XMM, его старшая 96/64-битная часть обнуляется. Скаляр с плавающей точкой одинарной/двойной точности перемещается аналогичным образом с помощью инструкций MOVSS (**MOV**e **S**calar **S**ingle) / MOVSD (**MOV**e **S**calar **D**ouble). При загрузке в регистр XMM его старшая часть также очищается. Эти команды рассматривались в главе 6.

В таблице 7.2 перечислены инструкции, которые могут быть использованы для перемещения упакованных данных из 128-битной области памяти в регистр XMM, перемещения данных из регистра XMM в 128-битную область памяти или перемещения данных между двумя регистрами XMM.

Таблица 7.2. — Команды перемещения данных SSE

Мнемоника	**Описание**
MOVDQA	Копирует выровненное двойное четверное слово из памяти в регистр XMM или наоборот. Эта инструкция также может использоваться для переноса данных между регистрами XMM.
MOVDQU	Копирует невыравненное двойное четверное слово из памяти в регистр XMM или наоборот.
MOVAP[S|D]	Копирует упакованные значения с плавающей точкой между выровненной областью памяти и регистром XMM или между двумя регистрами XMM.
MOVUP[S|D]	Копирует упакованные значения с плавающей точкой между невыровненной областью памяти и регистром XMM или между двумя регистрами XMM.

Команды MOVDQA (**MOV**e **D**ouble **Q**uadword **A**ligned), MOVAPS (**MOV**e **A**ligned **P**acked **S**ingle) и MOVAPD (**MOV**e **A**ligned **P**acked **D**ouble) перемещают 128-битные упакованные данные — целые числа (суффикс DQ), числа с плавающей точкой одинарной (суффикс PS) и двойной точности (суффикс PD) соответственно — между регистром XMM и областью памяти, выровненной по 16 байтам, или между двумя регистрами XMM. Команда для перемещения выровненных данных не может использоваться для переменной в памяти, которая не выровнена по 16 байтам, так как это приводит к ошибке во время выполнения программы. Если область памяти не выровнена или имеет неизвестное выравнивание, необходимо использовать одну из более медленных инструкций MOVDQU (**MOV**e **D**ouble **Q**uadword **U**naligned), MOVUPS (**MOV**e **U**naligned **P**acked **S**ingle) или MOVUPD (**MOV**e **U**naligned **P**acked **D**ouble). Невыровненные операнды вида `m128` должны быть сначала загружены в 128-битный регистр, например:

```
movups xmm1, [rsp]
addps  xmm0, xmm1
```

Здесь предполагается, что `rsp` указывает на 128-битный операнд в памяти. Если известно, что операнд `[rsp]` выровнен, то можно ограничиться одной командой:

```
addps xmm0, [rsp]
```

В таблице 7.3 приведены команды перемещения данных, которые затрагивают только верхнюю или нижнюю половину регистра XMM.

Таблица 7.3. — Команды перемещения данных SSE

Мнемоника	**Описание**
MOVHLPS	Копирует верхнее четверное слово упакованных значений с плавающей точкой одинарной точности из операнда-источника в нижнее четверное слово операнда-приемника. Оба операнда — регистры XMM.
MOVLHPS	Копирует нижнее четверное слово упакованных значений с плавающей точкой одинарной точности из операнда-источника в верхнее четверное слово операнда-приемника. Оба операнда — регистры XMM.
MOVHP[S|D]	Копирует верхнее четверное слово упакованных значений с плавающей точкой из памяти в регистр XMM или наоборот. Если операнд приемник — регистр XMM, его нижнее четверное слово не изменяется.
MOVLP[S|D]	Копирует нижнее четверное слово упакованных значений с плавающей точкой из памяти в регистр XMM или наоборот. Если операнд приемник — регистр XMM, его верхнее четверное слово не изменяется.

Команды MOVHLPS (**MOV**e **H**igh to **L**ow **P**acked **S**ingle) / MOVLHPS (**MOV**e **L**ow to **H**igh **P**acked **S**ingle) перемещают два верхних/нижних числа с плавающей точкой одинарной точности из исходного операнда (регистр XMM) на позиции двух нижних/верхних чисел с плавающей точкой одинарной точности конечного операнда (регистр XMM) как показано в примере ниже:

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
movhlps xmm0, xmm1
; xmm0: [ a3 | a2 | b3 | b2 ]

; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
movlhps xmm0, xmm1
; xmm0: [ b1 | b0 | a1 | a0 ]
```

Команды MOVHPD (**MOV**e **H**igh **P**acked **D**ouble) / MOVLPD (**MOV**e **L**ow **P**acked **D**ouble) перемещают число с плавающей точкой двойной точности между верхней/нижней половиной регистра XMM и 64-битной областью памяти. Если данные загружаются из памяти в одну часть регистра XMM, другая часть регистра не изменяется:

```
; xmm0: [ q1 | q0 ]
```

```
; mem:  [ x ]
movhpd xmm0, mem
; xmm0: [ x | q0 ]
```

Команды MOVHPS (**MOV**e **H**igh **P**acked **S**ingle) и MOVLPS (**MOV**e **L**ow **P**acked **S**ingle) используются аналогичным образом для перемещения двух чисел с плавающей точкой одинарной точности между регистром XMM и 64-битной областью памяти:

```
; xmm0: [ d3 | d2 | d1 | d0 ]
; mem:  ?
movlps mem, xmm0
; mem: [ d1 | d0 ]
```

В таблице 7.4 приведены команды вида MOV[D|SH|SL]DUP (**MOV**e **D**ouble / **S**ingle **H**igh / **S**ingle **L**ow and **DUP**licate), которые комбинируют перемещение данных с дублированием.

Таблица 7.4. — Команды перемещения с дублированием

Мнемоника	**Описание**
MOVDDUP	Копирует нижнее значение с плавающей точкой двойной точности из операнда-источника в нижнее и верхнее четверные слова операнда-приемника.
MOVSHDUP	Копирует верхнее значение с плавающей точкой одинарной точности из каждого четверного слова операнда-источника в ту же позицию операнда-приемника. Верхнее значение с плавающей точкой одинарной точности из каждого четверного слова операнда-приемника дублируется в нижние 32 бита.
MOVSLDUP	Копирует нижнее значение с плавающей точкой одинарной точности из каждого четверного слова операнда-источника в ту же позицию операнда-приемника. Нижнее значение с плавающей точкой одинарной точности из каждого четверного слова операнда-приемника дублируется в верхние 32 бита.

Например, команда MOVDDUP (**MOV**e **D**ouble and **DUP**licate) дублирует четверное слово исходного операнда в целевом операнде:

```
; xmm1/mem: [ ? | q0 ]
movddup xmm0, xmm1/mem
; xmm0:     [ q0 | q0 ]
```

Команды MOVSHDUP (**MOV**e **S**ingle **H**igh and **DUP**licate) / MOVSLDUP (**MOV**e **S**ingle **L**ow and **DUP**licate) дублируют двойные слова с нечетными индексами (1 и 3) / с четными индексами (0 и 2) в операнде назначения, например:

```
; xmm0: [ ?? | ?? | ?? | ?? ]
; xmm1: [ f3 | f2 | f1 | f0 ]
movshdup xmm0, xmm1
```

```
; xmm0: [ f3 | f3 | f1 | f1 ]

; xmm0: [ ?? | ?? | ?? | ?? ]
; xmm1: [ f3 | f2 | f1 | f0 ]
movsldup xmm0, xmm1
; xmm0: [ f2 | f2 | f0 | f0 ]
```

В таблице 7.5 приведены команды, которые перемещают один элемент данных между регистром общего назначения или областью памяти (`reg/mem`) и 128-битным регистром (`xmm`):

```
<*extr*> reg/mem, xmm, imm8
<*insr*>  xmm, reg/mem, imm8
insertps  xmm1, xmm2/mem, imm8
```

Непосредственное значение `imm8` задает индекс элемента в `xmm`.

Таблица 7.5. — Команды вставки и извлечения данных

Мнемоника	**Описание**
PEXTR[B|W|D|Q]	Копирует целочисленное значение из регистра XMM в область памяти или регистр общего назначения. Положение целого числа в исходном операнде задаётся с помощью непосредственного операнда.
EXTRACTPS	Извлекает элемент с плавающей точкой одинарной точности из регистра XMM и копирует его в область памяти или регистр общего назначения. Местоположение извлекаемого элемента указывается непосредственным операндом.
PINSR[B|W|D|Q]	Копирует целое число из области памяти или регистра общего назначения в регистр XMM. Положение целого числа в целевом операнде задаётся с помощью непосредственного операнда.
INSERTPS	Копирует значение с плавающей точкой одинарной точности из области памяти или регистра XMM и вставляет его в регистр XMM. Позиции элемента в исходном и целевом операндах задаются непосредственным операндом.
PMOVMSKB	Создаёт маску, составленную из старших битов каждого байта исходного операнда, и сохраняет ее в младшее слово целевого операнда.
MOVMSKP[S|D]	Извлекает биты знака из каждого упакованного элемента данных с плавающей точкой в регистре XMM, формирует из них маску (2 или 4 бита) и сохраняет их в младших битах регистра общего назначения. Старшие биты за пределами маски заполняются нулями.

Команды, в названии которых присутствует корень EXTR, извлекают упакованный элемент из регистра XMM в РОН или память, например

```
; xmm0: [ d3 | d2 | d1 | d0 ]
; r32:  [ ?? ]
PEXTRD r32, xmm0, 2
; r32:  [ d2 ]
```

```
; xmm0: [ w7 | w6 | w5 | w4 | w3 | w2 | w1 | w0 ]
; r32:                                [ a1 | a0 ]
PEXTRW r32, xmm0, 3
; r32:                                [  0 | w3 ]
```

Команды, в названии которых присутствует INSR, вставляют значение из РОН или памяти в указанный элемент регистра XMM:

```
; xmm0: [ a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 ]
; r32:                                [ ?? |  w ]
PINSRW xmm0, r32, 1
; xmm0: [ a7 | a6 | a5 | a4 | a3 | a2 | w | a0 ]
```

Команда INSERTPS имеет немного другой формат:

```
insertps xmm1, xmm2/mem, imm8
```

Непосредственное значение содержит три поля:

- Биты imm8[7:6] — задают индекс (от 0 до 3) элемента в исходном операнде xmm2, который будет скопирован. Если исходный операнд — область памяти, то это поле игнорируется.
- Биты imm8[5:4] — выбирают индекс (от 0 до 3) элемента в целевом операнде xmm1, который будет переписан.
- Биты imm8[3:0] — маска, каждый бит которой одному элементу в регистре назначения. Если бит установлен, соответствующий элемент обнуляется после вставки.

Следующий пример иллюстрирует работу команды INSERTPS:

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
insertps xmm0, xmm1, 11010100b
; xmm0: [ a3 | 0 | b3 | a0 ]
```

Здесь копируется элемент с индексом 3 из `xmm1` и вставляется в позицию с индексом 1 в `xmm0`. Затем элемент номер 2 в `xmm0` обнуляется согласно маске `0100b`.

