

Министерство образования и науки Российской Федерации  
федеральное государственное автономное образовательное учреждение высшего  
образования «Национальный исследовательский университет «Московский  
институт электронной техники»

# **Основы архитектуры вычислительных систем и Ассемблера**

с приложением методических указаний к лабораторным работам

Гагарина Л. Г., Кононова А. И.

2018

# Оглавление

<b>Введение</b> . . . . .	12
<b>Глава 1. Понятие вычислительной системы (ВС)</b> . . . . .	15
1.1. Терминология . . . . .	15
1.2. Структурная декомпозиция вычислительной системы . . . . .	17
1.2.1. Единицы измерения . . . . .	22
1.2.2. Порядок следования байтов . . . . .	23
1.2.3. Цикл выполнения команды . . . . .	27
1.2.4. Расположение программ и данных . . . . .	32
1.2.5. Память . . . . .	33
1.2.6. Регистры . . . . .	36
1.3. Иерархическая декомпозиция ВС . . . . .	37
1.3.1. Цифровой логический уровень . . . . .	38
1.3.2. Микроархитектурный уровень . . . . .	39
1.3.3. Уровень архитектуры команд . . . . .	40
1.3.4. Уровень операционной системы . . . . .	41
1.3.5. Уровень ассемблера . . . . .	43
1.3.6. Языки высокого уровня . . . . .	44
1.4. История . . . . .	46
1.4.1. Развитие вычислительной техники . . . . .	47
1.4.2. Операционные системы . . . . .	60
Контрольные вопросы . . . . .	61
<b>Глава 2. Представление данных</b> . . . . .	62
2.1. Качественные и количественные данные . . . . .	62
2.2. История чисел . . . . .	65

2.3.	Позиционные системы счисления . . . . .	70
2.3.1.	Перевод натуральных чисел между позиционными системами счисления . . . . .	72
2.3.2.	Экономичность системы счисления . . . . .	73
2.3.3.	Нецифровые символы в представлении чисел . . . . .	75
2.3.4.	Позиционное представление вещественных чисел . . . . .	76
2.4.	Двоичное представление беззнаковых целых чисел . . . . .	85
2.4.1.	Восьмеричное и шестнадцатеричное представление . . . . .	86
2.4.2.	Беззнаковая арифметика в вычислительных системах . . . . .	90
2.5.	Представление отрицательных чисел . . . . .	96
2.5.1.	Величина со знаком . . . . .	100
2.5.2.	Код с избытком . . . . .	100
2.5.3.	Дополнительный код . . . . .	101
2.5.4.	Знаковая арифметика в вычислительных системах . . . . .	103
2.6.	Альтернативная арифметика . . . . .	107
2.6.1.	Двоично-десятичная арифметика . . . . .	108
2.6.2.	Модулярная арифметика . . . . .	111
2.6.3.	Арифметика с насыщением . . . . .	113
2.7.	Битовые операции . . . . .	114
2.7.1.	Поразрядные операции . . . . .	114
2.7.2.	Расширение целых чисел . . . . .	117
2.7.3.	Битовые сдвиги . . . . .	118
2.8.	Представление вещественных чисел . . . . .	122
2.8.1.	Представление вещественных чисел с фиксированной запятой . . . . .	123
2.8.2.	Представление вещественных чисел с плавающей запятой . . . . .	125
	Контрольные вопросы . . . . .	134

<b>Глава 3. Архитектура команд семейства x86</b>	<b>135</b>
3.1. Развитие линейки x86 и режимы работы	135
3.1.1. История семейства x86	136
3.1.2. Режимы работы процессора	140
3.2. Сегменты памяти	143
3.2.1. Код и статические данные	145
3.2.2. Куча	147
3.2.3. Стек	148
3.3. Регистры	151
3.3.1. Регистры общего назначения, доступные в тридцатидвухбитном режиме	152
3.3.2. Регистры общего назначения, доступные в шестидесятичетырёхбитном режиме	154
3.3.3. Специальные регистры и регистры расширений	156
3.4. Математический сопроцессор (FPU x87)	157
3.4.1. Регистры FPU	157
3.4.2. Исключения FPU	161
3.5. Флаги	164
3.5.1. Флаги основного процессора	165
3.5.2. Флаги FPU	169
3.6. Структура команды и методы адресации	170
3.6.1. Методы адресации	170
3.6.2. Структура команды	171
3.6.3. Общие для тридцатидвухбитного и шестидесятичетырёхбитного режимов сведения об адресации	172
3.6.4. Адресация в тридцатидвухбитном режиме	176
3.6.5. Расширение регистров в шестидесятичетырёхбитном режиме	177

3.6.6.	Операнды и адресация в шестидесятичетырехбитном режиме . . . . .	180
3.6.7.	Адресация относительно указателя команды . . . . .	181
	Контрольные вопросы . . . . .	182
<b>Глава 4.</b>	<b>Связь уровней абстракции . . . . .</b>	<b>184</b>
4.1.	Компиляция . . . . .	185
4.1.1.	Инструменты разработки . . . . .	185
4.1.2.	Этапы компиляции . . . . .	190
4.1.3.	Расширения файлов исходного кода . . . . .	193
4.1.4.	Изменение имени выходного файла . . . . .	194
4.1.5.	Компиляция проекта, состоящего из нескольких модулей	194
4.1.6.	Импорт и экспорт функций . . . . .	195
4.2.	Препроцессор . . . . .	197
4.2.1.	Включение файла . . . . .	198
4.2.2.	Условная компиляция . . . . .	199
4.2.3.	Макросы . . . . .	200
4.3.	Ассемблерные вставки в код C++ . . . . .	205
4.3.1.	Синтаксис ассемблерных вставок в GCC . . . . .	206
4.3.2.	Расширенная форма . . . . .	208
4.3.3.	Практическое использование параметров . . . . .	214
4.3.4.	Ограничения на расположение операнда . . . . .	218
4.3.5.	Модификаторы параметров . . . . .	219
	Контрольные вопросы . . . . .	220
<b>Глава 5.</b>	<b>Синтаксис и команды GNU Assembler x86 . . . . .</b>	<b>222</b>
5.1.	Особенности GNU Assembler . . . . .	223
5.1.1.	Общие правила . . . . .	224

5.1.2.	Основные директивы . . . . .	225
5.1.3.	Порядок операндов . . . . .	228
5.1.4.	Адресация операндов . . . . .	229
5.1.5.	Размер операндов команды . . . . .	234
5.1.6.	Мнемоники . . . . .	236
5.1.7.	Префиксы . . . . .	237
5.2.	Основные команды . . . . .	238
5.2.1.	Общие команды . . . . .	239
5.2.2.	Передача управления, вызов и возврат . . . . .	243
5.2.3.	Обнуление регистра . . . . .	248
5.2.4.	Команды целочисленной арифметики . . . . .	248
5.2.5.	Битовые операции . . . . .	255
5.2.6.	Флаги . . . . .	257
5.2.7.	Условные команды . . . . .	259
5.3.	Команды FPU . . . . .	265
5.3.1.	Внутреннее представление чисел . . . . .	267
5.3.2.	Возможные форматы экспорта-импорта . . . . .	269
5.3.3.	Общие команды . . . . .	271
5.3.4.	Загрузка, выгрузка и пересылка данных . . . . .	272
5.3.5.	Основные арифметические команды . . . . .	279
5.3.6.	Дополнительные арифметические и трансцендентные команды . . . . .	285
5.3.7.	Сравнение вещественных чисел . . . . .	288
	Контрольные вопросы . . . . .	293
<b>Глава 6.</b>	<b>Программирование на языке Ассемблера . . . . .</b>	<b>294</b>
6.1.	Структура программы на ассемблере . . . . .	294
6.1.1.	Программирование с использованием libc . . . . .	295

6.1.2.	Программирование без libc . . . . .	298
6.2.	Подпрограммы и функции . . . . .	301
6.2.1.	Требования к вызовам функций . . . . .	302
6.2.2.	Механизм вызова подпрограммы . . . . .	303
6.2.3.	Соглашения о вызовах . . . . .	307
6.2.4.	Описание функций на ассемблере . . . . .	311
6.2.5.	Импорт функций из модулей на ассемблере в код на C++	315
6.2.6.	Импорт функций из модулей на C++ в код на ассемблере	317
6.2.7.	Искажение имён при компиляции . . . . .	319
6.2.8.	Системные вызовы . . . . .	323
6.3.	Программирование нелинейных алгоритмов и взаимодействие со структурами данных . . . . .	326
6.3.1.	Условие с операторами в одной ветви . . . . .	326
6.3.2.	Условие с операторами в двух ветвях . . . . .	329
6.3.3.	Цикл . . . . .	332
6.3.4.	Массивы . . . . .	334
6.3.5.	Структуры и объекты . . . . .	342
	Контрольные вопросы . . . . .	348
<b>Глава 7.</b>	<b>Программирование на языке высокого уровня: C++ . . . .</b>	<b>349</b>
7.1.	Структура программы . . . . .	350
7.2.	Типы данных . . . . .	351
7.2.1.	Целые типы . . . . .	352
7.2.2.	Вещественные типы . . . . .	356
7.2.3.	Специальные типы . . . . .	358
7.2.4.	Указатели . . . . .	358
7.3.	Преобразование типов . . . . .	359
7.4.	Литералы C++ . . . . .	362

7.4.1.	Целые . . . . .	363
7.4.2.	Вещественные . . . . .	364
7.4.3.	Строки . . . . .	365
7.5.	Средства автоматизации C++ . . . . .	366
7.5.1.	Шаблоны C++ . . . . .	366
7.5.2.	Макросы препроцессора C/C++ . . . . .	368
7.6.	Ввод-вывод . . . . .	369
7.6.1.	Ввод-вывод в поток . . . . .	369
7.6.2.	Ввод-вывод с помощью libc . . . . .	371
7.7.	Отладочная печать . . . . .	374
7.7.1.	Средства исследования переменных . . . . .	374
7.7.2.	Автоматизация отладочной печати . . . . .	377
	Контрольные вопросы . . . . .	381
<b>Заключение</b>	. . . . .	<b>383</b>
<b>Приложение А. Лабораторный практикум GNU Assembler</b>	. . . . .	<b>384</b>
	Требования к выполнению лабораторных работ . . . . .	384
Л1.	Представление данных в ЭВМ . . . . .	387
Л1.1.	Задание на лабораторную работу . . . . .	387
Л1.2.	Дополнительные бонусные и штрафные баллы . . . . .	390
Л1.3.	Рекомендации . . . . .	391
Л1.4.	Теоретические сведения . . . . .	392
Л1.5.	Вопросы . . . . .	396
Л2.	Отладка кода . . . . .	397
Л2.1.	Задание на лабораторную работу . . . . .	397
Л2.2.	Теоретические сведения . . . . .	399
Л2.3.	Вопросы . . . . .	403



Л3. Модули и функции на ассемблере . . . . .	404
Л3.1. Задание на лабораторную работу . . . . .	404
Л3.2. Теоретические сведения . . . . .	405
Л3.3. Вопросы . . . . .	406
Л4. Использование ассемблерных вставок в программах на С++ . . .	407
Л4.1. Задание на лабораторную работу . . . . .	407
Л4.2. Дополнительные бонусные и штрафные баллы . . . . .	408
Л4.3. Теоретические сведения . . . . .	408
Л4.4. Вопросы . . . . .	409
Л5. Программирование сопроцессора . . . . .	410
Л5.1. Задание на лабораторную работу . . . . .	410
Л5.2. Дополнительные бонусные и штрафные баллы . . . . .	411
Л5.3. Варианты заданий для программы calc . . . . .	411
Л5.4. Теоретические сведения . . . . .	413
Л5.5. Вопросы . . . . .	413
Л6. Программирование ветвящихся и циклических алгоритмов . . . .	414
Л6.1. Задание на лабораторную работу . . . . .	414
Л6.2. Дополнительные бонусные и штрафные баллы . . . . .	416
Л6.3. Теоретические сведения . . . . .	417
Л6.4. Вопросы . . . . .	417
Л7. Использование массивов . . . . .	418
Л7.1. Задание на лабораторную работу . . . . .	418
Л7.2. Дополнительные бонусные и штрафные баллы . . . . .	420
Л7.3. Теоретические сведения . . . . .	420
Л7.4. Вопросы . . . . .	420
<b>Приложение Б. Коды ASCII . . . . .</b>	<b>421</b>

<b>Список литературы . . . . .</b>	<b>422</b>
<b>Список таблиц . . . . .</b>	<b>429</b>
<b>Список иллюстраций . . . . .</b>	<b>431</b>
<b>Предметный указатель . . . . .</b>	<b>434</b>

## Аннотация

«Основы архитектуры вычислительных систем и Ассемблера» (учебное пособие).

Приведён систематизированный курс одной из основной дисциплин специализированной подготовки бакалавров по направлениям 09.03.04 «Программная инженерия» и 01.03.04 «Прикладная математика» с точки зрения прикладного программирования и парадигмы кроссплатформенности.

Рассмотрены основы архитектуры вычислительных систем как системной дисциплины, а также обзор архитектуры популярных процессоров семейства x86, в том числе шестидесятичетырехбитных, представление основных типов данных в памяти компьютера, основные команды набора x86 и синтаксис AT&T. Обозначена проблематика прикладного кроссплатформенного программирования, объединяющего язык высокого уровня и язык Ассемблера.

Строгий стиль изложения сопровождается доступными для понимания пояснениями и многочисленными примерами, а также контрольными вопросами к каждой главе, необходимыми для глубокого усвоения материала. Книга адресована студентам технических специальностей, соискателям степени бакалавра по указанным направлениям, слушателям институтов повышения квалификации; может быть использована для самообразования.

Приложение А содержит методические указания к лабораторным работам.

## Введение

Эта тема придёт,  
вовек не износится,  
только скажет:  
— Отныне гляди на меня! —  
И глядишь на неё,  
и идёшь знаменосцем,  
красношелкий огонь над землёй знамени.

*В. В. Маяковский. Про это*

Программирование на языке Ассемблера в этой книге описано на примере наиболее известной и доступной для экспериментов архитектуры — линейки x86. Исторически сложилось так, что разработчик данной архитектуры — компания Intel — использует один синтаксис языка Ассемблера (он так и называется — синтаксис Intel), а большая часть операционных систем, происходя от больших Unix'ов, предпочитает другой, так называемый синтаксис AT&T.

Синтаксис AT&T по умолчанию использует GNU Assembler (GAS) — неотъемлемая часть коллекции компиляторов GCC, используемая в процессе компиляции с различных языков высокого уровня, в частности, C, C++ и Фортран. GAS вместе с коллекцией GCC портирован более чем на 45 платформ, в том числе — на операционную систему Microsoft Windows для x86-совместимых процессоров (исторически этот порт носит название MinGW), так что распространённое мнение «AT&T — это только под Linux» в корне неверно. Напротив, использование GCC и AT&T позволяет сделать программу с ассемблерными вставками в код C++ столь же переносимой между операционными системами, как и чистый C++, а также облегчит переход на неинтеловские архитектуры.

Большая часть литературы на русском языке, посвящённой Ассемблеру x86, описывает синтаксис Intel, в то время как русскоязычная литература по

синтаксису АТ&Т практически отсутствует. Данное пособие призвано заполнить этот пробел.

В результате изучения курса «Архитектура вычислительных систем» студент будет:

- знать и понимать особенности архитектуры и принципы построения вычислительных систем;
- уметь применять язык низкого уровня Assembler, а также ассемблировать и отлаживать готовые программы на языке ассемблера IBM PC;
- владеть разработкой процедур и ассемблерных модулей в программах на языках высокого уровня.

Учебное пособие адресовано студентам бакалавриата по направлению подготовки 09.03.04 «Программная инженерия».

*Особая благодарность группе «Оргия праведников» (С. А. Калугин),  
а также писателям А. В. Жвалевскому и И. Е. Митько  
за разрешение использовать цитаты из их произведений в эпиграфах.  
Вы делаете этот мир ещё прекраснее!*

## Глава 1

# Понятие вычислительной системы (ВС)

Это требует, чтобы о нём написать. И напишу.

*В. В. Маяковский. Я сам*

Вычислительная система описывается как компонентами этой системы, так и языком программирования, предназначенным для взаимодействия с вычислительной машиной. В данной главе обе точки зрения рассматриваются на примере семейства x86, а также в разрезе истории развития вычислительной техники.

### 1.1. Терминология

Одно и то же равно самому себе.

*М. В. Ломоносов. Элементы  
математической химии*

Перед тем, как рассматривать архитектуру вычислительных систем, приведём определения из основных стандартов и справочной литературы.

**Архитектура системы** в стандарте ANSI/IEEE Std 1471-2000 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems) определяется как фундаментальная организация системы, реализованная в её компонентах, их взаимоотношениях друг с другом и средой, а также в принципах, определяющих её конструкцию (проектирование, дизайн) и развитие [67].

Таким образом, архитектура вычислительной системы реализована в компонентах этой системы и их взаимоотношениях. Что же такое вычислительная система?

**Вычислительная система (ВС)**, согласно Воройскому [29], определяется двояко:

1. Совокупность ЭВМ и средств программного обеспечения, предназначенная для выполнения вычислительных процессов.
2. Любая автоматизированная система, основанная на использовании ЭВМ.

Термин «автоматизированная система» (АС), в свою очередь, словарь Воройского также определяет двумя способами:

1. Совокупность управляемого объекта и автоматических управляющих устройств, в которых часть функций управления выполняет человек-оператор.
2. Комплекс технических, программных, др. средств и персонала, предназначенный для автоматизации различных процессов.

В отличие от автоматической системы не может функционировать без участия человека.

Если же обратиться к комплексу стандартов на автоматизированные системы [31], получим третье определение: автоматизированная система — система, состоящая из персонала и комплекса средств автоматизации его деятельности, реализующая информационную технологию выполнения установленных функций.

Таким образом, вычислительная система глобально может быть представлена как совокупность человека (оператора или программиста), технических средств (вычислительной машины, или компьютера) и связывающих их программных средств.

Соответственно, архитектура вычислительной системы может быть описана двояко:

- **структурная декомпозиция** рассматривает ВС с точки зрения её аппаратных составляющих и физических связей между ними и позволяет выделить её функциональные компоненты, в том числе компоненты, предназначенные



для хранения и обработки программ, и компоненты, взаимодействующие с пользователем;

- **иерархическая декомпозиция** рассматривает ВС с точки зрения её логическо-информационной структуры и описывает языки взаимодействия программиста с программными и техническими средствами системы.

## 1.2. Структурная декомпозиция вычислительной системы

Кура

дура

процедура

состоит из трёх частей:

карбюратор

вентилятор

и коробка скоростей!

*Надпись на парте*

Структурная декомпозиция применяется к аппаратной части ВС — вычислительной машине, или компьютеру.

Если посмотреть на персональный компьютер, то он, как правило, состоит из **системного блока** и **внешних устройств**, среди которых обязательно присутствуют **устройства ввода-вывода**, предназначенные для взаимодействия с пользователем — экран, клавиатура, мышь и т. д.

Внутри системного блока находятся **внутренние устройства**, из которых основными являются:

- системная, или материнская плата;
- центральный процессор;
- оперативная память (оперативное запоминающее устройство — ОЗУ);
- внешняя, или долговременная память (жёсткие диски, SSD и т. д.);

- видеокарта;
- звуковая карта и т. д.

Внутренние устройства защищены корпусом системного блока и получают постоянный ток через блок питания.

Все компоненты компьютера связывает воедино системная плата (рис. 1.1). Важную роль играют входящие в её состав два мощных контроллера-концентратора — северный мост и южный мост, обеспечивающие согласование и передачу информационных потоков между различными компонентами системного блока и внешними устройствами.

**Северный мост** определяет частоту системной шины, тем самым — вид и объём ОЗУ, тип шины видеоадаптера (обычно это PCI Express или AGP), осуществляет обмен между центральным процессором и скоростными устройствами, это

- оперативная память (ОЗУ);
- видеокарта;
- южный мост.

**Южный мост** осуществляет обмен с устройствами по низкоскоростным интерфейсам, это

- часы;
- энергозависимая память (ПЗУ);
- контроллер SATA;
- контроллер IDE;
- контроллер прерываний;
- контроллер USB;
- контроллер прямого доступа.

Внутренняя шина, связывающая северный мост и южный мост, обеспечивает непрерывность потоков информации.

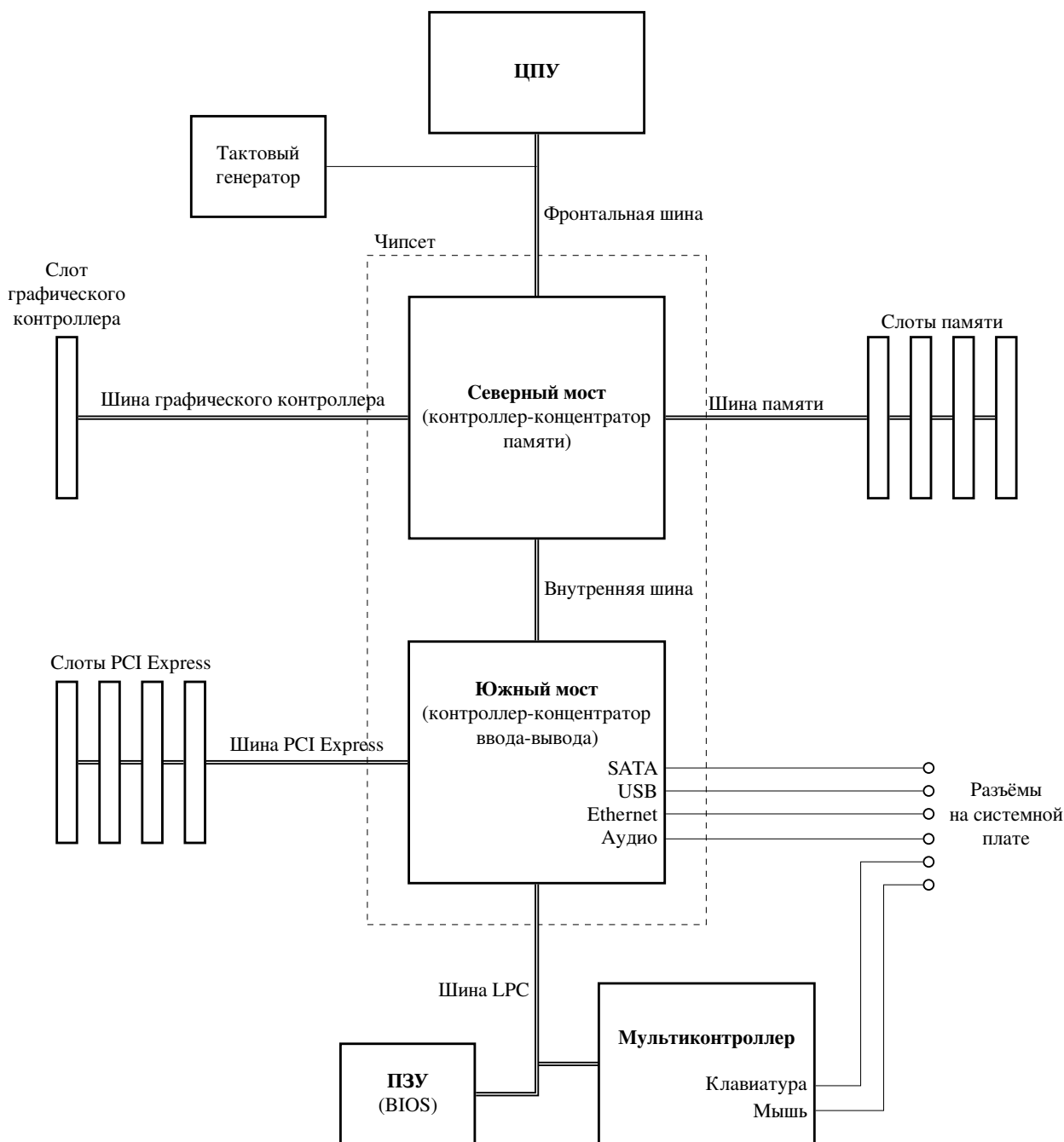


Рис. 1.1. Схема системной платы

Взаимосвязь различных элементов системной платы и устройств обеспечивают различные системные шины (магистраль). Иногда **системной шиной** называют только шину, связывающую процессор и северный мост.

Совокупность проводов системной магистрали можно разбить на четыре группы (рис. 1.2):

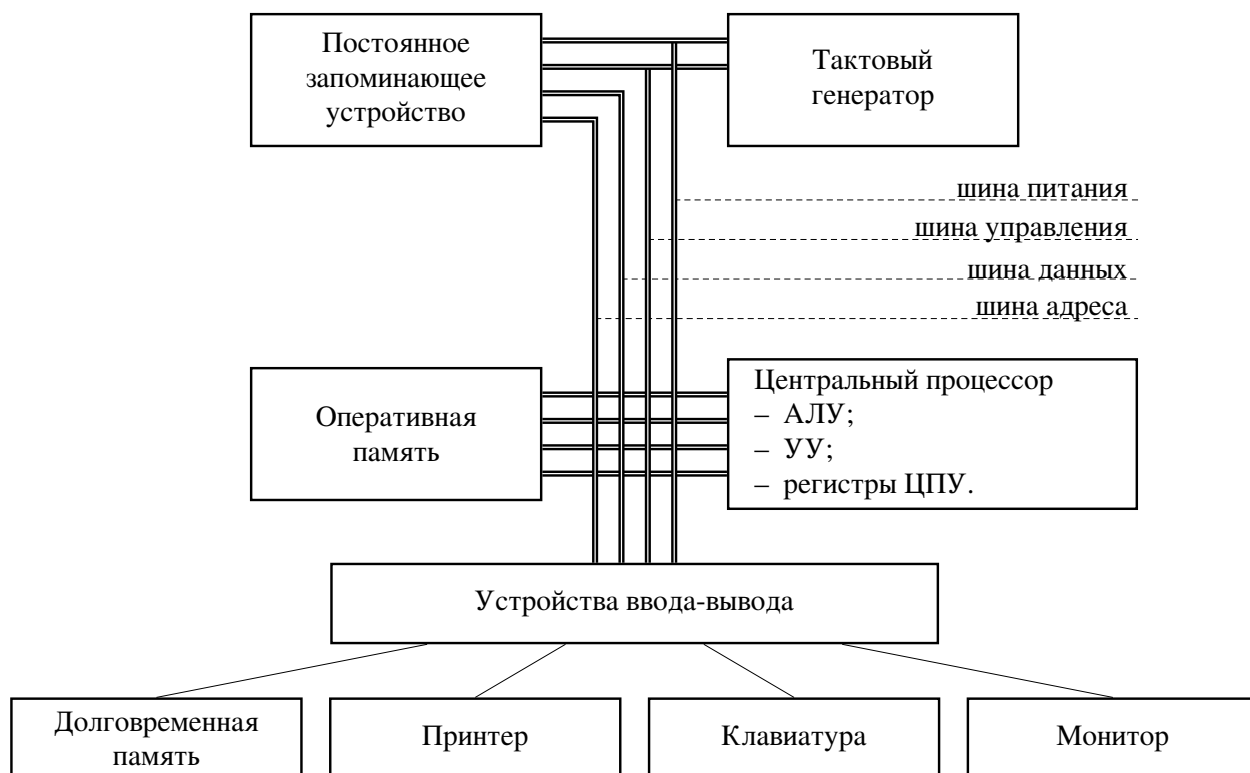


Рис. 1.2. Структура системной шины

- шина питания (так как по этой шине не передаётся никакой информации, её часто опускают на схемах);
- шина управления, используемая для организации обмена самой магистральной;
- шина данных;
- шина адреса.

Шины питания и управления связывают все устройства, в том числе **тактовый генератор**, предназначенный для синхронизации работы различных устройств системной платы. Тактовый генератор задаёт частоту работы процессора как самого быстрого устройства; частоты более медленных устройств являются делителями частоты процессора.

Кроме оперативной, в ВС имеется **постоянное запоминающее устройство (ПЗУ)** — энергонезависимая память, в которой записана неизменяемая информация, которая сохраняется после отключения питания. Там хранится

микропрограмма управления вычислительной машиной. Она обязательно включает программы начальной загрузки и самотестирования.

В постоянном запоминающем устройстве персонального компьютера записывается BIOS (Basic Input/Output System — базовая система ввода/вывода), включающая, кроме загрузчика, также настройки и функции ввода/вывода с помощью программных прерываний. Ввод-вывод при помощи BIOS доступен только в реальном режиме работы компьютера. Современные операционные системы сразу после загрузки переводят компьютер в защищённый режим и не используют BIOS.

Сама операционная система, как и загружаемые ею программы, располагается в **оперативной** (энергозависимой) памяти.

**Центральный процессор** (центральное процессорное устройство, ЦПУ) является «мозгом» ВС. Он исполняет код программ. ЦПУ часто называют просто процессором, иногда — микропроцессором (МП). Исторически микропроцессор — это процессор, выполненный на одной микросхеме; сейчас другие варианты исполнения нерентабельны и эти два термина стали синонимами. Ранее для специализированных процессоров, расширяющих функциональность ЦПУ, использовался термин «сопроцессор», но в настоящее время сопроцессоры либо вошли в состав ЦПУ (в частности, математический сопроцессор x87 входит в ядро современных ЦПУ семейства x86), либо получили иные названия (так, графический сопроцессор обычно называется видеокартой или графическим ускорителем).

Центральный процессор включает

- арифметико-логическое устройство (АЛУ), выполняющее обработку данных;
- управляющее устройство (УУ), декодирующее поступающие в процессор команды и формирующее на их основе сигналы для АЛУ;

- регистры — сверхбыструю память особой структуры и малого объёма, предназначенную для временного хранения данных. Часть регистров может быть использована программистом по своему усмотрению (регистры общего назначения), часть используется для специальных целей.

Также современные процессоры содержат кеш-память (сверхоперативную память), предназначенную для прозрачного временного хранения фрагментов оперативной памяти. Время обращения к кеш-памяти больше, чем к регистрам, но меньше, чем к ОЗУ.

### 1.2.1. Единицы измерения

Базовой единицей измерения информации в современных ЭВМ является **бит** — двоичный разряд.

В соответствии с характеристиками магистрали и регистров вводятся дополнительные платформозависимые единицы измерения:

- машинное **слово** — разрядность регистров процессора и/или шины данных;
- **байт** — минимальный независимо адресуемый набор данных.

Понятие машинного слова возникло раньше понятия байта. Вначале минимально адресуемый блок памяти (байт) и блок, загружаемый или обрабатываемый за один раз (слово) всегда совпадали. В настоящее время машинное слово может быть как равно, так и кратно байту.

В настоящее время байт обычно составляет восемь бит, но существуют DSP-процессоры, для которых байт состоит из шестнадцати или двенадцати бит. На начальных этапах развития вычислительной техники размер байта вообще не был стандартизирован. Впервые термин «байт» был употреблён для совокупности шести битов.

Если необходимо описать именно восемь двоичных разрядов, используется термин **октет**. В частности, эта единица измерения используется при описании сетевых протоколов.

В семействе процессоров x86 используется восьмибитный байт. Длина машинного слова менялась от шестнадцати бит у первых моделей до шестидесяти четырёх у современных. При этом для совместимости документации термин «слово» остался за шестнадцатью битами. Тридцать два бита называют двойным словом, шестьдесят четыре — четверным и так далее.