Команды маскирования PMOVMSKB (**P**acked **MOV**e **M**a**SK** **B**yte) и MOVMSKP[S|D] (**MOV**e **M**a**SK** **P**acked **S**ingle/**D**ouble) имеют следующий формат:

```
[p]movmsk[b|ps|pd] r32/64, xmm
```

Они формируют соответственно 16-битную, 4-битную и 2-битную маску из старших битов упакованных байтов, упакованных чисел с плавающей точкой одинарной точности и упакованных чисел с плавающей точкой двойной точ-

ности, находящихся в 128-битном исходном регистре, и сохраняют эту маску в 32 или 64-битном целевом регистре. Эти команды обычно используются для сжатия 128-битной маски, сгенерированной командой параллельного сравнения, в регистр общего назначения.

7.3.2 Арифметические команды

Практически все команды для выполнения арифметических операций над упакованными данными имеют следующий формат:

```
<команда> xmm1, xmm2/m128
```

где второй операнд — источник, первый операнд — приемник результата.

В таблице 7.6 перечислены команды выполняющие арифметические операции над целочисленными упакованными данными. Как уже упоминалось, тип упакованных данных закодирован в суффиксе: B — байт; W — слово; D — двойное слово; Q — четверное слово.

Таблица 7.6. — Команды целочисленной арифметики

Мнемоника	**Описание**
PADD[B|W|D|Q]	Сложение упакованных данных (знаковых и беззнаковых).
PSUB[B|W|D|Q]	Вычитание упакованных данных (знаковых и беззнаковых).
PADDS[B|W]	Сложение упакованных данных (знаковых) с насыщением.
PADDUS[B|W]	Сложение упакованных данных (беззнаковых) с насыщением.
PSUBS[B|W]	Вычитание упакованных данных (знаковых) с насыщением.
PSUBUS[B|W]	Вычитание упакованных данных (беззнаковых) с насыщением.
PMULL[W|D|Q]	Выполняет умножение упакованных целых со знаком. Младшая половина каждого произведения сохраняется в целевом операнде.
PMINS[B|W|D]	Минимумы упакованных данных (знаковых).
PMINU[B|W|D]	Минимумы упакованных данных (беззнаковых).
PMAXS[B|W|D]	Максимумы упакованных данных (знаковых).
PMAXU[B|W|D]	Максимумы упакованных данных (беззнаковых).
PABS[B|W|D]	Абсолютное значение упакованных данных.

Все перечисленные в таблице команды применяют соответствующую операцию параллельно во всех дорожках (или слотах) своих 128-разрядных операндов, как показано ниже на примере операции вычитания (SUB) упакованных двойных слов:

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
psubd xmm0, xmm1
; xmm0: [ a3 - b3 | a2 - b2 | a1 - b1 | a0 - b0 ]
```

Операции сложения и вычитания *с насыщением* отличаются от соответствующих операций с переполнением (*арифметика по модулю*) тем, что, если

результат вычисления выходит за допустимые пределы диапазона соответствующего типа данных, он ограничивается максимальным или минимальным значением этого диапазона.

Команды выполняющие арифметические операции над упакованными значениями с плавающей точкой перечислены в таблице 7.7.

Таблица 7.7. — Команды арифметики с плавающей точкой SSE

Мнемоника	**Операция**
ADDP[S|D]	Сложение
SUBP[S|D]	Вычитание
MULP[S|D]	Умножение
DIVP[S|D]	Деление
MINP[S|D]	Минимум
MAXP[S|D]	Максимум
SQRTP[S|D]	Квадратный корень
RCPPS	Приближенное обратное значение
RSQRTPS	Приближенное обратное значение квадратного корня

Все команды данной группы выполняют параллельную SIMD-обработку своих операндов, например параллельное умножение двойных слов:

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
mulpd xmm0, xmm1
; xmm0: [ a3 * b3 | a2 * b2 | a1 * b1 | a0 * b0 ]
```

В таблице 7.8 перечислены команды, выполняющие арифметических операции с некоторыми особенностями.

Таблица 7.8. — Специальные арифметические команды SSE

Мнемоника	**Расшифровка**
PAVG[B|W]	Упакованное среднее с округлением (беззнаковое)
PMULHW	Выполняет умножение упакованных слов со знаком. Старшее слово каждого произведения (двойного слова) сохраняется в целевом операнде.
PMULHUW	Выполняет умножение упакованных слов без знака. Старшее слово каждого произведения (двойного слова) сохраняется в целевом операнде.
PMULDQ	Умножает упакованные знаковые двойные слова с четными индексами из исходного и конечного операндов. Произведения (четверные слова) сохраняются в операнде назначения.
PMULUDQ	Умножает упакованные беззнаковые двойные слова с четными индексами из исходного и конечного операндов. Произведения (четверные слова) сохраняются в операнде назначения.

PSADBW	Вычисляет абсолютные разности упакованных однобайтных целых чисел без знака из первого и второго операндов. Затем 8 младших разностей и 8 старших разностей суммируются отдельно, формируя два результирующих беззнаковых слова.
PMADDWD	Выполняет умножение упакованных целых чисел со знаком, затем складывает соседние пары произведений.
PMADDUBSW	Выполняет умножение упакованных знаковых байтов исходного операнда исходного операнда и беззнаковых байтов конечного операнда. Смежные результирующие знаковые слова складываются с насыщением и суммы сохраняются в операнде назначения.
ADDSUBP[S|D]	Складывает элементы с плавающей точкой с нечётными номерами и вычитает элементы с плавающей точкой с чётными номерами.
HADDP[S|D]	Складывает соседние элементы с плавающей точкой, содержащиеся в исходном и целевом операндах.
HSUBP[S|D]	Вычитает соседние элементы с плавающей точкой, содержащиеся в исходном и целевом операндах.
PHADD[W|D]	Складывает соседние целочисленные элементы в исходном и целевом операндах.
PHADDSW	Выполняет сложение с насыщением соседних знаковых слов в исходном и целевом операндах. Результаты упаковывает в операнде назначения.
PHSUB[W|D]	Вычитает соседние целочисленные элементы в исходном и целевом операндах.
PHSUBSW	Выполняет вычитание с насыщением соседних знаковых слов в исходном и целевом операндах. Результаты упаковывает в операнде назначения.
PSIGN[B|W|D]	Меняет знак, обнуляет или сохраняет каждый целочисленный элемент данных в целевом операнде на основе знака соответствующего элемента данных в исходном операнде.

Команды PAVG[B|W] (**P**acked **AV**era**G**e **B**yte/**W**ord) для каждой пары упакованных байтов или слов из двух операндов упакованных данных байтов или слов вычисляют среднее арифметическое с округлением вверх:

```
dst[i] = (src1[i] + src2[i] + 1) >> 1
```

где `src1` и `src2` — первый и второй исходные операнды; `dst` — целевой операнд; `i` — индекс упакованного элемента. Команды PMUL[L|H]* (**P**acked **MUL**tiply **L**ow/**H**igh) выполняют параллельное умножение упакованных целых значений, сохраняя только нижнюю или верхнюю половину произведения.

Команда PSADBW (**P**acked **S**um of **A**bsolute **D**ifferences **B**yte to **W**ord) вычисляет суммы абсолютных разностей $s_0 \coloneqq \sum_{i=0}^{7}|a_i - b_i|$ и $s_1 \coloneqq \sum_{i=8}^{15}|a_i - b_i|$ упакованных байтов и сохраняет 32-битные суммы s_0 и s_1, соответственно, в 0-е и 4-е слово операнда назначения. Все остальные слова обнуляются.

Команда PMADDWD (**P**acked **M**ultiply and **ADD** **W**ord to **D**word) выполняет умножение упакованных слов со знаком и сохраняет 32-битные суммы попарных результатов в виде упакованных двойных слов в целевом операнде:

```
; xmm0: [  a7  |  a6  |  a5  |  a4  |  a3  |  a2  |  a1  |  a0  ]
; xmm1: [  b7  |  b6  |  b5  |  b4  |  b3  |  b2  |  b1  |  b0  ]
pmaddwd xmm0, xmm1
; xmm0: [ a7*b7+a6*b6 | a5*b5+a4*b4 | a3*b3+a2*b2 | a1*b1+a0*b0 ]
```

Эта команда может использоваться для вычисления целочисленного скалярного произведения. Похожим образом работает команда PMADDUBSW, однако вместо упакованных слов перемножает упакованные байты. При этом байты из исходного операнда интерпретируются как знаковые, а байты из конечного операнда, как беззнаковые. Кроме того, использует сложение с насыщением.

Команды ADDSUBP[S|D] (**ADD**/**SUB**tract **P**acked **S**ingle/**D**ouble) чередуют вычитание и сложение упакованных элементов данных, как показано в следующем примере:

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
addsubps xmm0, xmm1
; xmm0: [ a3+b3 | a2-b2 | a1+b1 | a0-b0 ]
```

Команды HADDP[S|D] (**H**orizontal **ADD** **P**acked **S**ingle/**D**ouble) и HSUBP[S|D] (**H**orizontal **SUB**tract **P**acked **S**ingle/**D**ouble) выполняют горизонтальное сложение и вычитание соответственно, например:

```
; xmm0: [   a1   |   a0   ]
; xmm1: [   b1   |   b0   ]
haddpd xmm0, xmm1
; xmm0: [  b0+b1 |  a0+a1 ]

; xmm0: [   a3  |   a2  |   a1  |   a0  ]
; xmm1: [   b3  |   b2  |   b1  |   b0  ]
hsubps xmm0, xmm1
; xmm0: [ b2-b3 | b0-b1 | a2-a3 | a0-a1 ]
```

Команды PHADD[W|D] и PHSUB[W|D] работают аналогично командам HADDP[S|D] и HSUBP[S|D] соответственно, только для упакованных целочисленных элементов.

Команды PHADDSW и PHSUBSW работают как PHADDW и PHSUBW соответственно, но используют арифметику с насыщением (отсюда суффикс S — **S**aturation).

Команды PSIGN[B|W|D] для каждого элемента первого операнда: меняют знак, если соответствующий элемент второго операнда меньше нуля; заменяет нулем, если соответствующий элемент второго операнда равен нулю; в противном случае оставляет элемент без изменения. Например:

```
; xmm0: [  10 |  -3 |  52 |   5 ]
; xmm1: [   0 |  -1 |  -1 |   1 ]
psignd xmm0, xmm1
```

```
; xmm0: [   0 |   3 | -52 |   5 ]
```

7.3.3 Логические команды

В таблице 7.9 перечислены команды выполняющие побитовые логические операции над упакованными 128-битными операндами. Имеют такой же формат, как и арифметические команды:

```
<команда> xmm1, xmm2/m128
```

Таблица 7.9. — Логические команды SSE

Мнемоника	**Описание**
PAND ANDP[S|D]	Побитовое логическое И: `dst = dst & src`
PANDN ANDNP[S|D]	Побитовое логическое И-НЕ: `dst = ~dst & src`
POR ORP[S|D]	Побитовое логическое ИЛИ: `dst = dst | src`
PXOR XORP[S|D]	Побитовое логическое исключающее ИЛИ: `dst = dst ^ src`

PTEST	Логическое сравнение упакованных данных:
	`ZF = (dst & src) == 0 ? 1 : 0`
	`CF = (~dst & src) == 0 ? 1 : 0`

Например, команда ANDPS (**AND P**acked **D**ouble) работает следующим образом:

```
; xmm0: [ 0110001100010011 | ... | ... | 0010000100000111 ]
; xmm1: [ 1100000100010001 | ... | ... | 0010010001001111 ]
andps xmm0, xmm1
; xmm0: [ 0100000100010001 | ... | ... | 0010000000000111 ]
```

Команда PTEST (**P**acked **TEST**) похожа на стандартную целочисленную инструкцию TEST. Она выполняет побитовую операцию AND между двумя операндами и устанавливает флаг нуля (ZF), если результат равен 0. Также она устанавливает флаг переноса (CF), если побитовая операция AND второго операнда с инвертированными битами первого операнда даёт результат 0.

Логические инструкции часто применяются к битовой маске, которая формируется командой сравнения, чтобы реализовать условные операции без использования явных переходов.

7.3.4 Команды сравнения

Команды, приведённые в таблице 7.10, сравнивают упакованные элементы данных в операнде назначения с упакованными элементами данных в исходном операнде, используя отношение, определяемое кодом сравнения «cc», и устанавливают все биты каждого отдельного элемента данных в операнде

назначения в единицы, если результат сравнения истинный, или в нули в противном случае. Эти команды не изменяют флаги состояния. Вместо этого полученная битовая маска обычно используется последующими логическими командами для выбора тех элементов данных, которые удовлетворяют определённому условию.

Таблица 7.10. — Команды сравнения SSE

Мнемоника	**Описание**
PCMPcc[B|W|D]	Сравнение упакованных целых значений.
CMPccP[S|D]	Сравнение упакованных значений с плавающей точкой.

Команды PCMPcc (**P**acked **C**o**MP**are with **c**omparison **c**ode) поддерживают сравнения упакованных целых значений с использованием кодов сравнения EQ (**EQ**ual, равно), применимых как к операндам со знаком, так и без знака, а также GT (**G**reater **T**han, больше), применимого к беззнаковым операндам. Ниже показан пример работы команды PCMPEQD, которая параллельно сравнивает на равенство упакованные двойные слова:

```
; xmm0: [          51 |           0 |         102 |         328 ]
; xmm1: [          51 |          30 |          10 |         328 ]
pcmpeqd xmm0, xmm1
; xmm0: [ 0ffffffffh | 000000000h | 000000000h | 0ffffffffh ]
```

Команды CMPccP[S|D] в действительности являются двухоперандными псевдокомандами, реализуемыми ассемблером, за которыми скрыта настоящая трехоперандная команда сравнения CMPP[S|D]:

```
CMPP[S|D] xmm, xmm/ml28, imm8
```

Она сравнивает значения с плавающей точкой в двух первых операндах. Два младших бита третьего непосредственного операнда `imm8` задают способ сравнения, как показано в таблице 7.11. Два значения с плавающей точкой *несравнимы* (unordered), если хотя бы одно из них имеет значение NaN (Not-a-Number).

Таблица 7.11. — Псевдокоманды сравнения упакованных чисел с плавающей точкой и их реализация

Псевдокоманда	**Реализация**	**Операция**
`CMPEQP[S|D] l, r`	`CMPP[S|D] l, r, 0`	`l == r`
`CMPLTP[S|D] l, r`	`CMPP[S|D] l, r, 1`	`l < r`
`CMPLEP[S|D] l, r`	`CMPP[S|D] l, r, 2`	`l <= r`
`CMPUNORDP[S|D] l, r`	`CMPP[S|D] l, r, 3`	`несравнимо`
`CMPNEQP[S|D] l, r`	`CMPP[S|D] l, r, 4`	`l != r`

`CMPNLTP[S|D] l, r`	`CMPP[S|D] l, r, 5`	`l >= r`
`CMPNLEP[S|D] l, r`	`CMPP[S|D] l, r, 6`	`l > r`
`CMPORDP[S|D] l, r`	`CMPP[S|D] l, r, 7`	`сравнимо`

Подобные команды и псевдокоманды существуют также для скалярных значений с плавающей точкой и рассматривались в главе 6.

7.3.5 Команды преобразования данных

В таблице 7.12 приведены команды преобразования упакованных данных. Команды, содержащие в названии PACK, преобразуют упакованные элементы данных в упакованные элементы данных с меньшей разрядностью, при этом насыщая отдельные элементы данных до соответствующего предельного значения диапазона.

Таблица 7.12. — Команды преобразования упакованных данных SSE

Мнемоника	**Описание**
PACKSS[WB|DW]	Преобразует упакованные слова/двойные слова со знаком в исходном и целевом операндах в упакованные байты/слова со знаком с использованием насыщения со знаком.
PACKUS[WB|DW]	Преобразует упакованные слова/двойные слова со знаком в исходном и целевом операндах в упакованные беззнаковые байты/слова с использованием беззнакового насыщения.
CVTDQ2PS	Преобразует четыре упакованных 32-битных знаковых целых числа в четыре упакованных значения с плавающей точкой одинарной точности.
CVTDQ2PD	Преобразует два нижних упакованных 32-битных знаковых целых числа в два упакованных значения с плавающей точкой двойной точности.
CVTPS2DQ	Преобразует четыре упакованных значения с плавающей точкой одинарной точности в четыре упакованных 32-битных знаковых целых числа.
CVTPD2DQ	Преобразует два упакованных значения с плавающей точкой двойной точности в два нижних упакованных 32-битных знаковых целых числа.
CVTPS2PD	Преобразует два упакованных значения с плавающей точкой одинарной точности в два упакованных значения с плавающей точкой двойной точности.
CVTPD2PS	Преобразует два упакованных значения с плавающей точкой двойной точности в два упакованных значения с плавающей точкой одинарной точности.
CVTTPS2DQ	Преобразует четыре упакованных значения с плавающей точкой одинарной точности в четыре упакованных 32-битных знаковых целых числа, используя усечение.
CVTTPD2DQ	Преобразует два упакованных значения с плавающей точкой двойной точности в два нижних упакованных 32-битных знаковых целых числа, используя усечение.

Другие инструкции преобразования используют формат CVT<s2d> для кода преобразования <s2d> (**s**ource type to **d**estination type). Например код DQ2PS означает преобразование упакованных знаковых двойных слов в упакованные числа с плавающей точкой одинарной точности.

При преобразовании чисел с плавающей точкой в знаковые целые режим округления задаётся битами 13 и 14 регистра управления и состояния MXCSR следующим образом:

- 00 — округление к ближайшему;
- 01 — округление к $-\infty$;
- 10 — округление к $+\infty$;
- 11 — округление к нулю (усечение).

Команды вида CVTT<s2d> выполняют округление с усечением. и одного из следующих кодов преобразования. Команды преобразования с кодами PD2DQ и PD2PS очищают два старших 32-битных элемента в целевом регистре.

7.3.6 Команды сдвига

В таблице 7.13 приведены команды, выполняющие логические и арифметические сдвиги упакованных слов, двойных слов и четверных слов в регистре XMM. Команды PSLLDQ (**P**acked **S**hift **L**eft **L**ogical **D**ouble **Q**word) и PSRLDQ (**P**acked **S**hift **R**ight **L**ogical **D**ouble **Q**word) выполняют побайтовые сдвиги.

Таблица 7.13. — Команды сдвига SSE

Мнемоника	**Описание**
PSLL[W|D|Q]	Выполняет (побитовый) логический сдвиг влево каждого элемента данных в операнде назначения (регистр XMM), заполняя младшие биты нулями. Количество битов для сдвига задаётся исходным операндом, который может быть областью памяти, регистром XMM или непосредственным операндом.
PSRA[W|D]	Выполняет (побитовый) арифметический сдвиг вправо каждого элемента данных в операнде назначения (регистр XMM), заполняя старшие биты исходным значением бита знака. Количество битов для сдвига задаётся исходным операндом, который может быть областью памяти, регистром XMM или непосредственным операндом.
PSRL[W|D|Q]	Выполняет (побитовый) логический сдвиг вправо каждого элемента данных в операнде назначения (регистр XMM), заполняя старшие биты нулями. Количество битов для сдвига задаётся исходным операндом, который может быть областью памяти, регистром XMM или непосредственным операндом.
PSLLDQ	Выполняет побайтовый сдвиг влево упакованного целого значения в регистре XMM, заполняя младшие байты нулями. Количество сдвигов задаётся непосредственным операндом.

PSRLDQ	Выполняет побайтовый сдвиг вправо упакованного целого значения в регистре XMM, заполняя старшие байты нулями. Количество сдвигов задаётся непосредственным операндом.

7.3.7 Команды перестановки

В таблице 7.14 перечислены команды, которые обеспечивают перераспределение упакованных данных.

Таблица 7.14. — Команды перестановки SSE

Мнемоника	**Описание**
PSHUFD	Копирует двойные слова из исходного операнда (регистр XMM или область памяти) в целевой операнд (регистр XMM), используя порядок, заданный непосредственным операндом.
PSHUFHW	Копирует старшие слова из исходного операнда (регистр XMM или область памяти) в старшие слова целевого операнда (регистр XMM), используя порядок, заданный непосредственным операндом.
PSHUFLW	Копирует младшие слова из исходного операнда (регистр XMM или область памяти) в младшие слова целевого операнда (регистр XMM), используя порядок, заданный непосредственным операндом.
PSHUFB	Переставляет байты в операнде назначения (регистр XMM) в соответствии с управляющей маской, заданной исходным операндом, который может быть областью памяти или регистром XMM.
SHUFP[S|D]	Перемещает указанные элементы в исходном (регистр XMM или область памяти) и целевом операндах (регистр XMM) в целевой операнд. 8-битный непосредственный операнд указывает, какие элементы следует переместить.

Команда PSHUFD (**P**acked **SHUF**fle **D**word) имеет формат

```
pshufd xmm, xmm/m128, imm8
```

Биты 2 *i* и 2 *i* + 1 в 8-битном непосредственном операнде кодируют индекс одного из четырёх упакованных двойных слов исходного операнда, которое будет помещено в двойное слово с индексом *i* операнда назначения. Например,

```
; xmm0: [ ?? | ?? | ?? | ?? ]
; xmm1: [ d3 | d2 | d1 | d0 ]
pshufd xmm0, xmm1, 10110001b
; xmm0: [ d2 | d3 | d0 | d1 ]
```

Команды PSHUFHW (**P**acked **SHUF**fle **H**igh **W**ord) и PSHUFLW (**P**acked **SHUF**fle **L**ow **W**ord) обеспечивают аналогичную функциональность для перестановки соответственно старших и младших слов, копируя непереставленную половину напрямую из исходного операнда в операнд назначения, как показано в следующем примере:

```
; xmm0: [  ??  |  ??  |  ??  |  ??  |  ??  |  ??  |  ??  |  ??  ]
```

```
; xmm1: [ hw3 | hw2 | hw1 | hw0 | lw3 | lw2 | lw1 | lw0 ]
pshufhw xmm0, xmm1, 11110000b
; xmm0: [ hw3 | hw3 | hw0 | hw0 | lw3 | lw2 | lw1 | lw0 ]
```

Команда PSHUFB (**P**acked **SHUF**fle **B**yte) имеет двухоперандный формат. Второй операнд этой команды играет роль управляющей маски для перестановки. Каждый *i*-й байт маски интерпретируется следующим образом: если бит 7 равен 0, младшие 4 бита кодируют номер байта из первого операнда, который нужно скопировать в *i*-й байт результата; если бит 7 равен 1, *i*-й байт результата следует обнулить:

```
; xmm0 = [ a15 | a14 | a13 | ... |  a2 |  a1 |  a0 ]
; xmm1 = [   0 |   1 | 128 | ... |   2 | 129 |  14 ]
pshufb xmm0, xmm1
; xmm0 = [  a0 |  a1 |   0 | ... |  a2 |   0 | a14 ]
```

Команды перестановки чисел с плавающей точкой SHUFP[S|D] (**SHUF**fle **P**acked **S**ingle/**D**ouble) используют соответственно 2-битные и 1-битные коды индексов в непосредственном операнде для выбора из упакованных чисел с плавающей точкой одинарной и двойной точности. В этом случае старшие элементы данных берутся из исходного операнда, а младшие элементы данных — из операнда назначения. Например,

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
shufps xmm0, xmm1, 10001101b
; xmm0: [ b2 | b0 | a3 | a1 ]

; xmm0: [ a1 | a0 ]
; xmm1: [ b1 | b0 ]
shufpd xmm0, xmm1, 1
; xmm0: [ b0 | a1 ]
```

7.3.8 Команды распаковки

В таблице 7.15 перечислены команды распаковки, которые чередуют нижние или верхние упакованные элементы данных из исходного операнда (регистр XMM или область памяти) и целевого операнда (регистр XMM).