Далее по тексту везде подразумевается, что байт состоит из восьми бит.

### 1.2.2. Порядок следования байтов

Для начала уточним: в современных ВС память адресуется побайтово, при этом начальный адрес равен нулю, и адрес каждого следующего байта возрастает на единицу. То есть в качестве модели памяти можно рассматривать непрерывную ленту из последовательно расположенных байтов с непрерывно возрастающими на единицу номерами — адресами. Подобные последовательности принято записывать, следуя обычному направлению письма, то есть адреса возрастают слева направо. Адресом слова считается адрес его самого левого байта (далее будем обозначать его  $\zeta$ ).

Кроме того, число в вычислительных системах представляется в двоичной позиционной системе счисления, то есть в виде совокупности двоичных разрядов — битов. Каждый бит имеет свой вес, соответственно которому разряды также можно упорядочить — от младшего к старшему.

$$X = x_0 + x_1 \cdot 2 + \dots + x_k \cdot 2^k, \quad x_i \in \{0,1\} \quad (1.1)$$

Если число включает восемь бит (занимает один байт), то никакой неоднозначности нет — число имеет адрес, равный адресу единственного байта. Биты внутри байта не имеют адресов, но ранжируются по весу.

Если число включает шестнадцать бит, оно записывается двумя байтами. Каждый байт включает восемь смежных по старшинству разрядов числа (рис. 1.3, а).

$$X = \chi_0 + \chi_1 \cdot 256, \quad \chi_i \in \{0,1, \dots, 255\} \quad (1.2)$$

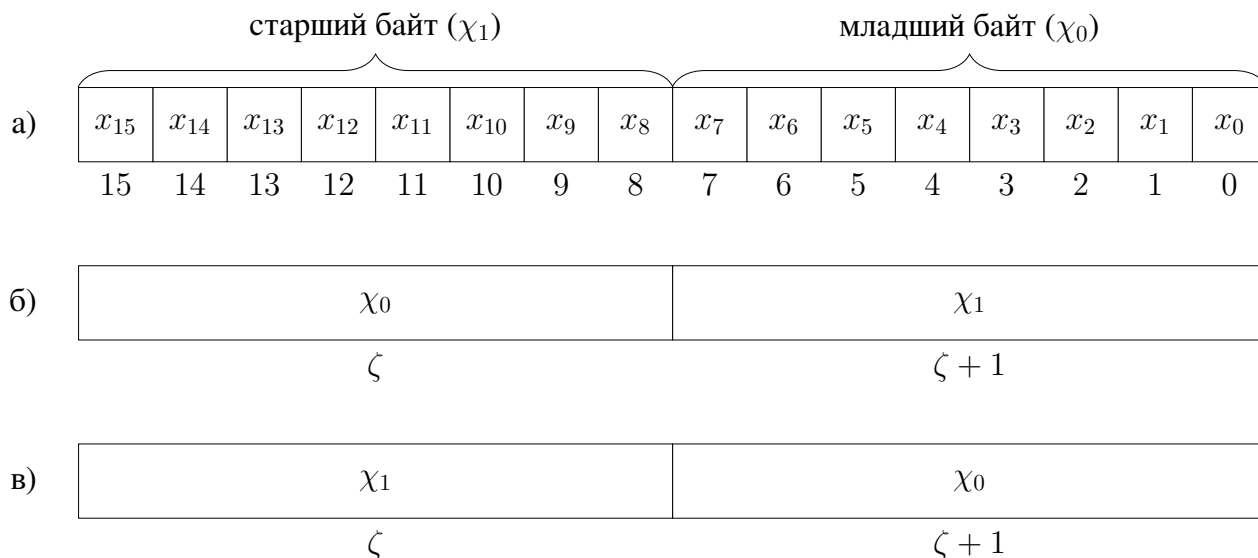


Рис. 1.3. Двухбайтовое число: а) биты старшего и младшего байтов, б) прямой порядок байтов в памяти, в) обратный порядок байтов в памяти

При этом каждому байту числа можно сопоставить и его вес (старшинство его битов в числе), и адрес (расположение в памяти). На любой платформе байты двухбайтового числа (младший  $\chi_0$  и старший  $\chi_1$ ) расположены в памяти рядом, но друг относительно друга они могут располагаться по-разному.

Для двухбайтового числа возможны только два варианта:

- прямой порядок (также называемый little-endian, Intel или VAX) — младший байт слова расположен по младшему адресу (рис. 1.3, б);
- обратный (big-endian, Motorola или сетевой порядок) — младший байт слова расположен по старшему адресу (рис. 1.3, в).

Числа в любой позиционной системе счисления принято записывать в арабской традиции — младший разряд справа, старший слева. Содержимое памяти (дамп памяти), напротив, записывается, следуя европейскому направлению письма — младшие адреса слева, старшие справа. Из-за этого на рис. 1.3 именно прямой порядок выглядит «перевернутым», а обратный — соответствует привычной позиционной записи числа.



Значение каждого байта в дампе записывается двумя шестнадцатеричными цифрами в арабской традиции (как число, которое уже не может быть разделено на части с разными адресами). Соответственно, конкретные числа в дампе памяти выглядят ещё экзотичнее. Пусть необходимо записать в память вычислительной машины с восьмибитным байтом шестнадцатибитное число  $x = 0x0A\ 0B$ . Оно содержит два байта: старший  $\chi_1 = 0A$ , младший  $\chi_0 = 0B$ . Дамп памяти с прямым порядком байтов, соответственно, будет выглядеть как  $\chi_0\chi_1 = 0B\ 0A$  (то есть разряды числа несколько «перетасованы»), с обратным порядком — как  $\chi_1\chi_0 = 0A\ 0B$ .

Если число состоит из нескольких байтов, эти байты в памяти ВС также могут быть расположены друг относительно друга по-разному. Чаще всего используется прямой или обратный порядок (рис. 1.4).

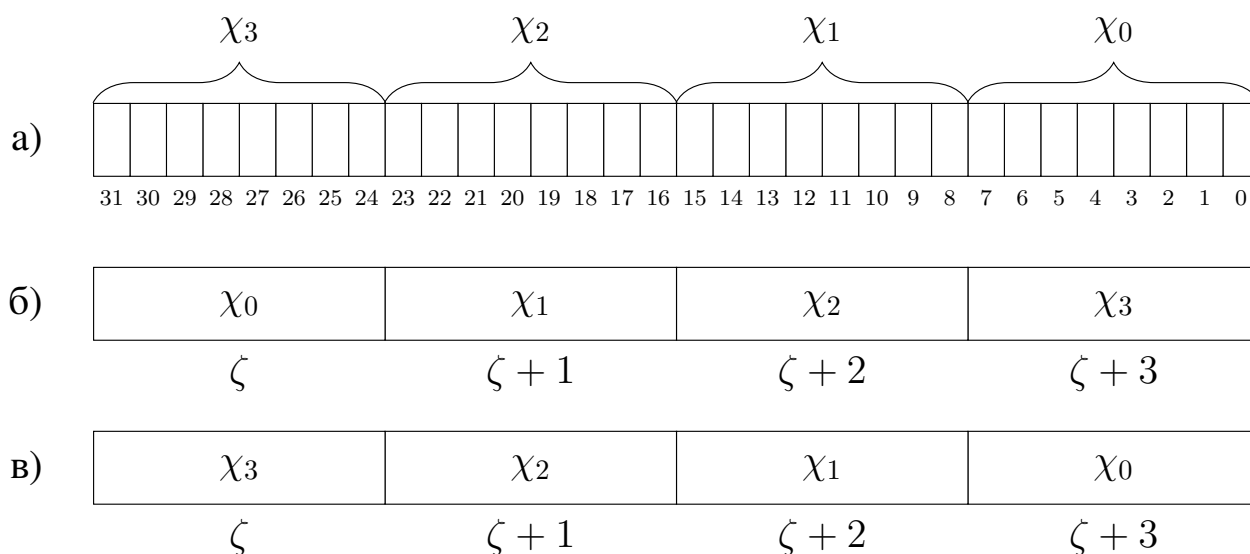


Рис. 1.4. Четырёхбайтовое число: а) байты и биты числа,

б) прямой порядок байтов в памяти,

в) обратный порядок байтов в памяти

Пусть необходимо записать в память вычислительной машины с восьмибитным байтом тридцатидвухбитное число  $x = 0x0A\ 0B\ 0C\ 0D$ . Оно займёт четыре смежных байта с адресами  $\zeta, \zeta + 1, \zeta + 2$  и  $\zeta + 3$ . Наименьший из них (младший)

адрес  $\zeta$  будет адресом числа  $x$ . Рассмотрим, как оно будет расположено в памяти при разных порядках размещения.

При записи с обратным порядком байтов это число при просмотре дампа памяти будет выглядеть как 0A 0B 0C 0D, то есть старший байт 0A будет записан по младшему адресу  $\zeta$  и, соответственно, напечатан первым (левее всего), байт 0B — по адресу  $\zeta + 1$ , 0C — по адресу  $\zeta + 2$ , младший байт 0D окажется записанным по самому старшему адресу  $\zeta + 3$  и при просмотре или печати дампа памяти окажется правее остальных.

При записи с прямым порядком байтов это число будет выглядеть как 0D 0C 0B 0A: младший байт 0D записан по младшему адресу  $\zeta$ , 0C — по адресу  $\zeta + 1$ , 0B — по адресу  $\zeta + 2$ , старший байт 0A — по старшему адресу  $\zeta + 3$ .

Очевидно, что обратный порядок байтов позволяет легко читать числа, записанные в памяти. Обратный порядок принят в протоколе TCP/IP.

Прямой порядок байтов удобен при обработке чисел большой разрядности с помощью процессора малой разрядности, так как позволяет при сложении таких чисел обращаться к памяти последовательно в порядке возрастания адресов, а такие запросы выполняются быстрее (это учитывается в схемной реализации алгоритмов обработки).

Некоторые системы позволяют переключать используемый порядок байтов при помощи переключки на материнской плате или программно (bi-endian, bytesexual).

Также иногда используется смешанный (middle-endian, mixed-endian) порядок байтов: байты в словах расположены в одном порядке, но, если число состоит из нескольких слов, слова располагаются наоборот. В частности, в PDP-11 младший байт слова расположен по младшему адресу, но младшее слово числа — по старшему (PDP-endian), так что наше число 0x0A 0B 0C 0D имеет в памяти вид 0B 0A 0D 0C. Другой вариант смешанного порядка — младший байт слова по старшему адресу, а младшее слово числа — по младшему, — даст 0C 0D 0A 0B.

В процессорах семейства x86 используется прямой порядок байтов (порядок Intel). Он применяется даже к вещественным числам, которые не имеет смысла обрабатывать по частям: число 3F F0 00 00 00 00 00 00 (1.0, то есть единица с плавающей запятой двойной точности) будет записано в памяти как 00 00 00 00 00 00 00 F0 3F.

### 1.2.3. Цикл выполнения команды

Проходящие через ВС потоки информации можно разделить на две основные группы: команды и данные.

Данные представляют собой информацию, подлежащую обработке и, как правило, размещаются в памяти ВС.

Команды предназначены организовать и выполнить обработку данных процессором ВС. Последовательность команд называется программой и также расположена в памяти ВС.

Выполнение команды процессором можно разбить на ряд этапов. Эту последовательность называют циклом выполнения команды, или рабочим циклом процессора.

1. Выборка (загрузка) команды из памяти. Адрес загружаемой команды хранится в специальном регистре — указателе команды (instruction pointer, *ip*). На рис. 1.5 указатель команд хранит адрес команды К-1. Двоичный код выбранной команды К-1 попадает в другой специальный регистр — регистр команд.
2. Декодирование команды. На этом этапе определяется, выбрана ли команда целиком или необходима дозагрузка (разные команды могут иметь различную длину). Когда команда загружена полностью, определяется наличие у неё операндов и их расположение, наличие числового результата и его расположение, а также формируется сигнал для АЛУ в соответствии с типом команды.

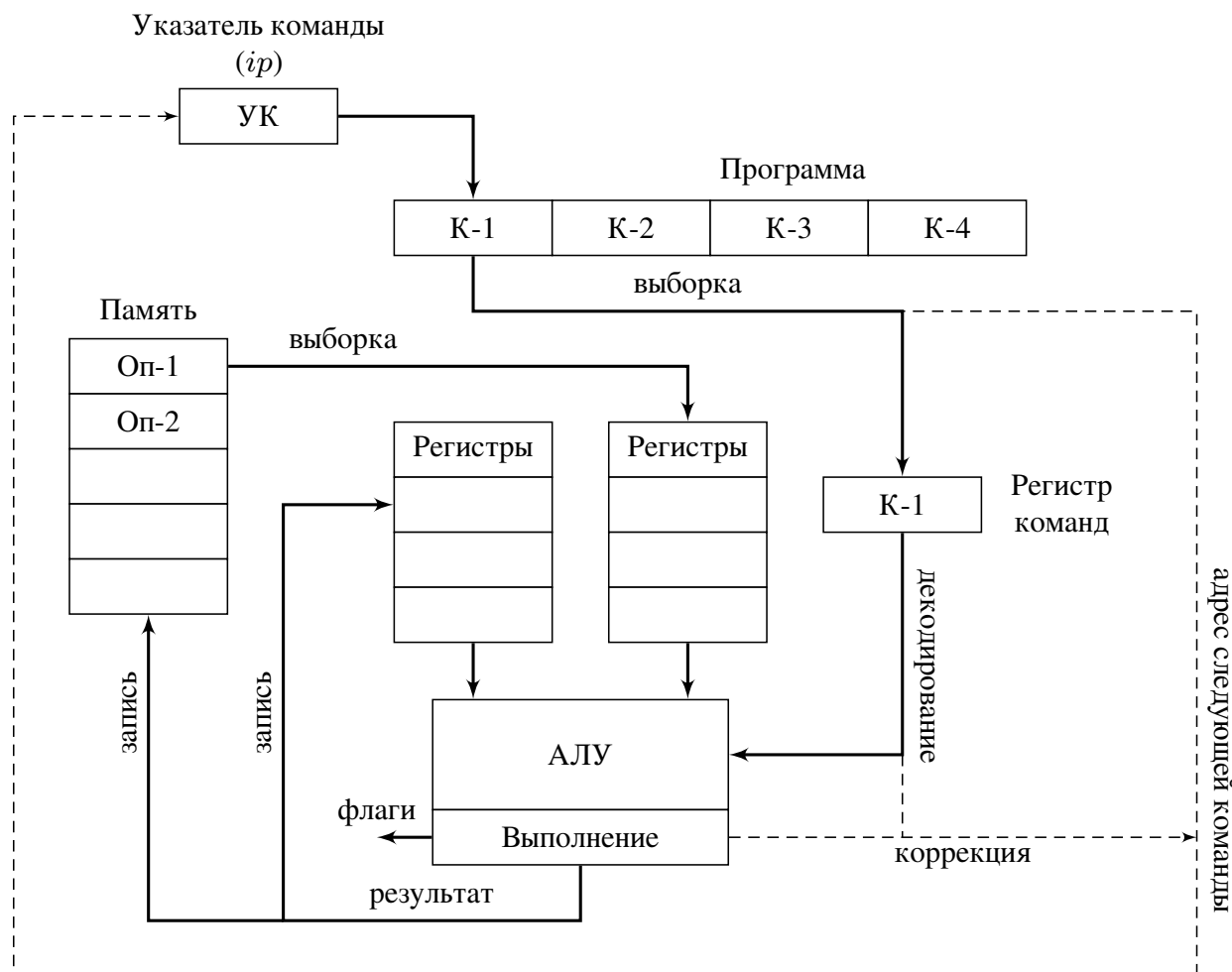


Рис. 1.5. Цикл выполнения команды

3. Выборка операндов. На следующем этапе из памяти загружаются операнды команды, которые затем помещаются в специальные регистры. Если операнды располагаются в регистрах общего назначения, они поступают в АЛУ на этапе выполнения [40].
4. Выполнение инструкции. Этап выполнения команды протекает различным образом для различных команд. В случае арифметических команд операнды поступают из регистров на вход АЛУ, АЛУ выполняет операцию, соответствующую команде, результат записывается в специальный регистр результата, формируются признаки результата (нулевое значение, некорректный результат и так далее).

5. Запись результатов и установка флагов. На этом этапе результат загружается из регистра результата в расположение, определённое при декодировании (это может быть как ячейка памяти, так и регистр общего назначения). Признаки результата записываются в регистр флагов, доступный для анализа дальнейшими командами.
6. Формирование адреса следующей команды. В регистр указателя команд помещается адрес следующей команды. Если К-1 была командой условного или безусловного перехода, вызова или возврата из функции и т. п., адрес следующей команды можно узнать только после выполнения К-1.

## Конвейер

Обработка разных команд при этом может вестись параллельно. Для этого цикл выполнения команды разбивают на несколько стадий — от двух для ранних ЭВМ, в частности, для ЭЦВМ Урал, до нескольких десятков в настоящее время. Выполнение каждой из этих стадий реализуется независимо от других.

При подобной реализации обработка следующей команды может выполняться, не дожидаясь конца текущей; такой способ организации вычислений называется **конвейером**.

Так, в большинстве случаев после команды К-1 будет выполняться команда, непосредственно следующая за ней в программе (К-2) — подобная последовательность называется естественным ходом выполнения. Таким образом, уже после полной выборки К-1 можно обновить указатель команд. Соответственно, если конвейер включает шесть стадий, описанных выше, то,

- пока К-1 будет декодироваться, можно выполнить выборку К-2;
- во время выборки операндов К-1 освободится блок декодирования, так что можно декодировать К-2;
- во время выполнения К-1 можно выбрать из памяти операнды К-2 и так далее.

Если нет сбоев или задержек, время выполнения команды будет определяться временем выполнения самой длинной стадии.

Сбои конвейерной обработки возможны в нескольких случаях:

1. Различные времена выполнения стадий для разных команд. Для решения этой проблемы перед блоками, исполняющими каждую стадию, вставляются блоки-диспетчеры, организующие очередь.
2. Конфликты по данным (в частности, операндом К-2 может быть результат К-1). Подобные зависимости отслеживаются на этапе декодирования и учитываются планировщиком на этапе выполнения. В некоторых процессорах планировщик может изменить порядок выполнения команд так, чтобы избежать зависимостей по данным между соседними командами.
3. Выполняемая команда нарушает естественный ход выполнения программы (например, К-1 может быть командой перехода к К-4). Это приводит к очистке и повторной загрузке конвейера, что существенно снижает быстродействие. Для предотвращения постоянной очистки конвейера в циклах современные процессоры используют различные алгоритмы прогнозирования переходов.

В линейке x86 конвейер впервые появился в процессоре i486 и включал пять стадий, что позволило более чем вдвое увеличить производительность.

Если процессор включает несколько конвейеров, возможна полностью одновременная обработка нескольких команд. Подобные процессоры называются **суперскалярными**. При этом параллельно могут выполняться только команды, не связанные зависимостями по данным. Отслеживание зависимостей и планирование исполнения реализуется внутри суперскалярных процессоров.

Перед выполнением программы её код должен быть загружен в память. Выполнение программы начинается с помещения в указатель команд *ip* адреса той команды, которая должна быть выполнена первой (точки входа).

## Классификация по набору команд

По количеству и структуре команд архитектуры делятся на два основных типа.

1. CISC (complex instruction set computer — компьютер с набором сложных команд) — набор команд огромен и разнообразен, сами команды имеют переменную длину и сложную структуру, а также используют сложные режимы адресации; регистров мало и функции многих из них предопределены.

Это было сделано для упрощения программирования в машинных кодах, компактности программ и удешевления самого процессора.

2. RISC (reduced instruction set computer — компьютер с набором упрощённых команд, архитектуры load/store) — набор команд включает команды простой постоянной структуры и фиксированной длины; при этом процессор содержит множество регистров, так что обращение к памяти производится только для загрузки (load) данных в регистры и выгрузки (store) их оттуда. Такая архитектура позволяет поднять частоту и параллельность и хорошо подходит для компиляции с языка высокого уровня.

Естественным продолжением идеологии RISC являются архитектуры типа VLIW (very long instruction word — сверхдлинное командное слово). Команда VLIW объединяет несколько команд RISC по числу конвейеров процессора; эти команды выполняются параллельно на соответствующих конвейерах.

В отличие от суперскалярных процессоров, где распределение команд по конвейерам происходит во время выполнения специальным устройством в составе процессора, командные слова VLIW формируются компилятором на этапе сборки программы. Это позволяет упростить и удешевить процессор, но усложняет разработку компиляторов и увеличивает длину программы.

### 1.2.4. Расположение программ и данных

Данные и команды поступают в процессор по системной шине из памяти. Соответственно, память может быть общей для команд и данных — в этом случае для связи с процессором достаточно одной общей шины (рис. 1.6, а). Такая архитектура требует меньшего количества элементов и дорожек, поэтому она дешевле и компактнее.

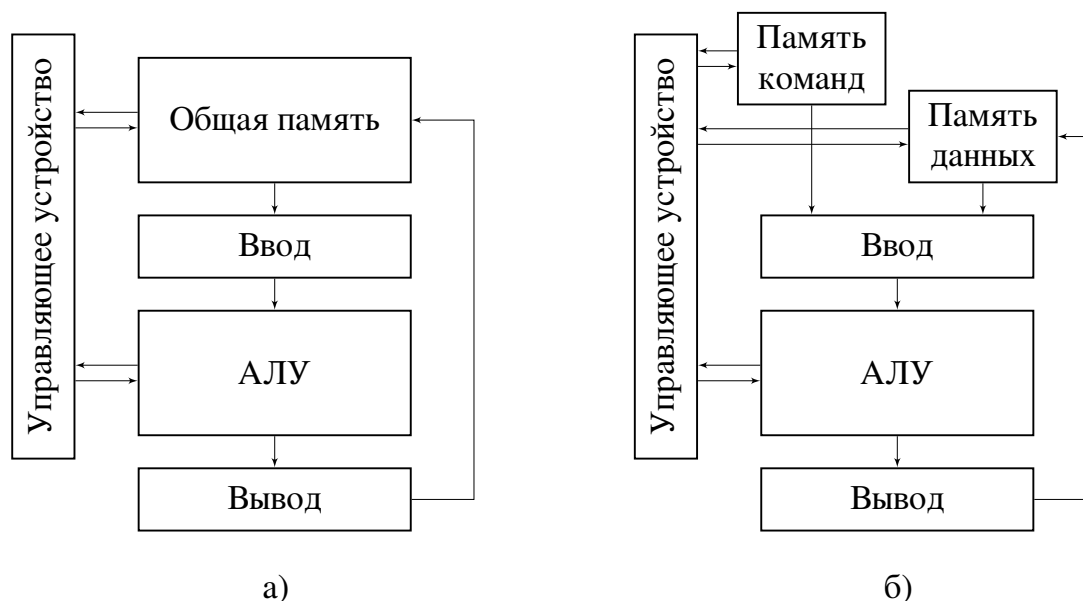


Рис. 1.6. Расположение программ и данных  
в фон-неймановской (а) и гарвардской (б) архитектурах

Идея общей памяти и общей шины для программ и данных впервые была реализована Конрадом Цузе в Германии. В США подобная архитектура разрабатывалась в школе Мура при Пенсильванском университете научным коллективом, куда входили, в частности, Джон Мокли и Джон Преспер Экерт. Незадолго до окончания работ результаты были описаны одним из участников проекта, Джоном фон Нейманом, без указания соавторов, а затем и опубликованы в таком виде куратором со стороны армии, Германом Голдштейном [22]. Так как фон Нейман к тому времени уже был известным математиком, архитектуру с общей



шиной обычно называют фон-неймановской (или принстонской, по основному месту работы фон Неймана).

Общая шина для памяти программ и данных — узкое место фон-неймановской архитектуры.

Ускорить обмен с памятью можно, введя отдельные шины и, соответственно, физически отдельные запоминающие устройства для программ и для данных (рис. 1.6, б). Это дороже и сложнее в реализации, поэтому, хотя сама идея отдельных шин использовалась в позднем проекте Бэббиджа, а практический проект подобной архитектуры разрабатывался в Гарвардском университете США одновременно с проектом школы Мура, **широко** использоваться на практике этот подход стал относительно недавно. Архитектура с отдельными шинами программ и данных обычно называется гарвардской.

В персональных компьютерах используется фон-неймановская архитектура. В первых процессорах линейки x86 и соответствующих системных платах такое решение было использовано для удешевления, в последующих из соображений совместимости также используется единое пространство памяти. При этом современные процессоры имеют отдельную кэш-память для программ и данных.

### 1.2.5. Память

Память в вычислительных системах образуют запоминающие устройства различной природы, имеющие разные характеристики по объёму памяти, по скорости обмена и по времени создания контакта (рис. 1.7).

Самыми быстрыми — но при этом и самыми дорогими — являются регистры. Поэтому объём регистровой памяти очень ограничен.

Оперативное запоминающее устройство — ОЗУ — характеризуется оптимальным соотношением цена-быстродействие. ОЗУ в современных вычислительных системах имеют довольно большие объёмы памяти, но и они не беспредельны.



Рис. 1.7. Иерархия запоминающих устройств. Сверху вниз увеличивается объём и уменьшается скорость обмена

Внешние запоминающие устройства — ВЗУ — характеризуются большим временем создания контакта и низкой скоростью обмена. Но при этом они, во-первых, могут иметь очень большие объёмы, и, во-вторых, несколько ВЗУ могут быть поочерёдно подключены и либо считаны, либо записаны. На заре компьютерной эры программист вручную организовывал обмен с нужными устройствами, учитывая их особенности и протоколы обмена.

Сейчас программисту не надо заботиться об этом. Операционные системы предоставляют ему необходимый объём памяти — в виртуальном адресном пространстве — и программист пользуется им, не заботясь о физических протоколах обмена. Операционная система сама организует и поддерживает соответствие между физическими адресами реальной памяти и адресами виртуальной памяти.

## Плоская модель памяти

Современные операционные системы используют так называемую **плоскую модель памяти**. Каждой исполняющейся программе (процессу) предоставляется диапазон виртуальных адресов от 0 до  $2^N$ , где  $N$  — разрядность системы (32 или 64). Каждому адресу соответствует один байт памяти.

При обращении к определённому виртуальному адресу он транслируется в физический аппаратным механизмом трансляции адресов, контролируемым операционной системой [2]. Виртуальный адрес может быть некорректным либо соответствовать байту в оперативной памяти или специализированной области жёсткого диска (в разделе подкачки; некоторые операционные системы также могут использовать файл подкачки). Операционная система может перемещать данные в оперативной памяти или из памяти на диск и обратно. Виртуальные адреса и, соответственно, указатели в программе при этом не изменяются.

Диапазон доступных физических адресов зависит от объёма установленных ОЗУ и возможностей системной платы. Разрядность физического адреса может составлять как 32, так и 40 или 52 бита [2].

Процесс не может обратиться к физическому адресу, принадлежащему другому процессу, без специальных средств межпроцессного взаимодействия.

## Страничная организация памяти

Виртуальное адресное пространство плоской модели памяти делится на части определённого размера *pagesize* — страницы.

Каждая страница включает непрерывный последовательный диапазон виртуальных адресов  $[\zeta, \zeta + \textit{pagesize})$  и соответствует последовательному фрагменту ОЗУ или раздела подкачки на жёстком диске. Страница может быть вытеснена из ОЗУ на диск или загружена с диска в ОЗУ только целиком, то есть страница — минимальный квант памяти при её распределении между процессами.

Страницы в принципе могут иметь различные атрибуты защиты (разрешения чтения, записи, исполнения и т. д.).

### 1.2.6. Регистры

Чем более объёмно запоминающее устройство, тем, как правило, медленнее оно работает. Оперативная память обладает меньшей скоростью, чем процессор; хотя частоты работы как оперативной памяти, так и процессора непрерывно растут, это соотношение сохраняется. В настоящее время частоты процессоров измеряются в гигагерцах, памяти — в сотнях мегагерц.

Соответственно, использование только данных, расположенных в памяти, сильно замедлило бы работу. Частично обмен данными ускоряется при помощи кеширования, но оно не уравнивает скорости процессора и памяти.

Кроме того, АЛУ процессора не может обрабатывать данные, расположенные непосредственно в оперативной памяти, так что оператор  $++i$  (или соответствующая ему команда ассемблера `inc i`) будет выполняться в несколько приёмов (раздел 1.2.3):

- загрузка значения переменной  $i$  из оперативной памяти в специальную ячейку внутри процессора — регистр;
- расчёт значения  $i + 1$ ;
- выгрузка значения  $i + 1$  из регистра в память по адресу переменной  $i$ ,

причём загрузка-выгрузка в ОЗУ занимает больше времени, чем расчёт значения. Для ускорения работы часто используемые переменные логично хранить прямо в процессоре, выделив для этого некоторое количество регистров, не используемых в цикле обработки команд. Они называются регистрами общего назначения.

В отличие от специальных регистров, недоступных программисту, их можно указать в команде как источник или приёмник значения, то есть использовать как сверхбыстрые ячейки памяти. На уровне машинных команд регистры общего назначения обозначаются короткими номерами, на уровне ассемблера — именами.

Специальные регистры процессора также имеют имена, но они используются только в документации [19]. Значение специальных регистров программист может узнать или модифицировать только косвенно, с помощью специальных команд.

### 1.3. Иерархическая декомпозиция ВС

По мере спуска от потолка скрепы к её полу они  
утрачивают знаки власти, из «образов Бога» делаются  
лучиной для самовара.

*В. Хлебников. Азбука Неба*

Согласно Таненбауму [45], вычислительную систему можно представить в виде ряда уровней абстракции. Каждому уровню соответствует свой язык, программы на котором путём компиляции или интерпретации переводятся на язык нижележащего уровня.

Большинство современных систем включает шесть уровней [45]:

- уровень языка прикладных программистов;
- уровень ассемблера;
- уровень операционной системы;
- уровень архитектуры команд;
- микроархитектурный уровень;
- цифровой логический уровень.

Над этой иерархией языков находится прикладной программист и воплощаемый им алгоритм; под цифровым логическим уровнем — физическая элементная база.

Современная шестиуровневая структура сформировалась далеко не сразу и не является пределом развития. Примитивные счётные устройства (абак, суанпан, русские счёты) рассчитаны на взаимодействие человека напрямую с эле-

ментной базой. Развитие вычислительной техники и программной инженерии ведёт к постепенному наращиванию уровней абстракции.

Иногда между уровнем языка прикладных программистов и уровнем операционной системы находится уровень байт-кода или р-кода, интерпретируемого виртуальной машиной. Он не эквивалентен уровню ассемблера, так как не отражает специфику аппаратной части вычислительной системы.

### 1.3.1. Цифровой логический уровень

На цифровом логическом уровне существуют цифровые **сигналы**, которые могут принимать ограниченный набор значений. Для современных ЭВМ эти значения — 0 и 1.

Соответственно количеству состояний сигнала определяется базовая единица измерения информации. Так как современные ЭВМ двоичны, единицей информации считается бит — один двоичный разряд, который может находиться в двух состояниях — 0 и 1. Для троичных ЭВМ, таких как Сетунь, информация измеряется в тритах.