Таблица 7.15. — Команды распаковки SSE

Мнемоника	**Описание**
PUNPCKH[BW|WD|DQ|QDQ]	Распаковывает и чередует старшие элементы данных исходного и целевого операндов. Эти инструкции могут использоваться для преобразования байтов в слова, слов в двойные слова, двойных слов в четверные слова или четверных слов в двойные четверные слова.
PUNPCKL[BW|WD|DQ|QDQ]	Распаковывает и чередует младшие элементы данных исходного и целевого операндов. Эти инструкции могут

	использоваться для преобразования байтов в слова, слов в двойные слова, двойных слов в четверные слова или четверных слов в двойные четверные слова.
UNPCKHP[S|D]	Распаковывает и чередует старшие элементы исходного и конечного операндов и помещает результат в конечный операнд.
UNPCKLP[S|D]	Распаковывает и чередует младшие элементы исходного и конечного операндов и помещает результат в конечный операнд.

Пример чередования слов, выполняемый командой PUNPCKHWD (**P**acked **UNP**a**CK** **H**igh **W**ord to **D**word):

```
; xmm0: [ a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 ]
; xmm1: [ b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 ]
punpckhwd xmm0, xmm1
; xmm0: [ b7 | a7 | b6 | a6 | b5 | a5 | b4 | a4 ]
```

Если исходный операнд состоит полностью из нулей, эта инструкция фактически выполняет расширение нулями упакованных элементов данных в операнде назначения. Аналогично работает команда распаковки нижних упакованных двойных слов PUNPCKLDQ (**P**acked **UNP**a**CK** **L**ow **D**word **Q**word):

```
; xmm0: [ a3 | a2 | a1 | a0 ]
; xmm1: [ b3 | b2 | b1 | b0 ]
punpckhwd xmm0, xmm1
; xmm0: [ b1 | a1 | b0 | a0 ]
```

Инструкции распаковки чисел с плавающей точкой выполняют аналогичное чередование элементов.

При определённых условиях инструкции распаковки, перестановки и некоторые инструкции перемещения данных выполняют одно и то же преобразование данных, как показано ниже.

```
shufps xmm0, xmm1, 68   <=>  movlhps  xmm0, xmm1
shufps xmm0, xmm0, 80   <=>  unpcklps xmm0, xmm0
shufps xmm0, xmm0, 160  <=>  movsldup xmm0, xmm0
shufps xmm0, xmm0, 238  <=>  movhlps  xmm0, xmm0
shufps xmm0, xmm0, 245  <=>  movshdup xmm0, xmm0
shufps xmm0, xmm0, 250  <=>  unpckhps xmm0, xmm0
shufpd xmm0, xmm0, 0    <=>  movddup  xmm0, xmm0
shufpd xmm0, xmm1, 0    <=>  unpcklpd xmm0, xmm1
shufpd xmm0, xmm1, 3    <=>  unpckhpd xmm0, xmm1
```

Так как время выполнения инструкций различается на разных процессорах, эти эквивалентности могут использоваться для оптимизации последовательностей инструкций по перестановке данных для конкретных целевых микроархитектур.

7.4 Система команд AVX

Большинство команд SSE, работающих с 128-битными операндами, имеют аналоги в системе команд AVX. Почти все такие команды имеют также форму, которая может использоваться с 256-битными операндами. Многие команды SSE модернизированы в AVX для использования более гибкого трехоперандного синтаксиса.

Например, инструкция SSE `mulps xmm0, xmm1` выполняет умножение упакованных чисел с плавающей точкой одинарной точности, содержащихся в регистрах `xmm0` и `xmm1`, и сохраняет результат в регистр `xmm0`. Эквивалентная трехоперандная инструкция AVX — это `vmulps xmm0, xmm0, xmm1`. Другой пример — инструкция SSE для сложения упакованных целочисленных байтов `paddb xmm0, xmm1`. Соответствующая ей инструкция AVX — `vpaddb xmm0, xmm0, xmm1`. В обоих примерах содержимое регистра `xmm0` модифицируется. Трехоперандный формат команд AVX допускает чтобы регистр назначения отличался от регистров-источников. Например, инструкции `vmulps xmm0, xmm1, xmm2` и `vpaddb xmm0, xmm1, xmm2`, не изменяют значения в регистрах `xmm1` и `xmm2`.

Кроме того, в AVX добавились совершенно новые команды, не имеющие прямых аналогов в SSE.

Совмещение (aliasing) наборов регистров XMM и YMM приводит к некоторым особенностям, которые должен учитывать программист. Первая проблема связана с тем, как процессор обрабатывает старшие 128 бит регистра YMM, когда происходит запись в регистр XMM. При выполнении инструкции SSE, использующей регистр XMM как операнд назначения, верхние 128 бит соответствующего регистра YMM никогда не изменяются. Однако эквивалентная инструкция AVX обнуляет верхние 128 бит соответствующего регистра YMM. Например, рассмотрим следующие варианты команды [V]CVTPS2PD (**C**on**V**er**T** **P**acked **S**ingle-Precision to **P**acked **D**ouble-Precision):

```
 cvtps2pd xmm0, xmm1
vcvtps2pd xmm0, xmm1
vcvtps2pd ymm0, ymm1
```

Команда SSE `cvtps2pd` преобразует два упакованных числа с плавающей точкой одинарной точности из младшей половины регистра `xmm1` в числа с плавающей точкой двойной точности и сохраняет результат в регистре `xmm0`. Эта инструкция не изменяет старшие 128 бит регистра `ymm0`. Первая инструкция `vcvtps2pd` выполняет ту же операцию преобразования чисел с плавающей точкой из одинарной точности в двойную, но дополнительно обнуляет старшие 128 бит регистра `ymm0`. Вторая инструкция `vcvtps2pd` преобразует четыре упакованных числа с плавающей точкой одинарной точности из младших 128 бит регистра `ymm1` в числа с плавающей точкой двойной точности и сохраняет результат в регистре `ymm0`.

Архитектура AVX ослабляет требования SSE к выравниванию упакованных операндов в памяти. За исключением команд, которые явно указывают на

необходимость выровненного операнда (например, VMOVAPS, VMOVDQA), правильное выравнивание 128-битных или 256-битных операндов в памяти не является обязательным. Однако такие операнды всегда следует выравнивать по возможности, чтобы предотвратить задержки при доступе к невыровненным операндам в памяти.

В одной программе допускается вместе использовать инструкции SSE и AVX, однако такое смешивание следует свести к минимуму, чтобы избежать снижения производительности. Процессору может потребоваться дополнительное время, чтобы сохранить верхние 128 бит каждого регистра YMM при переходе от выполнения инструкций AVX к инструкциям SSE.

Полностью избежать подобных накладных расходов можно с помощью инструкции VZEROUPPER, которая обнуляет верхние 128 бит всех регистров YMM. Эту инструкцию следует использовать перед любым переходом от 256-битного кода AVX (то есть любой инструкции AVX, использующей регистр YMM) к коду SSE. Можно также использовать инструкцию VZEROALL для обнуление всех регистров YMM.

Далее представлен краткий обзор новых команд AVX не имеющих аналогов в SSE.

Команды распространения данных. Команды этого типа копируют (транслируют) одно значение данных в несколько элементов упакованного операнда назначения. Подобные команды доступны для всех типов упакованных данных, включая числа с плавающей точкой одинарной и двойной точности, а также целые числа. В таблице 7.16 приведён обзор данных команд.

Таблица 7.16. — Команды AVX для распространения данных

Мнемоника	**Описание**
VBROADCASTSS	Копирует значение с плавающей точкой одинарной точности во все элементы целевого операнда.
VBROADCASTSD	Копирует значение с плавающей точкой двойной точности во все элементы целевого операнда.
VBROADCASTF128	Копирует упакованное 128-битное значение с плавающей точкой из памяти в нижнюю и верхнюю 128-битные части целевого операнда.
VBROADCASTI128	Копирует упакованное 128-битное целое значение из памяти в нижнюю и верхнюю 128-битные части целевого операнда.
VPBROADCAST[B|W|D|Q]	Копирует целое значение размером 8, 16, 32 или 64 бита во все элементы целевого операнда.

Команды смешивания. Такого рода команды условно объединяют элементы двух упакованных типов данных. В таблице 7.17 приведена одна команда данного типа.

Таблица 7.17. — Команды AVX для смешивания данных

Мнемоника	Описание
VPBLENDD	Смешивает 32-битные двойные слова из двух исходных операндов с помощью непосредственной маски. Каждый бит в маске соответствует одному элементу: если бит равен 1 — выбирается элемент из второго источника, если 0 — из первого.

Работу команды VPBLENDD поясняет следующий пример:

```
; ymm1 (src1):  [ a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 ]
; ymm2 (src2):  [ b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 ]
; imm8 (mask):  10110101b
VPBLENDD ymm0, ymm1, ymm2, 10110101b
; ymm0 (dest):  [ b7 | a6 | b5 | b4 | a3 | b2 | a1 | b0 ]
```

Команды перестановки. Данная группа включает инструкции, которые переупорядочивают или дублируют элементы упакованного типа данных. Поддерживаются различные типы упакованных данных, включая двойные слова, числа с плавающей точкой одинарной и двойной точности (смотри таблицу 7.18).

Таблица 7.18. — Команды AVX для перестановки данных

Мнемоника	Описание
VPERMD	Переставляет 32-битные элементы второго исходного операнда, используя индексы, заданные первым исходным операндом.
VPERMPS	Переставляет значения с плавающей точкой одинарной точности второго исходного операнда, используя индексы, заданные первым исходным операндом.
VPERMPD	Переставляет значения с плавающей точкой двойной точности первого исходного операнда, используя 2-битные индексы, заданные непосредственным операндом.
VPERMQ	Переставляет 64-битные элементы первого исходного операнда, используя 2-битные индексы, заданные непосредственным операндом.
VPERM2I128	Переставляет упакованные 128-битные целые значения из первых двух исходных операндов, используя индексы, заданные непосредственной маской.
VPERM2F128	Переставляет упакованные 128-битные значения с плавающей точкой из первых двух исходных операндов, используя индексы, заданные непосредственной маской.
VPERMILPD	Переставляет значения с плавающей точкой двойной точности в первом исходном операнде, используя управляющее значение, заданное вторым исходным операндом. Перестановки выполняются независимо для каждой 128-битной дорожки.
VPERMILPS	Переставляет значения с плавающей точкой одинарной точности в первом исходном операнде, используя управляющее значение, заданное вторым исходным операндом. Перестановки выполняются независимо для каждой 128-битной дорожки.