Сигналы обрабатываются базовыми элементами схем — **логическими вентилями**, преобразующими множество входных сигналов в выходной в соответствии с какой-либо из логических операций (в частности, «и», «или», «не», «и-не»).

Существует множество стандартов изображения логических вентиляей. В отечественной литературе чаще всего используются либо обозначения, описанные в стандарте ИЕС 60617-12 (1997), либо похожие обозначения ГОСТ 2.743-91 из единой системы конструкторской документации (ЕСКД).

Из-за особенностей элементной базы основными для построения схем являются вентили «и-не» и «или-не» (NOR и NAND), реализующие соответственно операции  $\neg(x \vee y)$  и  $\neg(x \wedge y)$  для входных сигналов  $x$  и  $y$ .

Из вентиляей конструируются более сложные элементы, в частности, двоичный полусумматор, рассчитывающий сумму двух входов, формируя два вы-

ходных сигнала: сумму входов по модулю 2 и разряд переноса, или полный одноразрядный сумматор, рассчитывающий сумму трёх входов (на один из них при построении многоразрядного сумматора подаётся перенос из предыдущего разряда).

Цифровой логический уровень — хронологически первый уровень абстракции. Впервые он появился ещё в непрограммируемых счётных машинах, в том числе вавилонских и греческих астрономических калькуляторах, но при этом отсутствует в простых устройствах наподобие счётов, где перенос между разрядами осуществляется вручную.

### **1.3.2. Микроархитектурный уровень**

Микроархитектура процессора — это аппаратная организация и логическая структура микропроцессора:

- регистры — сверхбыстрые ячейки памяти внутри процессора;
- набор микрокоманд;
- управляющие схемы;
- арифметико-логические устройства (АЛУ);
- запоминающие устройства;
- связывающие их информационные магистрали (шины).

В частности, к микроархитектуре относятся все решения, касающиеся цикла выполнения команды: декодирование команды вышележащего уровня в последовательность микрокоманд, конвейер, кеширование команд и данных, прогнозирование переходов и т. п.

Микроархитектурный уровень впервые начал отделяться от уровня архитектуры команд в ЭВМ EDSAC-2 в 1957 г., когда впервые было применено микропрограммирование для реализации команды как последовательности микрокоманд.

В настоящее время микроархитектура процессора качественно отличается от архитектуры уровня команд. Так, на уровне микроархитектуры все совре-

менные процессоры семейства x86 (то есть процессоры с CISC-набором команд) организованы по принципу RISC. Именно, они имеют набор микрокоманд простой структуры и множество регистров, которые поочерёдно играют роль восьми регистров общего назначения за счёт механизма переименования. Это позволяет ускорить выполнение программ, сохраняя при этом совместимость на уровне архитектуры команд.

### **1.3.3. Уровень архитектуры команд**

Уровень архитектуры команд включает

- архитектуру памяти;
- взаимодействие с внешними устройствами ввода/вывода;
- режимы адресации;
- регистры;
- набор машинных команд;
- различные типы внутренних данных (целочисленные, с плавающей запятой и т. д.);
- обработчики прерываний и исключений.

Эти пункты во многом повторяют компоненты микроархитектуры. Различие состоит в том, что уровень архитектуры команд доступен программисту и, как всякий интерфейс, должен соответствовать документации и относительно редко меняется. Микроархитектура скрыта от программиста и может различаться даже у разных партий процессоров одной и той же модели.

В соответствии со структурой памяти, разрядностью шины и регистров определяются платформозависимые единицы измерения — байт и машинное слово.

Программа уровня архитектуры команд — последовательность двоичных машинных кодов. Программы ранних поколений записывались непосредственно в двоичном коде в виде отверстий на перфокартах и перфолентах (отверстие соответствовало единице, нетронутая позиция — нулю). Позже для хранения



программ стали использоваться различные виды долговременной памяти (обычно магнитные ленты и диски), а запись — сокращать с помощью восьмеричного и шестнадцатеричного представления двоичных чисел.

Каждому процессору соответствует свой набор машинных кодов.

В частности, для процессоров семейства x86 опкод пересылки непосредственного тридцатидвухбитного значения в тридцатидвухбитный регистр общего назначения имеет вид B8, после этого кода записывается трёхбитный код регистра, после чего идёт само значение. Соответственно, запись значения CC CC CC CC в регистр с кодом 1 выполняется командой B9 CC CC CC CC. Часто одно и то же действие можно выполнить с помощью различных машинных команд.

Уровень архитектуры команд, отличающий программируемую вычислительную машину (компьютер) от калькулятора или арифмометра, появился в проекте аналитической машины Чарльза Бэббиджа, а впервые реализован был в механическом компьютере Z1 Конрадом Цузе в 1938 г. При этом сама идея цифровых программ появилась задолго до её применения для вычислительных машин. Двоичные программы, записанные на перфолентах, использовались в механических органах — шарманках в XV—XIX вв., на перфокартах — в автоматизированном ткацком станке Жозефа Жаккара (1802 г.).

Программирование в машинных кодах использовалось в течение долгого времени параллельно с ассемблером и языками высокого уровня. Даже в настоящее время бывает необходимо использовать вставки на машинном языке. Это может понадобиться либо в случае, когда ассемблер устарел и не распознаёт мнемоническое обозначение необходимой команды, либо при разработке защитных механизмов — для затруднения дизассемблирования и обратной разработки алгоритма защиты.

#### **1.3.4. Уровень операционной системы**

На уровне операционной системы осуществляется:

- управление памятью (распределение между процессами, организация виртуальной памяти);
- загрузка программ в оперативную память и их выполнение;
- исполнение запросов программ (системные вызовы);
- стандартизованный доступ к периферийным устройствам (устройства ввода-вывода);
- управление доступом к данным на энергонезависимых носителях (таких как жёсткий диск, оптические диски и др.), организованным в той или иной файловой системе;
- обеспечение пользовательского интерфейса;
- сохранение информации об ошибках системы.

Данный уровень является гибридным — большая часть команд его языка совпадает с командами нижележащего уровня (машинного языка) и только некоторые интерпретируются непосредственно операционной системой.

При этом операционная система работает с этим языком и как компилятор (преобразуя исполняемый файл в готовую к выполнению программу в оперативной памяти, что для современных форматов исполняемых файлов не сводится к простому копированию), и как интерпретатор (исполняя так называемые системные вызовы).

Именно различие в системных вызовах и формате исполняемых файлов не позволяет выполнять в Microsoft Windows программы, собранные для того же процессора под GNU/Linux и наоборот.

В частности, системные вызовы используются для получения параметров командной строки (открытие документа двойным щелчком в Microsoft Windows — тоже вызов программы с параметром!) и для завершения работы. Непосредственное обращение к системным вызовам обычно скрыты от прикладного программиста библиотечными функциями-обёртками.

Первой операционной системой можно считать аппаратный супервизор релейного многоядерного компьютера Bell Model V (1946 г.), обеспечивавший загрузку следующей программы из очереди на освободившееся ядро без участия оператора и переключение между перфолентами по команде условного перехода. Позже появилась пакетная операционная система GM-NAA I/O (General Motors & North American Aviation Input/Output system) для IBM 704 (1956 г.) [60].

В конце 1950-х гг. были разработаны первые широко используемые операционные системы FMS (Fortran Monitor System) [45, 60] и SOS (Share 709 System) [12, 60] для IBM 709, которые включали компиляторы (Фортран для FMS и ассемблер для SOS) и обрабатывали системные вызовы, представленные перфокартами со специальным содержанием.

### **1.3.5. Уровень ассемблера**

Программировать в машинных кодах не слишком удобно, даже если записывать их в компактном восьмеричном или шестнадцатеричном виде. Намного удобнее создавать программы, используя понятное человеку символическое представление машинных команд — мнемоники, а также символические имена регистров и адресов в памяти (переменных, меток).

Перевод программы из подобного символического представления в машинные команды реализуется путем трансляции (сборки, ассемблирования), а не интерпретации. Программа-транслятор для такого перевода соответственно называется ассемблером, а язык программирования — языком ассемблера. Язык ассемблера часто сокращённо называют просто ассемблером, как и транслятор.

Так как набор мнемоник — символическое представление набора команд процессора, процессорам с различным набором команд соответствуют разные языки ассемблера.

Синтаксис языка ассемблера также зависит от используемого транслятора, так что для одного и того же процессора могут быть разработаны несколько ассемблеров.

В частности, команда `B9 CC CC CC CC`, то есть команда записи значения `CC CC CC CC` в регистр с кодом 1 для x86 (этому коду для тридцатидвухбитного регистра в тексте программы соответствует имя *ecx*), на уровне ассемблера будет иметь вид `movl $0xCCCCCCCC, %ecx`. Строка `mov` называется мнемоническим обозначением, или мнемоникой команды пересылки. Одной мнемонике может соответствовать несколько машинных команд (в частности, обозначение `mov` объединяет множество команд пересылки данных); кроме того, одна машинная команда может на уровне ассемблера обозначаться несколькими мнемониками (так, `jge` — переход, если больше или равно и `jnl` — переход, если не меньше, — это одна и та же команда)

Считается, что первый ассемблер появился в 1949 г. По разным источникам, он был разработан Дэвидом Джоном Уилером для ЭВМ EDSAC [56] или Джоном Мокли и Джоном Преспером Экертом для ЭВМ BINAC [44].

Иногда ассемблер называют самым старым языком программирования после машинного кода, хотя первый язык высокого уровня появился ещё раньше. Тем не менее, ассемблер заслуженно считается вторым поколением языков программирования.

### 1.3.6. Языки высокого уровня

Третьим поколением считаются языки высокого уровня (ЯВУ), позволяющие программисту описывать алгоритм, а не его реализацию на данной конкретной машине.

Для языка высокого уровня возможна как компиляция до уровней ассемблера или операционной системы (или, иногда, до другого ЯВУ), так и пошаговая интерпретация. При этом программы на некоторых языках традиционно только компилируются (в частности, Паскаль/Delphi, C/C++), на некоторых — только интерпретируются (в частности, это языки командной оболочки, наиболее известным из которых является семейство `sh/bash`), для большинства есть и компиляторы, и интерпретаторы (Python, PHP и т. д.). В некоторых современных

языках высокого уровня (в частности, С#, Java) вводится дополнительный уровень абстракции — программа на ЯВУ компилируется до байт-кода, который затем интерпретируется.

Идея символического языка для прикладного программирования, который не был бы связан с архитектурой конкретной вычислительной машины, почти так же стара, как и само программирование.

Первый язык высокого уровня — Планкалькуль — был разработан Конрадом Цузе в 1943-45 гг., но для него в то время не был разработан компилятор.

Первый компилятор, переводивший программу в алгебраической форме на машинный язык, А-0, был разработан в 1952 г. Грейс Хоппер.

Первый отечественный компилятор с языка высокого уровня ПП-1 (программирующая программа) был разработан в 1954 г. В некоторых источниках считается первым компилятором с языка высокого уровня [46]. Язык ПП-1 (и его позднейшие потомки) был основан на математической нотации [32].

Первым языком высокого уровня, дожившим до настоящего времени под оригинальным именем (но при этом породивший другой известный язык — Бейсик), является Фортран. Он был создан в 1954-1957 гг. группой программистов под руководством Джона Бэкуса в IBM. Также в 1957 г. был создан первый язык функционального программирования APL.

Как конкурент языку от IBM, в 1958 г. группой под руководством Питера Наура был разработан язык Алгол — родоначальник большинства современных языков общего назначения, в частности, семейства, включающего линейки Паскаль/Delphi/С# и В/С/С++. В это же время появился язык обработки списков Lisp, существующий до сих пор под этим именем.

В 1959 г. под руководством Грейс Хоппер был создан язык Кобол, максимально приближённый к английскому языку. Сейчас аналогичную нишу в нашей стране занимает 1С.

В настоящее время иногда выделяются также сверхвысокоуровневые языки программирования, позволяющие описывать даже не алгоритм решения задачи, а саму задачу, в частности, Python, Ruby, AWK/Perl.

Иногда объектно-ориентированные, а также языки запросов и другие сверхвысокоуровневые языки выделяются в следующее — четвёртое поколение языков программирования. При этом естественные языки объединяются с языками экспертных систем и баз знаний в пятое поколение.

В данном пособии уровень языка прикладных программистов будет рассматриваться в основном на примере компилируемого языка высокого уровня C++.

## 1.4. История

...В самом деле, так как возникновение по направлению вверх не беспредельно, то необходимо, чтобы не было вечным то, из чего как из первого возникло что-то через его уничтожение.

*Аристотель. Метафизика*

Термин «компьютер» (вычислитель) в разное время обозначал различные понятия — и клерка-вычислителя, проводившего расчёты вручную или с использованием простых (счёты и подобные им устройства — абак, суанпан) или сложных (арифмометр, механический калькулятор) вспомогательных устройств; и сами эти устройства. При этом в настоящее время даже инженерные калькуляторы представляют собой миникомпьютеры, превосходящие многие ранние ЭВМ. Таким образом, многие аспекты архитектуры вычислительных систем необходимо рассматривать в контексте развития вычислительной техники и программного обеспечения.

Договоримся называть компьютером, или вычислительной машиной, устройство, которое может исполнять заданную изменяемую последовательность вычислительных операций — программу. Таким образом, компьютер содержит как минимум два уровня абстракции — цифровой логический и архитектуры команд.

Непрограммируемые вычислительные устройства назовём калькуляторами или арифмометрами. Калькулятор содержит цифровой логический уровень, но не включает уровня архитектуры команд.

Исторически можно выделить такой вид калькуляторов, как табуляторы, выполняющие единообразную обработку больших массивов данных, представленных на перфокартах. Табуляторы можно назвать также промежуточным звеном между калькуляторами и компьютерами.

#### **1.4.1. Развитие вычислительной техники**

Вычислительные машины, как правило, разделяют на четыре поколения в соответствии с используемой элементной базой.

1. Электронные лампы.
2. Транзисторы.
3. Интегральные схемы малой и средней плотности.
4. Интегральные схемы большой и сверхбольшой плотности.

Это деление достаточно условно. Вычислительные машины разных поколений достаточно долгое время существовали параллельно. Часто различные компоненты одного и того же компьютера строились на различных элементных базах. Кроме того, такое деление не отражает развития архитектуры.

В некоторых источниках вводится понятие пятого поколения, но нет единого его определения. Часть источников выделяет пятое поколение не по элементной базе, а по решаемым задачам. Одноимённый японский проект называет ЭВМ пятого поколения искусственный интеллект, направленный на обработку знаний. Также пятым поколением называют компактные персональные ЭВМ. Таненбаум

считает пятым поколением встраиваемые системы (микроконтроллеры, системы на одном кристалле) [45], что естественным образом продолжает устоявшееся деление.

Любое из приведённых определений пятого поколения ЭВМ подразумевает, что в настоящее время оно успешно сосуществует с четвёртым. Также многие источники относят все современные компьютеры к четвёртому поколению.

Часто также выделяют нулевое поколение — электромеханические вычислительные машины. По аналогии с этим чисто механические вычислительные машины можно назвать минус первым поколением.

Большинство поколений делится на три периода: вначале выпускается прототип, основанный на новой элементной базе (или несколько независимо разработанных прототипов). Такие проекты часто остаются неизвестными из-за секретности или стечения обстоятельств. Через какое-то время множество стран и/или корпораций одновременно выпускают более совершенные компьютеры (одиночные или серийные). Соответственно, для каждого поколения указывается три даты: выпуск первого устройства, начало массового использования и выход из употребления.

### **Минус первое поколение (античность—XVIII в.—конец XX в.) — зубчатые колёса и рейки**

К этому поколению можно отнести всего два полноценных компьютера — проект Бэббиджа конца XIX в., реализованный только в 1985-1991 гг., и Z1 Конрада Цузе (1938 г.).