Команды VPERMD и VPERMPS переставляют 32-битные элементы:

```
; ymm2 (src):      [ d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 ]
; ymm1 (indices):  [  7 |  0 |  5 |  2 |  3 |  6 |  1 |  4 ]
VPERMD ymm0, ymm1, ymm2
; ymm0 (dest):     [ d7 | d0 | d5 | d2 | d3 | d6 | d1 | d4 ]
```

Команды VPERMPD и VPERMQ переставляют 64-битные элементы:

```
; ymm1 (src):      [ q3 | q2 | q1 | q0 ]
; imm8 (control):  00'11'10'01b (selects q0, q3, q2, q1)
VPERMPD ymm0, ymm1, 00111001b
; ymm0 (dest):     [ q0 | q3 | q2 | q1 ]
```

Семантика команд VPERM2I128 и VPERM2F128 описывается следующим псевдокодом (`SRC1` и `SRC2` — первый и второй исходные операнды; `DEST` — операнд назначения; `IMM8` — непосредственный операнд-маска, управляющий перестановкой):

```
CASE IMM8[1:0] of
0: DEST[127:0] := SRC1[127:0]
1: DEST[127:0] := SRC1[255:128]
2: DEST[127:0] := SRC2[127:0]
3: DEST[127:0] := SRC2[255:128]
ESAC
CASE IMM8[5:4] of
0: DEST[255:128] := SRC1[127:0]
1: DEST[255:128] := SRC1[255:128]
2: DEST[255:128] := SRC2[127:0]
3: DEST[255:128] := SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] := 0
FI
IF (imm8[7])
DEST[255:128] := 0
FI
```

Ниже показаны конкретные примеры:

```
; ymm1 (src1):    [ H1 | L1 ]
; ymm2 (src2):    [ H2 | L2 ]
; imm8 (control): 00'10'00'01b
VPERM2I128 ymm0, ymm1, ymm2, 00100001b
; ymm0 (dest):    [ L2 | H1 ]

; ymm1 (src1):    [ H1 | L1 ]
; ymm2 (src2):    [ H2 | L2 ]
; imm8 (control): 00'10'10'01b
VPERM2I128 ymm0, ymm1, ymm2, 00101001b
```

```
; ymm0 (dest):    [ L2 |  0 ]
```

VPERMILP[S|D] переставляют элементы независимо в пределах каждой 128-битной части:

```
; ymm1 (src):      [ s7 s6 s5 s4 | s3 s2 s1 s0 ]
; imm8 (control):  10'01'11'00b
;                  в нижней половине выбирает : s2, s1, s3, s0
;                  в верхней половине выбирает: s6, s5, s7, s4
VPERMILPS ymm0, ymm1, 10011100b
; ymm0 (dest):     [ s6 s5 s7 s4 | s2 s1 s3 s0 ]

; ymm1 (src):      [ q3 q2 | q1 q0 ]
; imm8 (control):  ****1'1'0'1b
VPERMILPD ymm0, ymm1, 1101b
; ymm0 (dest):     [ q3 q3 | q0 q1 ]
```

Команды извлечения и вставки. В таблице 7.19 приведены инструкции, которые копируют 128-битные упакованные целочисленные значения между регистром YMM и регистром XMM или областью памяти.

Таблица 7.19. — Команды AVX для извлечения и вставки

Мнемоника	Описание
VEXTRACTI128	Извлекает младшее или старшее упакованное 128-битное целое значение из исходного операнда и копирует его в целевой операнд. Значение для извлечения выбирается непосредственным операндом.
VINSERTI128	Вставляет упакованное 128-битное целое значение из второго исходного операнда в целевой операнд. Местоположение в целевом операнде (нижние или верхние 128 бит) задаётся непосредственным операндом. Оставшийся элемент целевого операнда заполняется соответствующим элементом из первого исходного операнда.

Команды маскированного перемещения. Данная группа включает инструкции, которые выполняют условное перемещение элементов в упакованном значении данных. Управляющая маска определяет, копируется ли конкретный элемент из исходного операнда в операнд назначения. Элемент копируется, если старший бит соответствующего элемента маски равен 1, в противном случае в соответствующий элемент операнда назначения записывается ноль. В таблице 7.20 перечислены команды данной группы.

Таблица 7.20. — Команды AVX для маскированного перемещения

Мнемоника	Описание
VMASKMOVP[S|D]	Условно копирует элементы с плавающей точкой из второго исходного операнда в соответствующие элементы целевого операнда в соответствии с управляющей маской, заданной первым исходным операндом.

VPMASKMOVD	Условно копирует 32-битные элементы из второго исходного операнда в соответствующие элементы целевого операнда в соответствии с управляющей маской, заданной первым исходным операндом.
VPMASKMOVQ	Условно копирует 64-битные элементы из второго исходного операнда в соответствующие элементы целевого операнда в соответствии с управляющей маской, заданной первым исходным операндом.

Все команды этой группы функционируют схожим образом, только VMASKMOVPS и VMASKMOVD условно копируют 32-битные элементы, тогда как VMASKMOVPD и VMASKMOVQ условно копируют 64-битные элементы:

```
; mem:               [ m7 | m6 | m5 | m4 | m3 | m2 | m1 | m0 ]
; ymm1 (bit mask):   [1...|0...|1...|1...|0...|1...|0...|1...]
VMASKMOVPS ymm0, ymm1, [mem]
; ymm0 (dest):       [ m7 |  0 | m5 | m4 |  0 | m2 |  0 | m0 ]

; mem:               [ m3 | m2 | m1 | m0 ]
; ymm1 (bit mask):   [0...|1...|0...|1...]
VMASKMOVPD ymm0, ymm1, [mem]
; ymm0 (dest):       [  0 | m2 |  0 | m0 ]
```

Команды переменного сдвига битов. В таблице 7.21 приведен обзор команд, которые выполняют арифметические или логические сдвиги упакованных двойных или четверных слов. Количество двоичных разрядов для сдвига может быть разным для каждого упакованного элемента.

Таблица 7.21. — Команды AVX для сдвига

Мнемоника	**Описание**
VPSLLVD VPSLLVQ	Выполняет побитовый сдвиг влево каждого 32/64-битного элемента данных первого исходного операнда. Количество битов для сдвига задаётся соответствующим элементом данных второго исходного операнда.
VPSRAVD	Выполняет побитовый арифметический сдвиг вправо каждого 32-битного элемента данных первого исходного операнда (освободившиеся верхние биты заполняются первоначальным битом знака элемента). Количество битов для сдвига задаётся соответствующим элементом данных второго исходного операнда.
VPSRLVD VPSRLVQ	Выполняет побитовый сдвиг вправо для каждого 32/64-битного элемента данных первого исходного операнда. Количество битов для сдвига задаётся соответствующим элементом данных второго исходного операнда.

Команды сбора данных. Эта группа включает команды, которые условно копируют элементы данных из массива в памяти в регистр XMM или YMM. Эти команды используют векторный вариант режима адресации памяти, когда эффективный адрес вычисляется на основе выражения

Base + Index * Scale + Displacement.

Здесь Base — регистр общего назначения, указывающий на начало массива в памяти (база); Scale — масштабный коэффициент соответствующий размеру элемента массива (1, 2, 4 или 8); Index — векторный регистр (XMM или YMM), содержащий знаковые индексы массива (в виде двойных или четверных слов); Displacement — необязательное фиксированное смещение от начала массива.

В зависимости от инструкции, векторный регистр должен содержать 2, 4 или 8 знаковых целочисленных индексов. Эти индексы используются для выбора элементов из массива. Данный режим адресации называется VSIB (vector scale-index-base). Ниже показан пример выполнения команды VGATHERDPS, использующей такого рода адресацию:

```
; rsi -> [ 32.0| 23.0| 47.0|101.0| 67.0| 39.0| 18.0|-75.0|...
             0     1     2     3     4     5     6     7   ...
; xmm0 = [     200.0 |     335.0 |    -144.0 |      16.0 ]
; xmm1 = [       5   |       7   |       3   |       2   ]
; xmm2 = [ 80000000h | 80000000h | 00000000h | 80000000h ]

vgatherdps xmm0, [rsi + xmm1*4], xmm2

; xmm0 = [      39.0 |     -75.0 |    -144.0 |      47.0 ]
; xmm1 = [       5   |       7   |       3   |       2   ]
; xmm2 = [ 00000000h | 00000000h | 00000000h | 00000000h ]
```

Регистр `rsi` указывает на начало массива, содержащего значения с плавающей точкой одинарной точности. Регистр `xmm1` содержит четыре знаковых индекса массива (в виде двойных слов). Регистр `xmm2` содержит маску управления условным копированием (копировать элемент из памяти, если старший бит соответствующего элемента маски равен 1).

Операнд назначения и второй исходный операнд (маска управления копированием) в инструкциях этой группы должны быть регистрами XMM или YMM. Первый исходный операнд указывает компоненты VSIB (то есть: регистр базы массива, коэффициент масштабирования, индексы массива и необязательное смещение). Инструкции этой группы не проверяют корректность индексов массива. Использование недопустимого индекса приведёт к неправильному результату. В таблице 7.22 приведён обзор инструкций этой группы. В таблице каждая мнемоника использует префикс VGATHERD или VGATHERQ для индексов массива в виде двойных слов или четверных слов соответственно.

Таблица 7.22. — Команды AVX для собирания данных

Мнемоника	**Описание**
VGATHER[D|Q]PD	Условно копирует два или четыре значения с плавающей точкой двойной точности из массива, расположенного в памяти, с использованием адресации VSIB.

VGATHER[D|Q]PS	Условно копирует четыре или восемь значений с плавающей точкой одинарной точности из массива, расположенного в памяти, с использованием адресации VSIB.
VGATHER[D|Q]D	Условно копирует четыре или восемь значений двойных слов из массива, расположенного в памяти, с использованием адресации VSIB.
VGATHER[D|Q]Q	Условно копирует два или четыре значения четверных слов из массива, расположенного в памяти, с использованием адресации VSIB.

7.5 Команды FMA

В главе 6 упоминались скалярные варианты команд FMA (**F**used **M**ultiply-**A**dd), выполняют комбинированную операцию умножения и сложения за один шаг. FMA фактически удваивает количество операций с плавающей точкой в секунду (FLOPS), которые может выполнять одно ядро процессора. Команды FMA выполняют однократное округление конечного результата. Это особенно важно в алгоритмах, где ошибки могут накапливаться на протяжении многих итераций. FMA позволяет писать более чистый и компактный код. В таблице 7.23 приведены команды этого типа для упакованных данных. Последние шесть команд в таблице не имеют скалярных версий. Для описания выполняемых операций используются следующие обозначения: src1, src2 и src3 — три исходных операнда; des — целевой операнд (совпадает с src1). Все три операнда — регистры XMM (YMM); третий операнд может быть 128(256)-битной областью в памяти.