При этом, если заменить в определении компьютера «последовательность вычислительных операций» на просто «последовательность операций», как это сделано в **некоторых источниках**, к минус первому поколению компьютеров также можно отнести цифровые мультимедийные механические устройства, серийно выпускавшиеся в XV—XIX вв. (шарманки и музыкальные шкатулки),



а также механические станки с ЧПУ — первый известный такой станок был разработан в 1802 г. Жозефом Жаккаром.

Цифровые механические калькуляторы существенно более разнообразны. Сохранились упоминания об античных и вавилонских вычислительных устройствах, предназначенных для моделирования астрономических событий. Были найдены остатки подобного устройства — механизма из Антикитеры, собранного, по разным оценкам, в 140-80 гг. до н. э.

В современной европейской истории цифровой логический уровень присутствует в проекте тринадцатиразрядной машины Леонардо да Винчи, впервые реализован в 1623 г. в счётной машине Вильгельма Шиккарда, позже — в суммирующей машине Паскаля (1642 г.).

Позже были разработаны механические калькуляторы, выполняющие как сложение и вычитание, так и умножение и даже деление в десятичной системе — арифмометры Лейбница (1672 г.), Тома де Кольмара (1820 г.), Однера (1890 г.) и другие. В СССР наиболее популярен был «Феликс» (усовершенствованный арифмометр Однера), производившийся до 1978 г. Компактные, надёжные и энергонезависимые арифмометры повсеместно использовались до 1970-х гг. (а счёты — ещё дольше).

Около 1840 г. Томасом Фаулером был разработан механический калькулятор, работавший в сбалансированной троичной системе.

Дальнейшим развитием стал калькулятор, вычисляющий значение многочлена в десятичной системе — малая разностная машина, успешно построенная Чарльзом Бэббиджем в 1822 г. На её основе Георг Шутц и Мартин Виберг создали другие разностные калькуляторы.

Наиболее известная сейчас работа Чарльза Бэббиджа — постоянно дорабатывавшийся им проект механической десятичной аналитической машины, включающий управляющий барабан (УУ), хранилище (регистровую память), мельницу (арифметическое устройство — АУ).

На вход машины в последнем варианте проекта подавались два потока перфокарт: операционные карты (команды) и карты переменных (данные), что в современной классификации соответствует гарвардской архитектуре [63].

В 1930-х гг. над своим проектом вычислительной машины независимо от Бэббиджа начал работать Конрад Цузе, в это время — студент Берлинского политехнического. Компьютер Цузе был двоичным, для ввода-вывода данные преобразовывались в десятичный вид.

Цузе сформулировал основные принципы построения вычислительных машин:

- двоичная система счисления;
- использование устройств, работающих по принципу «да/нет» (логические 1 и 0);
- полностью автоматизированный процесс работы вычислителя;
- программное управление процессом вычислений;
- поддержка арифметики с плавающей запятой;
- использование памяти большой ёмкости.

Цузе впервые ввёл понятие «да/нет-статуса», аналогичное современному биту, термин «машинное слово», объединил в вычислителе арифметические и логические операции.

Первая демонстрационная модель Z1 была механической с электроприводом. Вместо использованных Бэббиджем шестерёнок логические и арифметические операции были реализованы на скользящих металлических рейках. Z1 обрабатывал 22-битные числа с плавающей запятой, включал не только регистры, но и механическую память (очень малого объёма, так что программа загружалась непосредственно с бумажной перфоленты). Поддерживались команды сложения и вычитания, умножения и деления, ввода и вывода, загрузки и сохранения в память. В системе команд Z1 не было условных переходов, так как их затруд-

нительно выполнять на перфоленте. Цикл реализовывался склейкой перфоленты в кольцо [23, 65].

В отличие от проекта аналитической машины Бэббиджа, который так и остался проектом, Z1 был реализован в 1938 г. и является первым в истории компьютером.

Серийно выпускаться механические компьютеры общего назначения так и не стали; но выпуск специализированных устройств (мультимедийных проигрывателей, калькуляторов, цифровых сигнальных процессоров) был налажен достаточно широко.

К электромеханическим цифровым сигнальным процессорам (DSP) можно отнести, в частности, знаменитую Энигму (1923-1945 гг.) и более совершенную шифровальную машину Lorenz SZ. Так как механическую часть представляли не реле, а зубчатые роторы, их можно условно отнести к минус первому поколению.

### **Нулевое поколение (1890–1941–1960) — реле**

Устройства нулевого поколения построены на основе телеграфных реле, идея которых была предложена ещё в 1830 г. Реле состоит из металлического переключателя, который, в зависимости от положения, может либо замыкать, либо размыкать электрическую цепь, и электромагнита, управляющего положением переключателя.

Таким образом, реле — электромеханическая ячейка, которая, в отличие как от более ранней механической памяти, так и от более поздних разновидностей, может принимать два и только два состояния.

В начале XX в. на основе реле были разработаны первые автоматические телефонные станции. Для этого был разработан и запатентован релейный регистр.

Первым счётным устройством, основанным на электромеханических реле, был табулятор Германа Холлерита (1890 г.). Для его разработки была создана компания, которая позже будет переименована в IBM.

В Германии на основе реле в 1939 г. Цузе разработал компьютер Z2 с механической памятью, по архитектуре практически полностью повторяющий Z1. В 1941 г. — Z3 с полностью релейной памятью, к системе команд которого было добавлено вычисление квадратного корня [23, 65]. Перфоленты для программ Z2 и Z3 делались уже не из бумаги, а из более прочной киноплёнки.

В США независимо от Цузе, но на основе проекта Бэббиджа, разрабатывались две линейки релейных вычислительных устройств — в фирме Bell (Джордж Штибитц) и в Гарвардском университете совместно с IBM (Говард Айкен).

Первый релейный калькулятор линейки Bell, «вычислитель комплексных чисел» был построен в 1940 г., последним (Bell Model V в 1946 г.) был полноценный многоядерный компьютер, где распределение заданий по ядрам выполнял аппаратный супервизор, который можно назвать ранней операционной системой. Система команд Bell Model V включала условные переходы, реализующиеся как переключение между различными перфолентами.

Первым из линейки Гарвардского университета и IBM первоначально также был калькулятор Harvard Mark I, или ASCC (1941 г.) с релейным процессором и механической памятью, который в 1944 г. был доработан до компьютера, загружающего инструкции с бумажной перфоленты.

Позже были разработаны полностью релейный Harvard Mark II (1947 г.) и релейно-ламповый Mark III/ADEC (1949 г.). Именно в электромеханическом реле Harvard Mark II, согласно легенде, был обнаружен первый баг.

В системе команд Mark I и Mark II, так же как и в ранних компьютерах Цузе, не было условных переходов, а циклы выполнялись закольцовыванием перфоленты. Программы Harvard Mark I и Mark II (аналогично позднему проекту Бэббиджа) хранились на перфолентах отдельно от данных, что позже получило название гарвардской архитектуры.

Из-за механического элемента в реле быстродействие таких машин было ограничено. Кроме того, у реле ограниченный ресурс срабатывания, поэтому ре-

лейные компьютеры были не слишком надёжны (хотя иногда надёжнее ламповых). Релейные компьютеры устойчивы к радиации и потребляют мало мощности.

### **Первое поколение (1943–1949–1965) — электронные лампы**

В 1918 г. М. А. Бонч-Бруевичем было изобретено электронное реле — триггер, состоящее из двух электронных ламп — триодов. Триггер может менять своё состояние быстрее электромеханического реле, что позволило ускорить быстроедействие вычислительных устройств.

Первым счётным устройством на электронных лампах считаются, согласно различным источникам, британский калькулятор Colossus Mark I и американский табулятор ENIAC. Оба этих калькулятора были построены в 1943 г.

Проект Colossus был рассекречен только в конце 1970-х гг., поэтому большая часть источников приписывает первенство ENIAC. **В рамках этого проекта был разработан также Colossus Mark II (1944 г.), который** не только работал в пять раз быстрее предшественника, но и был программируемым, так что Colossus Mark II может быть назван первым электронным компьютером (но не первым цифровым). В Colossus не было памяти, так что данные хранились на замкнутой перфоленте.

Табулятор ENIAC был разработан в школе Мура Пенсильванского Университета, США и обрабатывал десятичные числа. Каждый десятичный разряд представлялся десятью двоичными, при этом включён был только один из них — соответствующий нужной десятичной цифре. ENIAC иногда называют первой ЭВМ, но он не являлся вычислительной машиной (компьютером) в современном понимании, так как не имел уровня архитектуры команд. «Программирование» ENIAC выполнялось перекоммутацией связей, то есть фактически как перестройка машины.

В 1948-1950-х гг. началась повсеместная разработка ламповых компьютеров: EDVAC, BINAC и Harvard Mark III/ADEC в США, EDSAC в Британии, МЭСМ и М-1 в СССР, CSIRAC в Австралии.

Почти сразу, в 1951-1956 гг. был начат выпуск серийных компьютеров: UNIVAC, Ferranti, Минск-1, БЭСМ-1 и БЭСМ-2, серия «Стрела», Z22. В целом поколение электронно-ламповых компьютеров было многочисленным и разнообразным. В это время сложилось большинство архитектурных решений.

Кроме больших ЭВМ, занимавших иногда несколько этажей здания, стали появляться и малые. Так, первый мобильный компьютер общего назначения — Урал-1 (1955 г., СССР) — мог перевозиться на двух грузовиках.

Ненадёжность ламп приводила в том числе и к частым перестройкам и усовершенствованиям компьютеров, так что второе поколение включало множество моделей, большинство из которых существовали в единственном экземпляре. При этом программное обеспечение уже было достаточно сложным и функциональным, так что его было нерентабельно переписывать заново под язык каждой новой машины. Соответственно, в 1954–1960 гг. началось формирование двух новых уровней абстракции, обеспечивающих переносимость программ:

- декодирование машинной команды на набор микрокоманд, что позволяет нескольким различным компьютерам выполнять один и тот же набор команд (EDSAC-2);
- компиляция с языков высокого уровня (Фортран, Алгол, APL, Кобол).

### **Второе поколение (1955–1960–1970) — дискретные транзисторы**

В 1947 г. Уолтер Браттейн и Джон Бардин создали первый твердотельный аналог лампы-триода — полупроводниковый транзистор.

Транзисторы оказались компактнее, быстрее и надёжнее, чем триоды. Первым транзисторным компьютером считается экспериментальный TX-0 (1955 г., США, МТИ), на основе которого позже разработали TX-2, а затем PDP-1 (1961 г., США, DEC).

Первые частично транзисторные компьютеры общего назначения появились в 1958 г. Сразу же началось их серийное производство: Elliot-803 в Британии, Siemens-2002 в Германии, H-1 в Японии, Раздан-2, Минск-2 (а также его модифи-

кации Минск-22 и Минск 22М) и Минск-32, М-220 и М-222, Урал-14, Наири-1 и Наири-2, МИР, БЭСМ-4 и БЭСМ-6 в СССР, PDP-1, IBM 7030 (Stretch) и CDC 6600 в США и т. д.

Новая элементная база позволила уменьшить не только габариты больших ЭВМ (до нескольких десятков шкафов), но и мобильные компьютеры общего назначения (один-два шкафа). Такими были Раздан-2 и двоично-десятичный Проминь в СССР, PDP-4, PDP-5 и PDP-8 в США. Активно разрабатывались специализированные (обычно военные) мобильные компьютеры: бортовой компьютер самолёта TRADIC в США, Гранит, Клён, Диана, Радон и т. д. в СССР.

Также ко второму поколению относится агрегатная система средств вычислительной техники (набор устройств с унифицированными внешними связями, из которых можно компоновать различные вычислительные модели, начиная от простейших вычислительных машин сбора информации до сложных многопроцессорных систем обработки данных, систем массового обслуживания и т. д.) АСВТ-Д [49]. На основе АСВТ-Д собирались такие компьютеры, как М-1000, М-1010 (Ангара-2, 40 м<sup>2</sup>), М-2000 (144 м<sup>2</sup>) и М-3000 (170-220 м<sup>2</sup>). Для сравнения — площадь, занимаемая IBM 7030, составляет около 250 м<sup>2</sup>.

Появляется механизм трансляции адресов и страничная организация оперативной памяти.

Активно развиваются компьютеры на нестандартной элементной базе: Senac-1 на параметронах в Японии, троичная Сетунь в СССР, САВ-500 на магнитных элементах во Франции.

В процессе проектирования IBM 7030 (Stretch) возникло понятие байта как совокупности шести битов. В компьютерах других производителей байт мог быть равен семи или девяти битам.

Начиная со второго поколения, практически все компьютеры выпускаются сериями различного объёма. Кроме того, большинство описанных названий — не одна модель, а семейство компьютеров схожей архитектуры и, как правило,

с совместимыми наборами команд. Разные семейства (даже разрабатываемые на основе друг друга, как PDP-1 и PDP-4 или IBM 7030 и IBM 360) часто были несовместимы между собой. Тем не менее, в некоторых компьютерах, в частности, Минск-32 в СССР, поддерживались режимы программной совместимости с более ранними моделями [37].

### **Третье поколение (1961–1966–1980) — малые и средние интегральные схемы**

Первые интегральные схемы (отдельные триггеры в США и логические вентили в СССР) были созданы в 1961–1962 гг. параллельно Джеком Килби из Texas Instruments, Робертом Нойсом из Fairchild и Ю. В. Осокиным из КБ Рижского завода полупроводниковых приборов.

Первые компьютеры, построенные с использованием подобных схем, мало отличались от компьютеров, построенных только из отдельных транзисторов, то есть граница между вторым и третьим поколением достаточно условна. Некоторые модели компьютеров собирались вначале из дискретных элементов, затем из интегральных схем.

С увеличением степени интеграции появилась возможность выполнить на одном кристалле целый блок ЭВМ — регистр, дешифратор, счётчик и т. д. Примерно в это же время появляется понятие процессора, объединяющего в себе АЛУ (возможно, несколько специализированных АЛУ) и УУ.

К третьему поколению больших ЭВМ относятся, в частности, Днепр-2 и МИР-2, Урал-11М и Урал-25, Наири-3 и Наири-4 в СССР, серии IBM 360 и IBM 370 в США. Часто к третьему поколению по особенностям архитектуры относят и БЭСМ-6, хотя этот компьютер собирался из дискретных элементов.

Выросло количество мобильных компьютеров. Наиболее известны двенадцатиразрядная линейка компьютеров общего назначения PDP-8 и шестнадцатиразрядная — PDP-11 фирмы DEC. Для PDP-8 ввели термин «миникомпьютер»,



и, согласно легенде, это был первый компьютер, украденный частными лицами (он занимал всего один шкаф и весил менее 50 кг).

Также выпускались специализированные мобильные компьютеры. В СССР это были, в частности, Карат, Алмаз на основе модулярной арифметики и ряд программно-совместимых компьютеров Атака, Арка и Арфа, а также управляющий миникомпьютер Параметр. Параметр, в свою очередь, послужил основой для агрегатной системы средств вычислительной техники АСВТ-М, на основе которой собирались М-4000, М-6000.

Как было сказано ранее, в период создания машин из отдельных транзисторов или малых интегральных схем каждая компания устанавливала свои стандарты на аппаратные интерфейсы. К концу 1960-х гг. не было практически никаких общих стандартов. Проблема переносимости программного обеспечения, актуальная ещё в начале 1960 гг., встала крайне остро.

В капиталистических странах большая часть избыточных стандартов исчезла вместе с создавшими их фирмами, так как компьютеры без программного обеспечения были непопулярны. Остались несколько несовместимых между собой серий компьютеров разных фирм, в основном IBM и DEC.

В СССР был поставлен вопрос о создании единственного ряда компьютеров, совместимых на уровне команд друг с другом и основанных на системе команд наиболее известной западной линейки — IBM 360. Для IBM 360 к тому времени было написано много программ, но сами эти компьютеры, разработанные в начале 1960-х, устарели. Официальное сотрудничество с IBM было невозможно как из-за холодной войны, так и из-за политики самой IBM. Изначально выдвигался проект переориентации на сходные с IBM 360 архитектуры английской фирмы ICL или немецкой Siemens. Обе они были готовы официально поделиться существующими технологиями и немедленно начать совместную разработку компьютеров четвёртого поколения.

Тем не менее, в 1970 г. появился административный приказ о копировании устаревшей системы IBM 360. При этом предполагалось копировать не архитектуру команд, а микроархитектуру по нелегально полученным снимкам интегральных схем и фрагментам документации [42, 75].

Чуть позже в 1970 г. IBM анонсировала линейку IBM 370. А в 1971 г. в СССР был выпущен морально устаревший к этому времени клон IBM 360 — первый представитель линейки ЕС ЭВМ. Большая часть оригинальных разработок была прекращена и забыта; многие архитектурные решения, воплощённые в МИР и БЭСМ, были переоткрыты в Intel и AMD в 1990-2000 гг.

В описываемое время окончательно сформировались уровни архитектуры команд и языка высокого уровня.

Появляется ещё один уровень абстракции — промежуточные универсальные языки, облегчающие компиляцию (такие, как Алмо, Эпсилон, внутренний язык системы Бета в СССР). При компиляции с  $m$  языков высокого уровня для  $n$  машин через промежуточный язык достаточно разработать всего  $m + n$  трансляторов, при компиляции напрямую в машинные или ассемблерные коды  $m \cdot n$ . Этот уровень не прижился в 1970-е гг., но в настоящее время он активно используется для языков, подобных Java или C#, как уровень байт-кода.

#### **Четвёртое поколение (1971–1980–настоящее время) — большие и сверхбольшие интегральные схемы**

Граница между третьим и четвёртым поколением ещё более условна, чем между вторым и третьим. Часто считается, что переход к четвёртому поколению — размещение процессора на одной микросхеме (микропроцессоре).

Первой подобной микросхемой был процессор Intel 4004 (1971 г.).

Естественно, что первые микропроцессоры были относительно простыми, а сложные и высокопроизводительные процессоры четвёртого поколения по-прежнему были модульными.

В это время выпускались как большие ЭВМ, в основном уменьшившиеся до одного шкафа, наиболее производительные из которых стали называть суперкомпьютерами (Cray-1 в США, серия Эльбрус на основе БЭСМ-6 в СССР), так и миникомпьютеры на основе микропроцессоров, габариты и стоимость которых также уменьшались со временем. Это DEC VAX в США, различные линии СМ ЭВМ в СССР (среди них были и продолжения АСВТ-М, и клоны PDP-11, и оригинальные разработки, совместимые на уровне архитектуры команд с VAX или, позже, с Intel x86). Из специализированных мобильных компьютеров 1980-1990 гг. в СССР можно отметить Карат-КМ-Е, Акация, Лада-2.

В 1973 г. появился прототип Xerox Alto — недорогой миникомпьютер с экраном, клавиатурой, мышью и сетевой картой Ethernet, операционная система которого, как и Unix, поддерживала графический интерфейс пользователя. Подобный миникомпьютер позже получил название персонального компьютера.

В 1975 г. поступил в продажу персональный компьютер MITS Altair 8800 на основе процессора Intel 8080 и специально разработанной системной шины S-100. В базовой комплектации он не имел ни экрана, ни алфавитной клавиатуры, но поддерживал карты расширения. Благодаря низкой цене, гибкой и открыто опубликованной архитектуре, а также качественной рекламе Альтаир стал популярен, а энтузиасты разработали для него как полноценные периферийные устройства, так и качественное программное обеспечение.

Позже, начиная с 1977 г. были выпущены несколько недорогих персональных компьютеров: Apple II, IBM 5100, Tandy TRS-80, Commodore PET, Электроника НЦ-8010 и т. д. При разработке базового программного обеспечения для них отказались от графического интерфейса.

В 1981 г. появился персональный компьютер IBM 5150, или IBM PC. В отличие от более ранних моделей IBM, в IBM PC использовались сторонние компоненты, в частности, процессор Intel 8088. Архитектура IBM PC, вопреки обычной

политике IBM, была открытой, что привело к популярности и огромному количеству клонов (IBM PC-совместимых компьютеров).

В 1983 г. появился первый процессор с архитектурой ARM, предназначенной для встраиваемых систем.

Сейчас под словом «компьютер» чаще всего понимают «IBM PC-совместимый компьютер», хотя эта архитектура (и, соответственно, процессоры семейства x86) в настоящее время не является самой распространённой.

Наиболее популярными персональными компьютерами сейчас являются смартфоны и планшеты, где применяются процессоры семейства ARM. Ещё более распространены цифровые сигнальные процессоры и специализированные компьютеры. Часто специализированный компьютер реализуется в виде одной интегральной схемы — микроконтроллера.

Также в настоящее время продолжается выпуск суперкомпьютеров, таких как Cray-X1E. С ними успешно соперничают кластеры, объединяющие множество компьютеров общего назначения.

#### 1.4.2. Операционные системы

Наиболее известным семейством операционных систем является семейство Unix. Первоначально операционная система Unics (Uniplexed Information and Computing System) была разработана Кеном Томпсоном, Денисом Ритчи и Брайаном Керниганом как порт системы Multics (Multiplexed Information and Computing Service) на миникомпьютер DEC PDP-7. Первая версия Unics была написана на ассемблере [69].

Первая версия Unix (V1) появилась в 1971 г. Начиная с версии V6 (1975 г.) операционная система Unix распространилась в университетах, что привело к появлению множества различных веток.

В настоящее время потомки операционной системы Unix называются Unix-подобными операционными системами. Для архитектуры x86 наиболее распространены такие представители этого семейства, как GNU/Linux и разнообразные

ветки BSD; для ARM — основанная на ядре Linux система Android. Именно Android сейчас является наиболее распространённой операционной системой.

Понятие Unix-системы описывается семейством стандартов Single UNIX Specification (SUS). Зарегистрированными Unix-системами являются коммерческие операционные системы.

Кроме того, существует стандарт POSIX, описывающий взаимодействие операционной системы с прикладной программой, служащий для обеспечения совместимости Unix-подобных систем на уровне исходного кода. При этом операционная система может поддерживать POSIX и не являясь Unix-подобной.

Для процессоров семейства x86, кроме множества вариантов Unix-подобных систем, доступны также специфические операционные системы: линейка операционных систем с закрытым исходным кодом Microsoft Windows NT (в настоящее время NT является единственной поддерживаемой линейкой Microsoft Windows, поэтому обозначение NT часто опускают), семейство MenuetOS и так далее. Эти операционные системы не поддерживают стандарт POSIX, хотя для Microsoft Windows периодически заявляется частичная поддержка (часть POSIX, причём не для всех версий и не для всех типов Microsoft Windows).

## Контрольные вопросы

1. Что включает архитектура системы?
2. Что такое вычислительная система?
3. Какие вы знаете единицы измерения информации?
4. Какие вы знаете типы наборов команд?
5. Какие вы знаете виды архитектуры, различающиеся расположением программ и данных?
6. Какие уровни абстракции включает современная вычислительная система?

## Глава 2

# Представление данных

Проволока мира — число.

*В. Хлебников. Зангези*

Пифагорейцы в VI–IV веках до н. э. считали числа первоосновой мира. В настоящее время этот принцип воплощается на практике — всё больше информации переходит в цифровой формат, то есть описывается при помощи чисел. Числа же в свою очередь представляются в вычислительных системах при помощи специальных кодов в виде набора нулей и единиц.

### 2.1. Качественные и количественные данные

Всё познаваемое имеет число, потому что без числа  
невозможно что-либо понять или распознать.

*Филолай. Антология Стобея, I, 21*

Данные делятся на две основные группы — количественные и качественные. К **количественным** данным относятся в основном либо сведения о числе объектов, удовлетворяющих тем или иным условиям, либо числовые результаты измерений. Для таких данных имеют смысл базовые арифметические действия — сложение, вычитание, умножение на число или другую величину подходящей размерности. Именно для описания количественных данных изначально и появилось само понятие числа.

Для подсчёта количества объектов (яблок, землекопов и т. д.) используются **натуральные числа**: 1, 2, 3, ... Множество натуральных чисел  $\{1, 2, 3, \dots\}$  обычно обозначается как  $\mathbb{N}$ .

Ноль не используется при счёте, поэтому не считается натуральным числом. Тем не менее, часто удобно рассматривать множество  $\mathbb{N} \cup \{0\} = \{0, 1, 2, 3, \dots\}$ . Оно обозначается  $\mathbb{N}_0$ . В некоторых зарубежных источниках именно  $\mathbb{N}_0$  называется множеством натуральных чисел, но в отечественной литературе, как и в значительной части зарубежной, принято определение из предыдущего абзаца, не включающее ноль. Универсальное название элементов  $\mathbb{N}_0$  — неотрицательные целые числа, также их часто называют **беззнаковыми** целыми числами.

Для описания некоторых величин требуются отрицательные числа ( $-1, -2, -3, \dots$ ). Множество, включающее натуральные числа, противоположные им отрицательные и ноль, называется множеством целых чисел  $\mathbb{Z}$ . В противоположность беззнаковым (неотрицательным) величинам те, которые могут принимать как положительные, так и отрицательные значения, называются **знаковыми**.

Результаты измерений большей части величин (длина, путь, масса, время и т. д.) невозможно представить в виде целых чисел, знаковых или беззнаковых. Соответствующие им **вещественные** числа (в частности,  $\frac{1}{3}, \sqrt{2}, \pi, e$ ) образуют множество  $\mathbb{R}$ .

Числа и, соответственно, количественные данные лучше всего подходят для обработки с помощью вычислительных систем. Таким образом, чтобы ввести и использовать какую-то информацию в вычислительной системе, эту информацию необходимо представить в виде числа или набора чисел.

**Качественные** данные представляют собой, как правило, элементы (символы), несравнимые между собой или цепочки (строки) подобных символов. В частности, к качественным данным относятся цвет, буквы, цифры, ноты, символы шахматных фигур, карточных мастей и т. д. Множество всех возможных символов называется алфавитом.

Для обработки с помощью вычислительных систем качественные данные также необходимо представить в виде чисел. Чтобы сделать это, символы качественных данных упорядочиваются, затем каждому символу сопоставляется его

порядковый номер в списке — код символа, беззнаковое целое число. Полученное соответствие называется **кодовой таблицей**.

Наиболее известны кодовые таблицы, сопоставляющие коды знакам письменных языков — цифрам, буквам кириллицы, латиницы, знакам препинания и т. д. (чаще всего под термином «символ» подразумевается именно знак письменности). В частности, старейшая из принятых в настоящее время кодовая таблица ASCII (American Standard Code for Interchange of Information) описывает цифры, строчные и заглавные буквы латиницы, некоторое количество знаков препинания и специальных символов (так называемые ASCII-символы) и сопоставляет им коды в диапазоне от 0 до 127 [57]. Все более современные кодовые таблицы, включающие кириллицу и другие национальные знаки, описывает ASCII-символы этими же кодами.

Кодовая таблица ASCII настолько прочно закрепилась в информационных технологиях, что такие языки, как C/C++, вообще не делают различия между ASCII-символом и его кодом: тип *char* является одновременно и символьным, и коротким целым. При этом представление символов не из таблицы ASCII, в частности, кириллицы, может быть различным в разных реализациях.

Сейчас для представления различных национальных алфавитов чаще всего используется кодовая таблица Unicode, сопоставляющая кириллице коды от 1024 до 1279 (обычно коды Unicode записывают в шестнадцатеричном виде — от 0400 до 04FF). Для того, чтобы представление буквы кириллицы в памяти компьютера не могло совпасть с представлением последовательности из нескольких ASCII-символов, используются различные **кодировки** Unicode. Наиболее распространённая из них — UTF-8 — записывает символ в виде цепочки байтов, включающих, кроме собственно кода, ещё и служебную информацию. Соответственно, кириллица, представленная в UTF-8, занимает диапазон от D080 до D19F, так что кириллические буквы занимают два байта.



## 2.2. История чисел

Всякое начало трудно, — эта истина справедлива  
для каждой науки.

*К. Маркс. Капитал*

**Система счисления** — это представление чисел с помощью специальных письменных знаков — цифр.

Старейшая система представления чисел — единичная, когда при счёте каждому объекту сопоставляют один счётный предмет или символ — загибают палец, передвигают бусину чётка, ставят галочку на бумаге или зарубку на доске. Таким образом, чтобы записать число  $N$ , нужно поставить  $N$  галочек или зарубок. Значение счётного предмета или знака при этом не зависит от его положения и всегда равно единице. Этот громоздкий способ записи подходит только для небольших натуральных чисел.

Для подсчёта большого количества объектов естественно разбивать их на группы одного размера и подсчитывать уже эти группы. Размер такой группы называется **основанием** системы счисления.

Даже в единичной системе обычно используется группировка по какому-либо основанию. В чётках через определённое количество бусин вставляется отличная размером или цветом; пальцы рук естественным путём сгруппированы по 5 и 10, фаланги — по 12 (на всех пальцах одной руки, исключая большой), 15 (включая большой), 24 (на двух руках, исключая большие пальцы) и 30.

Исторически чаще всего использовались основания, равные 10 (по числу пальцев на руках) или 12 (по числу фаланг пальцев одной руки, кроме большого — он используется для указания на текущую фалангу-цифру). Системы счисления с такими основаниями называются соответственно десятичной и двенадцатеричной. Некоторыми народами использовались также восьмеричная си-

стема счисления (по суставам пальцев одной руки, кроме большого) и даже девятеричная. Сейчас общепринятой в быту является десятичная система счисления.

Группировка счётных знаков позволяет нагляднее представить относительно большие числа, но единичная запись всё равно будет громоздкой. Для сокращения длины логично записывать число не объектов, а групп и при этом как-то различать записи « $N$  объектов» и « $N$  групп объектов». Простейшее решение — ввести для группы, например, десятка, специальный счётный знак.

Подобная запись использовалась, в частности, в Древнем Египте. Там была принята десятичная система, при этом использовались различные знаки для записи единиц ( $|$ ), десятков ( $\cap$ ), сотен и так далее. Например, для записи числа  $29 = 2 \cdot 10 + 9 \cdot 1$  были необходимы два знака десятков и девять знаков единицы:

$\cap \cap ||||$  . Порядок записи этих знаков не был чётко определён.

Таким образом, значение счётного знака зависело только от его формы, но не от его положения относительно других знаков. Такие системы счисления называются непозиционными.

В Древнем Риме применялась смешанная система счисления — пятично-десятичная. Специальные знаки существовали для единиц (I, схематичное изображение пальца), пятёрок (V, напоминающая руку с отставленным большим пальцем), десятков (X, две руки), 50, 100 и т. д.

Значение цифры зависело от того, находилась ли она справа от более крупной цифры (в этом случае их значения складывались) или слева (в этом случае меньшее значение вычиталось из большего). В частности,  $29 = 2 \cdot 10 + 9 = 2 \cdot 10 + 10 - 1 = XXIX$ ,  $31 = 3 \cdot 10 + 1 = XXXI$ . Таким образом, значение счётного знака зависит и от его формы, и от его положения относительно других знаков, но при этом, например, единица и десяток обозначаются при помощи принципиально разных символов. Такая система счисления называется смешанной.