Таблица 7.23. — Команды FMA

Мнемоника	**Операция**
VFMADD132P[D|S]	des = src1 * src3 + src2
VFMADD213P[D|S]	des = src2 * src1 + src3
VFMADD231P[D|S]	des = src2 * src3 + src1
VFMSUB132P[D|S]	des = src1 * src3 – src2
VFMSUB213P[D|S]	des = src2 * src1 – src3
VFMSUB231P[D|S]	des = src2 * src3 – src1
VFNMADD132P[D|S]	des = –(src1 * src3) + src2
VFNMADD213P[D|S]	des = –(src2 * src1) + src3
VFNMADD231P[D|S]	des = –(src2 * src3) + src1
VFNMSUB132P[D|S]	des = –(src1 * src3) – src2
VFNMSUB213P[D|S]	des = –(src2 * src1) – src3
VFNMSUB231P[D|S]	des = –(src2 * src3) – src1
VFMADDSUB132P[D|S]	des = src1 * src3 + src2 (нечетные элементы) des = src1 * src3 – src2 (четные элементы)
VFMADDSUB213P[D|S]	des = src2 * src1 + src3 (нечетные элементы) des = src2 * src1 – src3 (четные элементы)
VFMADDSUB231P[D|S]	des = src2 * src3 + src1 (нечетные элементы)

	des = src2 * src3 – src1 (четные элементы)
VFMSUBADD132P[D|S]	des = src1 * src3 – src2 (нечетные элементы) des = src1 * src3 + src2 (четные элементы)
VFMSUBADD213P[D|S]	des = src2 * src1 – src3 (нечетные элементы) des = src2 * src1 + src3 (четные элементы)
VFMSUBADD231P[D|S]	des = src2 * src3 – src1 (нечетные элементы) des = src2 * src3 + src1 (четные элементы)

7.6 Примеры использования команд SSE/AVX

Типичная задача, которая хорошо подходит для векторной обработки, это вычисление суммы двух векторов. Алгоритм для обычного скалярного процессора (SISD, Single Instruction Single Data) выглядит следующим образом:

```
for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

Каждая операция сложения выполняется последовательно. Векторный процессор (SIMD, Single Instruction Multiple Data) способен одновременно выполнить несколько операций сложения (например, четыре) с помощью одной команды. Преобразование скалярной программы в программу для векторного процессора (*векторизацию*) можно представить как раскрутку цикла:

```
for (i = 0; i < N; i += 4) {
    c[i+0] = a[i+0] + b[i+0];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
    c[i+3] = a[i+3] + b[i+3];
}
```

Число итераций цикла уменьшилось в четыре раза. Поскольку четыре операции сложения на каждой итерации независимы, их можно параллельно выполнить одной векторной командой сложения:

```
for (i = 0; i < N; i += 4)
    c[i..i+3] = a[i..i+3] + b[i..i+3]; // Псевдокод
```

Таким образом на уровне ассемблера на каждом шаге цикла необходимо: загрузить в векторный регистр порцию массива `a` (размер порции определяется размером векторного регистра) из памяти; загрузить в другой векторный регистр порцию массива `b`; выполнить векторное сложение двух регистров; сохранить результат в массив `c`. Если `N` не кратно 4, оставшиеся `N % 4` элементов векторов придется обработать обычным скалярным образом. Другой способ — дополнить каждый массив нулевыми значениями до размера кратного 4.

Далее рассматриваются основные шаблоны программирования с использованием векторных команд SSE/AVX.

Загрузка и сохранение данных. Загрузка данных из памяти в регистры SIMD и их сохранение обратно — это первый и последний шаг в любых

вычислениях SIMD. Существуют инструкции как для выровненного, так и для невыровненного доступа к памяти. Выровненный доступ выполняется быстрее, но требует, чтобы размер адреса памяти был кратен размеру вектора (16 байт для SSE, 32 байта для AVX). Примеры загрузки и сохранения упакованных данных с плавающей точкой:

```
; SSE (128 бит)
; rcx = указатель на исходный массив (выровнен по 16 байт)
; rdx = указатель на конечный массив (выровнен по 16 байт)
movaps xmm0, [rcx]    ; Загрузить 4 выровненных float
movups xmm1, [rcx+1]  ; Загрузить 4 невыровненных float
movaps [rdx], xmm0    ; Сохранить 4 выровненных float

; AVX (256 бит)
; rcx = указатель на исходный массив (выровнен по 32 байта)
; rdx = указатель на конечный массив (выровнен по 32 байта)
vmovaps ymm0, [rcx]   ; Загрузить 8 выровненных float
vmovups ymm1, [rcx+1] ; Загрузить 8 невыровненных float
vmovaps [rdx], ymm0   ; Сохранить 8 выровненных float
```

Для загрузки и сохранения целочисленных данных следует использовать команды `[v]movdqa/[v]movdqu`.

Вертикальная арифметика (поэлементные операции). Это наиболее распространенный шаблон SIMD вычислений. Одна операция применяется вертикально к соответствующим элементам двух исходных векторов, а результат сохраняется в конечный вектор. Примеры вертикальных операций над упакованными значениями с плавающей точкой:

```
; SSE (128 бит)
; xmm0 = [a3, a2, a1, a0]
; xmm1 = [b3, b2, b1, b0]
addps xmm0, xmm1  ; Результат в xmm0: [a3+b3, a2+b2, a1+b1, a0+b0]

; AVX (256 бит)
; ymm0 = [a7, a6, a5, a4, a3, a2, a1, a0]
; ymm1 = [b7, b6, b5, b4, b3, b2, b1, b0]
vaddps ymm0, ymm0, ymm1 ; Результат в ymm0: [a7+b7, ..., a0+b0]
```

Для сложения упакованных 32-битных целочисленных значений в этом примере следует использовать команды `[v]paddd`.

Распространение. Этот шаблон подразумевает загрузку одного значения из памяти или регистра общего назначения и распространение его по всем элементам регистра SIMD. Это используется, например, для масштабирования вектора на постоянный коэффициент. Примеры распространения скаляра с плавающей точкой:

```
; SSE (128 бит)
```

```
; RCX = указывает на значение float val в памяти
movss xmm0, [rcx]     ; Загрузить один float val
shufps xmm0, xmm0, 0  ; Распространить: [val, val, val, val]

; AVX (256 бит)
; Загрузить и распространить 1 float во все 8 элементов ymm0
vbroadcastss ymm0, dword ptr [rcx] ; ymm0 = [val, ..., val]
```

Примеры распространения 32-битного целого значения:

```
; SSE (128 бит)
; RCX = указывает на 32-битное значение val в памяти
movd   xmm0, [rcx]    ; Загрузить целое число val
pshufd xmm0, xmm0, 0  ; Распространить: [val, val, val, val]

; AVX (256 бит)
; Загрузить и распространить целое число во все 8 элементов ymm0
vpbroadcastd ymm0, [rcx]  ; ymm0 = [val,...,val]
```

Чтобы распространить однобайтное целое по векторному регистру можно использовать команду перестановки байтов `pshufb`, которая имеет некоторые особенности (см. таблицу 7.14). Ниже показан пример программы, прибавляющей постоянное значение ко всем элементам массива байтов:

```
; SSE: Увеличение каждого элемента массива на константу
     .data
n = 16
a    byte 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8
d    byte 2  ; добавляемая константа

     .code
main proc
     lea rbx, a

     movdqa xmm0, xmmword ptr [rbx]
     movsx  eax, d
     movd   xmm1, eax
     pxor   xmm2, xmm2  ; управляющая маска для перемешивания
     pshufb xmm1, xmm2  ; распространяет младший байт во все слоты
     paddb  xmm0, xmm1  ; поэлементное сложение
     movdqa xmmword ptr [rbx], xmm0

     ret
main endp
     end
```

Горизонтальные операции. Часто возникает необходимость редукции упакованных в векторном регистре значений в одно скалярное значение, например, чтобы найти сумму элементов вектора. Такие операции называют *горизонтальными*, т. к. они выполняются над разными элементами одного векто-

ра, в отличие от *вертикальных* операций, которые выполняются над соответствующими элементами разных векторов. Горизонтальные операции обычно медленнее вертикальных и требуют нескольких шагов.

Горизонтальное суммирование упакованных чисел с плавающей точкой одинарной точности в регистре `xmm0` можно выполнить так:

```
; SSE (128 бит)
; Горизонтальное суммирование   xmm0 = [   f3|   f2|   f1|   f0]
movaps  xmm1, xmm0
shufps  xmm1, xmm0, 10110001b ; xmm1 = [   f2|   f3|   f0|   f1]
addps   xmm0, xmm1            ; xmm0 = [f3+f2|f2+f3|f1+f0|f0+f1]
movhlps xmm1, xmm0            ; xmm1 = [   f2|   f3|f3+f2|f2+f3]
paddd   xmm0, xmm1            ; xmm0 = [ - | - | - |f3+f2+f1+f0]
```

Результат оказывается в младшей дорожке `xmm0`. Команда `movhlps` копирует два старших упакованных элемента второго операнда на места двух младших элементов первого операнда (см. таблицу 7.3). С помощью команды `haddps` (таблица 7.8) этот же пример реализуется проще:

```
; SSE (128 бит)
; Горизонтальное суммирование   xmm0 = [   f3|   f2|   f1|   f0]
haddps xmm0, xmm0             ; xmm0 = [f3+f2|f1+f0|f3+f2|f1+f0]
haddps xmm0, xmm0             ; xmm0 = [  sum|  sum|  sum|  sum]
```

Однако такой способ может быть медленнее.

Для суммирования элементов регистра YMM требуется извлечение и сложение верхней и нижней 128-битных половин.

```
; AVX (256 бит)
; ymm0 = [ f7 f6 f5 f4 | f3 f2 f1 f0 ]
vextractf128 xmm1, ymm0, 1    ; xmm1 = [f7 f6 f5 f4]
vaddps       xmm0, xmm0, xmm1 ; xmm0 = [f7+f3 f6+f2 f5+f1 f4+f0]
vhaddps      xmm0, xmm0, xmm0
vhaddps      xmm0, xmm0, xmm0
; Сумма теперь в нижнем элементе xmm0
```

Горизонтальное суммирование упакованных 32-битных целых значений можно выполнить, комбинируя команды перестановки и сложения:

```
; SSE (128 бит)
; Горизонтальное суммирование   xmm0 = [   a3|   a2|   a1|   a0]
pshufd xmm1, xmm0, 01001110b  ; xmm1 = [   a1|   a0|   a3|   a2]
paddd  xmm0, xmm1             ; xmm0 = [a3+a1|a2+a0|a1+a3|a0+a2]
pshufd xmm1, xmm0, 10110001b  ; xmm1 = [a2+a0|a3+a1|a0+a2|a1+a3]
paddd  xmm0, xmm1             ; xmm0 = все элементы a3+a2+a1+a0

; AVX (256 бит)
; ymm0 = [ f7 f6 f5 f4 | f3 f2 f1 f0 ]
vextracti128 xmm1, ymm0, 1  ; xmm1 = [f7 f6 f5 f4]
```

```
vpaddd    xmm0, xmm0, xmm1  ; xmm0 = [f7+f3 f6+f2 f5+f1 f4+f0]

; Теперь нужно сложить 4 значения в xmm0.
; Применяем шаблон из предыдущего примера для SSE
vpshufd xmm1, xmm0,  4Eh  ; xmm1 = [f5+f1 f4+f0 f7+f3 f6+f2]
vpaddd  xmm0, xmm0, xmm1  ; xmm1 = [.. .. f7+f3+f5+f1 f6+f2+f4+f0]
vpshufd xmm1, xmm0, 0B1h  ; xmm1 = [.. .. f6+f2+f4+f0 f7+f3+f5+f1]
vpaddd  xmm0, xmm0, xmm1  ; xmm0 = 4 копии итоговой суммы
```

Если процессор поддерживает расширение SSSE3, то горизонтальное суммирование упакованных 32-битных целых можно выполнить с помощью команды `phaddd`, которая аналогична команде `haddps` для чисел `float`:

```
; SSSE3
; xmm0 = [ d3 d2 d1 d0 ]
phaddd xmm0, xmm0   ; xmm0 = [ d3+d2 d1+d0 d3+d2 d1+d0 ]
phaddd xmm0, xmm0   ; xmm0 = [ .. .. .. (d3+d2)+(d1+d0)]
```

Условная обработка (маскирование). Данный подход реализует логику if-then-else без использования ветвлений (которые негативно сказываются на производительности). Например, необходимо векторизовать следующий алгоритм:

```
for (i=0..N) C[i] = (A[i] > B[i]) ? X[i] : Y[i];
```

Идея состоит в вычислении результатов обоих ветвей (if и else). Инструкция сравнения записывает в регистр битовую маску, в которой каждый элемент имеет все значения 1 (true) или все 0 (false). Затем эта маска используется для смешивания значений из двух регистров (в которых отдельно хранится результат ветвей if и else) как показано в следующем примере:

```
; SSE (128 бит)
; xmm0 = A, xmm1 = B, xmm2 = X, xmm3 = Y
movaps xmm4, xmm0
cmpps  xmm4, xmm1, 6  ; A > B ? => в xmm4 маска
andps  xmm2, xmm4  ; Обнуляет элементы X для которых маска = false
andnps xmm4, xmm3  ; Обнуляет элементы y для которых маска = true
orps   xmm2, xmm4  ; Объединяет результаты в xmm2

; AVX (256 бит)
; ymm0 = A, ymm1 = B, ymm2 = X, ymm3 = Y
vcmpps ymm4, ymm0, ymm1, 6 ; A > B ? -> в xmm4 маска
vblendvps ymm0, ymm3, ymm2, ymm4 ; Смешивает X и Y используя маску
                                 ; Результат в ymm0
```

Если процессор поддерживает SSE4.1, можно использовать команду `blendvps`, которая эквивалентна команде `vblendvps` из AVX. Аналогичный пример для 32-битных целых чисел:

```
; SSE (128 бит)
; xmm0 = A, xmm1 = B, xmm2 = X, xmm3 = Y
movdqa  xmm4, xmm0
pcmpgtd xmm4, xmm1  ; A > B ? => маска в xmm4
pand    xmm2, xmm4  ; Выбрать элементы из X
pandn   xmm4, xmm3  ; Выбрать элементы из Y
por     xmm2, xmm4  ; Объединить результаты в xmm2

; AVX (256 бит)
; ymm0 = A, ymm1 = B, ymm2 = X, ymm3 = Y
vpcmpgtd ymm4, ymm0, ymm1  ; A > B ? => маска в ymm4
vpblendvb ymm0, ymm3, ymm2, ymm4 ; Смешать X с Y используя маску
                                 ; Результат в ymm0
```

Операции сбора данных. Операции сбора данных загружают данные из несмежных областей памяти в один векторный регистр. Адреса памяти задаются вектором индексов. Это полезно для алгоритмов, использующих табличный поиск (table look-up):

```
for (i=0..N) result[i] = lookup_table[indices[i]];
```

В SSE нет аппаратной поддержки операции сбора данных. Для этого нужно использовать обычный скалярный цикл. В AVX2 были введены инструкции сбора данных, которые часто работают быстрее, чем скалярный цикл, но медленнее, чем загрузка массивов смежных элементов.

```
; AVX2 (256 бит)
; RDI  = базовый адрес массива элементов float
; ymm1 = вектор из 8 целых индексов [i7, i6, ..., i0]
; Хотим загрузить [base+i7*4, ..., base+i0*4] в ymm0.
; ymm2 = маска (все биты 1 чтобы загрузить все элементы).
;        Элементы со всеми битами 0 в маске не загружаются.
vgatherdps ymm0, [rdi + ymm1*4], ymm2
```

Производительность операций сбора данных зависит от паттернов доступа к памяти и может быть низкой, если индексы обращаются к совершенно разным местам памяти. Тем не менее, это, как правило, улучшение по сравнению с последовательным выполнением отдельных скалярных загрузок.

FMA. Команды FMA (смотри таблицу 7.23) объединяют умножение и сложение в одну инструкцию с одним шагом округления. Это повышает производительность за счет уменьшения количества инструкций и повышает точность за счет исключения промежуточного округления. Инструкции FMA доступны на процессорах с поддержкой AVX2.

```
; FMA
; ymm0 = A, ymm1 = B, ymm2 = C
; Вычислить A * B + C и сохранить результат в ymm0
vfmadd213ps ymm0, ymm1, ymm2  ; ymm0 = ymm1 * ymm0 + ymm2
```

```
; Суффикс '213' обозначает порядок операндов: dst = op2*op1+op3
```

Команды FMA не доступны для целых чисел. Тем не менее, существуют похожие инструкции, комбинирующие операции умножения и сложения. Например, команда VPMADDWD умножает упакованные 16-битные слова, получая 32-битные промежуточные слова, а затем складывает соседние пары.

```
; AVX2
; Перемножает 16-битные целые и складывает соседние результаты
; ymm0 = [..., a3,a2, a1,a0] (по 16 бит)
; ymm1 = [..., b3,b2, b1,b0] (по 16 бит)
vpmaddwd ymm0, ymm0, ymm1
; Результат в ymm0: [..., a2*b2+a3*b3, a0*b0+a1*b1] (по 32 бита)
```

Задания для самостоятельной и индивидуальной работы

6.1 Выполнить индивидуальный вариант задания используя векторные операции SSE.

1. Заменить нулем каждый отрицательный элемент заданного массива.
2. Для каждого элемента заданного массива, если значение элемента меньше m заменить его на m, если значение элемента больше M заменить его на M. Значения m и M заданы.
3. Заданы два массива A, B и значение t. Для каждого i, если элемент $A[i] > t$, прибавить к этому элементу $B[i]$.
4. Задан массив A. Заменить каждый элемент массива его абсолютным значением.
5. Заданы два массива A и B. Для каждого i если $A[i] < B[i]$, $A[i]$ заменить его квадратом, иначе — квадратным корнем.
6. Задан массив A. Если $A[i] > 100$, умножить $A[i]$ на 2, иначе поделить на 2.
7. Для заданных массивов A, B и C для каждого i, если $A[i]$ четное, $C[i] := A[i] + B[i]$, иначе $C[i] := A[i] – B[i]$.
8. Задан массив A, нижняя граница диапазона L и верхняя граница U. Обнулить значения элементов массива, выходящих за пределы диапазона $[L, U]$.
9. Заданы массивы A и B однобайтных беззнаковых целых чисел. Сложить соответствующие элементы массивов, но в случае переполнения, заменить сумму значением 255.
10. Заданы два массива A, B и пороговое значение t. Создать массив C такой что, если $A[i] > t$, $C[i] := A[i]$, иначе $C[i] := B[i]$.
11. Заданы три массива A, B и C. Если $A[i] > 0$, $C[i] := (A[i] + B[i]) / 2$, иначе $C[i] := (A[i] – B[i]) / 2$.

12. Для заданного массива A и порога t: если $A[i] \geq t$, $A[i] := 1$, иначе $A[i] := 0$.

6.2 Выполнить задания используя операции SSE для упакованных чисел с плавающей точкой.

1. Вычислить декартово расстояние между двумя заданными векторами **x** и **y**: $d = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2}$.
2. Вычислить «манхэттенское расстояние» между двумя заданными векторами **x** и **y**: $d = \sum_{k=1}^{n}|x_k - y_k|$.
3. Вычислить косинус угла между двумя заданными векторами **x** и **y**: $cos\,\theta = \sum_{k=1}^{n} x_k y_k \Big/ \left(\sqrt{\sum_{k=1}^{n} x_k^2}\sqrt{\sum_{k=1}^{n} y_k^2}\right)$.
4. Вычислить дисперсию линейной аппроксимации по методу наименьших квадратов $S^2 = \sum_{k=1}^{n}(y_k - a_0 - a_1 x_k)^2$, где a_0, a_1 — коэффициенты многочлена первой степени (считать заданными); $y_1, y_2, \ldots, y_n$ — известные приближенные значения аппроксимируемой функции в заданных точках $x_1, x_2, \ldots, x_n$.
5. Вычислить взвешенную сумму квадратов разностей для заданных векторов **x** и **y**: $E = \sum_{k=1}^{n} w_k(x_k - y_k)^2$. Значения весов считать заданными.
6. Вычислить выборочное среднеквадратическое отклонение заданного массива значений: $s = \sqrt{\frac{1}{n-1}\sum_{k=1}^{n}(x_k - \overline{x})^2}$, где $\overline{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$ — выборочное среднее.
7. Нормировать заданный вектор **x**, т. е. разделить каждый его элемент на длину вектора $\|\mathbf{x}\| = \sqrt{\sum_{k=1}^{n} x_k^2}$.
8. Применить фильтр «скользящего среднего» 4-го порядка к заданному входному сигналу. Отсчеты входного сигнала хранятся в виде массива значений с плавающей точкой x_k. Получить массив отсчетов выходного сигнала используя формулу $y_k = (x_k + x_{k-1} + x_{k-2} + x_{k-3})/4$.
9. Вычислить свёртку входного сигнала с ядром фильтра длины m: $y_k = \sum_{j=0}^{m-1} h_j\, x_{k-j}$, где h_j — коэффициенты фильтра (считать заданными).
10. Вычислить произведение матрицы на вектор: $y_i = \sum_{j=1}^{m} A_{ij}\, x_j,\ i = 1, \ldots, n$, где A — заданная матрица $n \times m$, x — вектор длины m.
11. Вычислить внешнее произведение векторов **x** и **y** длины n, т. е. матричное произведение вектора-столбца на вектор-строку:

$$\mathbf{x}\mathbf{y}^{\mathrm{T}} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} [y_1 \quad \cdots \quad y_n] = \begin{bmatrix} x_1 y_1 & \cdots & x_1 y_n \\ \vdots & \ddots & \vdots \\ x_n y_1 & \cdots & x_n y_n \end{bmatrix}$$

6.3 Вычислить значение многочлена заданного порядка *n* по схеме Горнера во всех точках вектора **x**, используя операции SSE для упакованных чисел с плавающей точкой.

1. $y = 10 + \sum_{k=1}^{n} kx^{2k}$
2. $y = 5 + \sum_{k=1}^{n} kx^{2k-1}$
3. $y = 10 + \sum_{k=1}^{n} 2kx^{2k}$
4. $y = 10 + \sum_{k=1}^{n} (2k-1)x^{2k-1}$
5. $y = 5 + \sum_{k=1}^{n} (-1)^k kx^{2k-1}$
6. $y = 10 + \sum_{k=1}^{n} (-1)^{k+1} 2kx^{2k}$
7. $y = 10 + \sum_{k=1}^{n} (-1)^{k+1} (2k-1)x^{2k-1}$
8. $y = 6 + \sum_{k=1}^{n} (-1)^k kx^{2k}$
9. $y = 10 + \sum_{k=1}^{n} (2k-1)x^{2k}$
10. $y = 5 + \sum_{k=1}^{n} 2kx^{2k-1}$
11. $y = \sum_{k=1}^{n} (-1)^k kx^k$
12. $y = 1 + \sum_{k=1}^{n} (-1)^k 2kx^k$

7.1 Выполнить задание 6.1 используя векторные операции AVX.

7.2 Выполнить задание 6.2 используя векторные операции AVX.

7.3 Выполнить задание 6.3 используя векторные операции AVX/FMA.

Лабораторная работа № 8

8 ИССЛЕДОВАНИЕ КЭШ ПАМЯТИ

Цель работы: экспериментально исследовать характеристики кэш-памяти процессора (среднее время доступа, объем, пропускная способность).

8.1 Программные и аппаратные таймеры

Для измерения времени выполнения обычно используют два разных таймера, которые предоставляются всеми современными платформами:

Системный высокоточный таймер. Это системный таймер, который обычно реализован как простой счётчик количества тиков, прошедших с некоторой произвольной начальной даты, называемой эпохой. Этот таймер монотонный, то есть его значение всегда увеличивается. Он имеет разрешение до наносекунд и согласован между всеми процессорами. Подходит для измерения событий продолжительностью более одной микросекунды. Системное время можно получить из ОС с помощью системного вызова. Системный таймер не зависит от частоты процессора. Де-факто стандартом для доступа к системному таймеру в C++ является использование std::chrono:

```
#include <chrono>
// Возвращает истекшее время в наносекундах
double timeWithChrono() {
    using namespace std::chrono;
    auto start = high_resolution_clock::now();
    // выполнить что-либо
    auto end = high_resolution_clock::now();
    duration<double, std::nano> elapsed_time = end - start;
    return elapsed_time.count();
}
```

Счётчик тактов (TSC, Time Stamp Counter). Это аппаратный таймер, реализованный как регистр процессора. Он монотонный и работает с постоянной частотой, то есть не учитывает изменения частоты CPU. У каждого процессора есть свой TSC, который представляет собой количество прошедших опорных тактов. Подходит для измерения коротких событий — от наносекунд до примерно минуты. Значение TSC можно получить с помощью встроенной функции компилятора `__rdtsc()`, которая использует ассемблерную инструкцию RDTSC.

```
#include <intrin.h>
// Возвращает количество прошедших опорных тактов
uint64_t timeWithTSC() {
    uint64_t start = __rdtsc();
    // выполнить что-либо
    return __rdtsc() - start;
}
```

Выбор таймера зависит от того, как долго длится измеряемое событие. Если нужно измерить что-то за очень короткий промежуток времени, TSC обеспечит лучшую точность. Напротив, бессмысленно использовать TSC для измерения программы, работающей часами. Если не требуется точность на уровне тактов, системного таймера будет достаточно в большинстве случаев. Важно учитывать, что доступ к системному таймеру обычно имеет большую задержку, чем к TSC.

8.2 Оценка латентности и емкости каждого уровня кэш-памяти

Чтобы найти размер каждого уровня кэш-памяти можно применить следующую стратегию. Сначала необходимо прочитать *N* байт из памяти, что приведет к помещению их в кэш. Затем выполнить повторное чтение этих же данных. Измерив время выполнения повторного чтения и разделив его на количество операций чтения, найдем *среднее время доступа* к кэш-памяти (*латентность*). Если все *N* байт помещаются в кэше, то повторное чтение будет быстрым (нет промахов). Если байты не помещаются, то некоторые или все повторные операции чтения будут приводить к промахам.

Если *N* не превышает размер кэша первого уровня L1, можно ожидать, что повторное чтение будет быстрым (время доступа к L1). Если *N* больше размера кэша L1, но не превышает размер кэша второго уровня (L2), должно наблюдаться несколько более медленное время доступа (оценка времени доступа к кэшу L2) и т. д. Результат эксперимента может выглядеть как на рисунке 8.1. Нижняя «ступенька» отражает среднее время доступа, связанное с попаданиями в кэш L1, вторая «ступенька» — L2, третья — L3. Еще выше можно ожидать «ступеньку», соответствующую времени доступа к основной памяти. Значения *N*, при которых наблюдаются скачки времени доступа, примерно соответствуют размерам различных уровней каш-памяти.

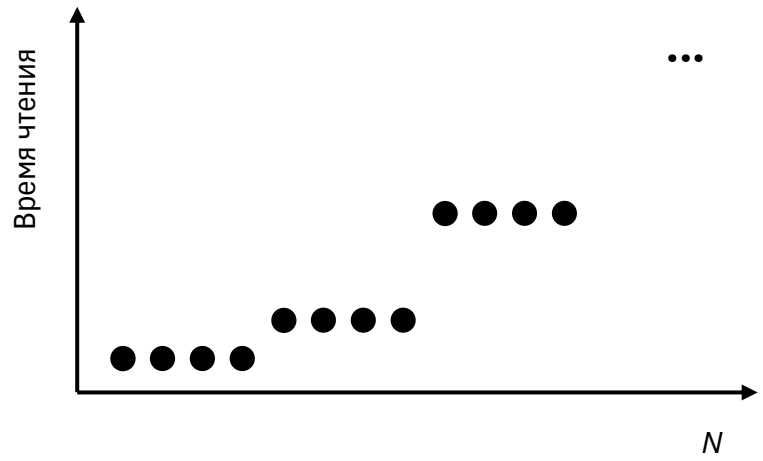

Рисунок 8.1. — Ожидаемая зависимость среднего времени доступа при чтении от объема читаемых данных в байтах

Описанную стратегию можно реализовать в виде следующего алгоритма:

1. Начать с небольшого размера буфера N (например, 4 КБ).
2. Выделить массив размером N байт.
3. «Прогреть» кэш, запустив тестовый цикл один раз.
4. Стартовать таймер.
5. Запустить тестовый цикл, который многократно обращается к элементам массива.
6. Записать размер N и время прошедшее от старта, деленное на количество обращений к массиву.
7. Удалить буфер.
8. Увеличить N (например, в два раза) и повторить шаг 2, пока размер буфера не превысит максимальный (больше типичного размера кэша L3, например, 64 МБ).

При реализации тестового цикла в C++ следует иметь в виду возможные оптимизации компилятора. Например, следующий цикл компилятор может удалить, т. к. результат переменной `c` не используется:

```
for (int i = 0; i < N; ++i) {
    char c = buffer[i];
}
```

Чтобы предотвратить удаление цикла, можно объявить переменную `c` волатильной:

```
for (int i = 0; i < N; ++i) {
    volatile char c = buffer[i];
}
```

Ключевое слово `volatile` сообщает компилятору, что значение переменной может меняться неожиданным образом, например, из другого потока или аппаратного прерывания. Другой способ сообщить компилятору, что результат переменной будет использоваться — это имитировать вывод ее значения:

```
int c = 0;
for (int i = 0; i < N; ++i) {
    c = buffer[i];
}
if (time(nullptr) == 0) std::cout << c;
```

Здесь условие `if` заведомо ложно, поэтому переменная `c` в действительности никогда не выведется в поток, однако компилятор об этом не знает.

Еще одна проблема реализации последовательного чтения из массива связана с микроархитектурными особенностями процессоров. Современные процессоры выполняют команды вне очереди, запускают несколько инструкций за такт и могут одновременно держать от 5 до 50 незавершённых операций

загрузки, ожидающих ответа от памяти. В результате многие промахи кэша могут обрабатываться примерно за то же время, что и один промах, что снижает кажущееся время промаха более чем в 10 раз. Это искажает результаты измерений латентности кэша.

Чтобы предотвратить внеочередное исполнение и параллельную обработку загрузок, необходимо создать *зависимость по данным* между операциями обращения к памяти. Для этого вместо обхода массива, можно выполнять обход односвязного списка. Процессор не сможет загрузить следующий элемент, пока не завершена текущая загрузка, потому что адрес следующего элемента берётся из значения текущего.

Кроме того, контроллер кэша часто может выполнять *предвыборку* блока из основной памяти. Например, при обращении к блоку k в кэш также заносится блок $k + 1$. Также блок предвыборки может определять иной постоянный шаг обращений к памяти. Чтобы снизить эффект влияния предвыборки на результаты измерений, можно перемешать связи в односвязном списке, чтобы адрес следующего элемента при последовательном обходе списка было сложно предсказать.

Таким образом, для оценки среднего времени доступа к различным уровням кэша можно предложить следующий улучшенный алгоритм.

1. Задать `byte_size` — предполагаемый размер кэша в байтах; `stride` — смещение между соседними указателями (в количестве 8-байтных слов).

2. Выделить буфер памяти `buf` размером `byte_size` байт.

3. В буфере создать циклическую цепочку указателей, расположенных на расстоянии `stride * 8` байт друг от друга (8 байт — размер адреса). Порядок соединения указателей будет задавать необходимый паттерн обращений к памяти. Например простую линейную цепочку указателей можно создать с помощью следующей функции:

```
void** create_linear_chain(size_t byte_size, size_t stride) {
    size_t len = byte_size / sizeof(void*);
    void** buf = new void*[len];
    size_t n = len / stride;

    for (size_t i = 1; i < n; ++i)
    {
        buf[(i - 1) * stride] = &buf[i * stride];
    }
    buf[(n - 1) * stride] = &buf[0 * stride];

    return buf;
}
```

Чтобы перемешать связи между указателями, можно создать массив индексов от 0 до n и перемешать их в случайном порядке. Затем воспользоваться приве-

денным выше циклом, в котором вместо индекса *i* использовать *i*-й элемент из массива индексов.

4. Измерить время выполнения заданного количества обращений `count` к памяти при движении по цепочке указателей:

```
void** p = (void**) buf[0];
while (count-- > 0) {
   p = (void**) *p;
}
```

Среднее время доступа равно отношению измеренного времени к `count`.

В результатах измерений неизбежно будет присутствовать шум, т. к. операционная система будет вызывать прерывания и переключения контекста между разными процессами. Чтобы минимизировать шум, может потребоваться запустить внутренний тестовый цикл много раз и взять минимальное зафиксированное время. Шум только увеличивает время, поэтому именно минимум ближе всего к истинному невозмущенному значению.

Кроме того, операционная система может переключать выполнение тестовой программы между разными ядрами процессора. Чтобы этого избежать, необходимо привязать тестовый процесс к одному ядру. Это можно сделать, например, запуская приложение в консоли с помощью команды «`start /affinity 1 cache_test.exe`». Эта команда запускает программу `cache_test.exe` на ядре 0. Число после `/affinity` представляет собой битовую маску, с помощью которой выбираются необходимые ядра.

Задания для самостоятельной и индивидуальной работы

8.1 Реализуйте измерение среднего времени доступа к кэш-памяти для разных значений `byte_size` (от 1 КБ до 128 МБ) и `stride` (от 1 до 32) используя циклическую цепочку указателей (линейную и рандомизированную). Результаты измерений сохраните в файл в формате CSV (Comma-separated values): в каждой строке по три значения отделенных запятыми (`byte_size,stride,time`). Используя сторонние приложения (Excel, Matlab, Gnuplot и т. п.), по сохраненным данным постройте график зависимости `time(byte_size, stride)`. По характерным ступенькам на графике оцените размер каждого уровня кэш-памяти. Узнайте значения размеров уровней кэша вашего процессора средствами операционной системы или из документации в интернет. Сравните с экспериментальной оценкой.

8.2 Используя найденные размеры уровней кэш памяти, экспериментально оцените пропускную способность каждого уровня в ГБ / с. Для этого выполните копирование массива, размер которого точно совпадает с размером выбранного уровня кэша. Для копирования байтов можно воспользоваться стандартной функцией `memcpy()` из заголовка `cstring`.