Римская запись сложна, почти так же громоздка, как египетская и, кроме, того, неоднозначна — например, число 4 записывалось и как IIII ( $4 \cdot 1$ ), и как IV ( $5 - 1$ ). Существуют различные виды римской записи, позволяющие либо сократить длину числа, либо убрать неоднозначность, но не то и другое сразу.

В Индии была придумана более удобная запись, в которой использовались различные счётные знаки для всех чисел от единицы до девяти, а также впервые возникло специальное обозначение для нуля. Для подобной записи числа, как и в египетской системе, разбивается на сумму некоторого числа единиц, десятков и т. д., причём для записи количества единиц и десятков будет использован один и тот же набор цифр. Значение цифры определяется её положением в записи; такая система счисления называется **позиционной**. Десятичная позиционная запись была перенята у индийцев арабами, а затем распространилась повсеместно. Начертание счётных знаков со временем превратилось в так называемые арабские цифры, которые сейчас привычны нам с детства, как и позиционная система записи:  $2 \cdot 10 + 9 \cdot 1 = 29$ .

В настоящее в быту повсеместно используется именно десятичная позиционная запись. На десятичной позиционной системе основано наиболее совершенное из простых счётных устройств — русские счёты. Ряд спиц в них соответствует позициям, десять костяшек на каждой спице — цифрам.

Одна из первых механическая суммирующая машина — Паскалина — также была построена на основе десятичной позиционной записи, несмотря на то, что использовалась для финансовых расчётов в принятой тогда недесятичной денежной системе. Каждому десятичному разряду соответствовало одно зубчатое колесо, на которое были нанесены цифры от 0 до 9. Поворачиваясь от 9 к 0, колесо сдвигало следующее за ним, формируя перенос.

Десятичная позиционная система счисления использовалась в механических калькуляторах, арифмометрах, разностной машине Бэббиджа, весившей более

трёх тонн, а также в его же проекте аналитической машины, не реализованном из-за своих габаритов и высокой стоимости. Даже много позже в самом известном из первых электронных (то есть построенных на вакуумных лампах) калькуляторов — ЭНИАК — использовалась десятичная система, из-за чего размеры ЭНИАК превысили размеры более поздних ламповых вычислительных машин.

При этом, как уже было сказано, основание системы счисления, равное десяти, было популярным, но не единственным. В XVI–XVII вв. различные учёные рассматривали двоичную, троичную и т. д. системы счисления и их свойства [11, 21]. Лейбниц видел в двоичной системе мистическое отражение реальности, но не рекомендовал использовать её на практике для вычислений (знаменитый арифмометр Лейбница был основан на десятичной системе).

Первая реально построенная вычислительная машина Z1 была механической и была создана в Германии Конрадом Цузе. Идеи Цузе были поразительно схожи с планами Бэббиджа (несмотря на независимую работу) и теми, что позже будут реализованы фон Нейманом. При этом Z1 помещалась в гостиной Цузе, а её масса составила всего 500 кг. Такая компактность была достигнута за счёт использования другой позиционной системы счисления — **двоичной**. Для ввода-вывода в Z1 было реализовано двоично-десятичное преобразование. Усовершенствованный вариант вычислителя Цузе, электромеханический Z2 на основе телеграфных реле, закрепил использование двоичной логики — реле может находиться только в двух состояниях — оно либо открыто, либо закрыто. Используемые в последующих вычислительных машинах вакуумные лампы, а затем транзисторы также могли находиться в двух состояниях. Таким образом, в настоящее время двоичная позиционная система применяется в электронике повсеместно.

Но двоичная система не является самой экономичной. Теоретически оптимальной является систем счисления по основанию  $e$ , а из натуральных оснований

наиболее экономичным является 3 [47]. Троичная ЭВМ — Сетунь была разработана в МГУ, выпускалась серийно и оказалась весьма эффективной. Но, так как для построения троичных логических элементов использовалась двоичная элементная база (транзисторы), существенно уменьшить габариты и стоимость за счёт экономичности основания системы счисления не удалось, так что в дальнейшем от троичной логики надолго отказались. Тем не менее, недавно о ней снова пришлось вспомнить: элементы квантовых компьютеров могут принимать как раз три значения.

Кроме позиционных систем счисления, в вычислительной технике использовались и другие, более оригинальные системы представления чисел. Наиболее интересной представляется система остаточных классов, или модулярная арифметика — представление числа  $x$  в виде последовательности остатков от деления  $x$  на набор взаимно простых чисел  $p_1, p_2, \dots, p_n$ .

$$x = (x \bmod p_1, x \bmod p_2, \dots, x \bmod p_n)$$

Такое представление позволяет выполнять умножение с той же скоростью, что и сложение.

Первым модулярным компьютером была чехословацкая ЭВМ Эпос, позже в СССР разрабатывалась серия модулярных ЭВМ. Наиболее известная советская модулярная ЭВМ, К-340А, используется до сих пор из-за своей поразительной надёжности и быстродействия [38, 39]. В настоящее время на модулярной арифметике основаны многие специализированные процессоры, предназначенные для обработки сигналов в режиме реального времени.

Программная реализация модулярной арифметики в настоящее время используется для ускорения вычислений в криптографии.

## 2.3. Позиционные системы счисления

Человек, который дружит с тиранами, подобен камешку  
при вычислении, значение которого бывает иногда  
большое, а иногда малое.

*Приписывается Солону.*

*Диоген Лаэртский. О жизни, учениях  
и изречениях знаменитых философов*

Все позиционные системы счисления строятся по одному общему принципу. Выбирается некоторое натуральное число  $N > 1$  — основание системы счисления, и каждое число  $X \in \mathbb{N}_0$  представляется в виде комбинации его степеней с коэффициентами, принимающими значения от 0 до  $N - 1$ , т. е. в виде

$$X = x_0 + x_1 \cdot N + \dots + x_k \cdot N^k \quad (2.1)$$

где  $0 \leq x_i < N$  — целые. Такое разложение существует и единственно для каждого  $X \in \mathbb{N}_0$ .

Далее число  $X$  сокращённо записывается в виде  $x_k \dots x_1 x_0$ . Для того, чтобы отличать это представление от сокращённой записи умножения  $x_k \cdot \dots \cdot x_1 \cdot x_0$ , над позиционной записью может быть проведена горизонтальная черта; основание системы счисления  $N$  может быть указано в виде нижнего индекса, таким образом:

$$X = x_k \cdot N^k + \dots + x_1 \cdot N + x_0 = \overline{x_k \dots x_1 x_0}_N \quad (2.2)$$

Если позиционность представления и основание системы очевидны из контекста, эти обозначения могут опускаться. Так, запись 13 для позиционного десятичного представления числа «тринадцать» будет, скорее всего, прочитана правильно, как и 0400 — 04FF для позиционного шестнадцатеричного представления диапазона кодов Unicode.

Каждому возможному значению  $x_i$ , от 0 до  $N - 1$ , соответствует специальный знак — цифра. Для записи значений, не превышающих девяти, используются привычные нам знаки  $0 \dots 9$ . Если необходимо записать одной цифрой значение «десять», для этого обычно используется знак А, одиннадцать записывается как В и так далее.

Таким образом, в часто используемой в информационных технологиях шестнадцатеричной системе счисления числа записываются с помощью шестнадцати цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, А, В, С, D, Е, F. Знаки А . . . F в этом контексте называются именно цифрами, а не буквами, и качественно ничем не отличаются от знаков  $0 \dots 9$ . В двенадцатеричной системе используется двенадцать цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, А, В.

$$13_{10} = 11_{12} = D_{16}$$

Величина, обозначаемая цифрой в записи числа, зависит от её позиции (разряда). Разряды числа, в отличие от текста, записываются по-арабски — справа налево. Цифра, записываемая крайней справа, обозначает количество единиц (младший разряд), вторая справа в десятичной системе обозначает количество десятков, в двенадцатеричной — количество дюжин и т. д. Крайний слева разряд называется старшим.

$$12345_{10} = 5 \cdot 10^0 + 4 \cdot 10^1 + 3 \cdot 10^2 + 2 \cdot 10^3 + 1 \cdot 10^4$$

Иногда разряды числа нумеруются, но порядок нумерации в различных источниках не совпадает. Младший разряд обозначается иногда как первый (так сложилось исторически), иногда как нулевой (так как  $1 = N^0$ ). В некоторых источниках разряды чисел фиксированной ширины (например, регистров) вообще нумеруются от старшего к младшему, по направлению текста.

В любой позиционной системе счисления ноль записывается как 0, единица — как 1. Основание этой системы счисления всегда записывается в виде 10.

### 2.3.1. Перевод натуральных чисел между позиционными системами счисления

В общем случае для перевода натуральных чисел между системами счисления используется деление с остатком. Действительно, сгруппируем в (2.1) все члены, куда входит  $N$  в степени, большей 1:

$$X = x_0 + (x_1 \cdot N + \dots + x_k \cdot N^k) \quad (2.3)$$

вынесем за скобки  $N$  и обозначим оставшееся значение как  $X_1$ :

$$X = x_0 + \underbrace{(x_1 + \dots + x_k \cdot N^{k-1})}_{X_1} \cdot N = x_0 + X_1 \cdot N, \quad (2.4)$$

где  $x_0$  и  $X_1$  — целые неотрицательные числа, причём  $0 \leq x_0 < N$ . Таким образом, младшая цифра  $x_0$  числа  $X$  в  $N$ -ичной системе счисления — остаток от целочисленного деления  $X$  на  $N$ , также в процессе этого деления мы получим частное, равное  $X_1$ . Так как в соответствии с (2.4)

$$X_1 = x_1 + x_2 \cdot N + \dots + x_k \cdot N^{k-1}, \quad (2.5)$$

следующую цифру  $x_1$  можно найти как остаток от деления  $X_1$  на  $N$  и так далее:

$$\left. \begin{array}{rcl} 266 & = & 22 \cdot 12 + 2 \\ 22 & = & 1 \cdot 12 + 10 \\ 1 & = & 0 \cdot 12 + 1 \end{array} \right\} 266_{10} = 1A2_{12}. \quad (2.6)$$

Если необходимо перевести число из системы счисления по произвольному основанию  $N$  в ту систему счисления, которая используется в расчётах (при расчётах вручную это обычно десятичная система), это можно сделать, используя определение (2.2):

$$12345_{12} = 5 \cdot 12^0 + 4 \cdot 12^1 + 3 \cdot 12^2 + 2 \cdot 12^3 + 1 \cdot 12^4 = 24677_{10}$$

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 5_{10}$$

$$1A_{16} = 10 \cdot 16^0 + 1 \cdot 16^1 = 26_{10}$$



Компьютер оперирует с данными, представленными в двоичной системе, таким образом, при вводе десятичных данных необходимое преобразование также можно выполнить по (2.2):

$$13_{10} = 11 \cdot 1010^0 + 1 \cdot 1010^1 = 11 + 1010 = 1101$$

Впрочем, десятично-двоичное преобразование уже реализовано в библиотеке ввода-вывода любого языка высокого уровня и выполняется неявно для программиста и тем более для пользователя.

Если число необходимо перевести из системы счисления по основанию  $N$  в систему по основанию  $N^k$ ,  $k \in \mathbb{N}$  или наоборот, то также нет необходимости в сложных вычислениях. В этом случае существует взаимно однозначное соответствие между группой из  $k$  разрядов в системе по основанию  $N$  и одним разрядом в системе по основанию  $N^k$ , что будет подробнее рассмотрено ниже на примере двоичной, восьмеричной и шестнадцатеричной систем.

### 2.3.2. Экономичность системы счисления

С точки зрения математики, все позиционные системы счисления равнозначны. Но, как показывает практика, вычислительные системы, обрабатывающие данные в двоичной системе, более просто устроены и, соответственно, имеют меньшие габариты, чем десятичные (при сопоставимых возможностях). Это связано с таким свойством, как экономичность, или компактность системы счисления. Считается, что понятие экономичности введено фон Нейманом.

Рассмотрим сравнительную экономичность двоичной и десятичной систем на следующем примере [47]. В десятичной системе для представления  $10^3$  чисел от 0 до 999 используется три разряда, каждый из которых помещает одну из десяти различных цифр, то есть всего используется 30 цифр. В двоичной системе из тех же 30 цифр можно составить 15 разрядов, каждый из которых хранит ноль или единицу. Таким образом, всего можно представить  $2^{15} = 32\,768$  различных чисел — более чем на порядок больше, чем в десятичной.

Обобщая эти рассуждения, можно оценить количество  $E_n(N)$  различных чисел, представимых в системе счисления по основанию  $N$  с помощью  $n$  цифр:

$$E_n(N) = N^{\frac{n}{N}} \quad (2.7)$$

Хотя на практике используются только натуральные основания систем счисления, (2.7) можно распространить и на вещественные положительные  $N$ . На рис. 2.1 представлена зависимость  $E_n(N)$  для трёх значений количества цифр  $n$ .

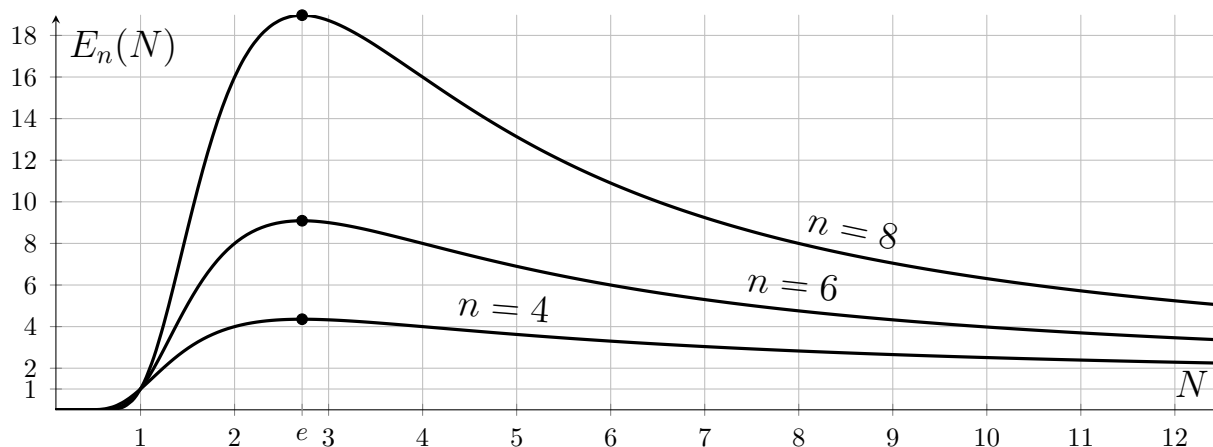


Рис. 2.1. Эффективность систем счисления

Как видно из рис. 2.1, количество используемых цифр  $n$  не влияет на то, какая из систем счисления более экономична, от него зависит только выраженность пика: чем большее количество цифр используется, тем больше разрыв между различными системами. Для всех значений  $n$  двоичная система существенно превосходит десятичную, но отнюдь не является самой эффективной. Максимум  $E_n(N)$  находится в точке  $N = e = 2,71828 \dots$ , что можно доказать дифференцированием (2.7). Из натуральных  $N$  наибольшее значение достигается при  $N = 3$ , то есть троичная система ещё экономичнее двоичной. Далее эффективность снижается с ростом  $N$ : четверичная система так же экономична, как и двоичная; при  $N > 4$  экономичность плавно снижается и асимптотически стремится к единице.

Также можно заметить, что для всех  $n$  получаем  $E_n(1) = 1$ . Действительно (хотя единичная система не является в полной мере позиционной и, соответственно, не может однозначно рассматриваться как частный случай (2.7)): в единичной системе с помощью  $n$  цифр можно записать только одно число — оно равно  $n$ .

Таким образом, наиболее эффективной должна быть вычислительная машина, построенная с использованием троичной системы счисления. Двоичная система, использование которой связано с тем, что большинство существующих электронных компонентов может находиться только в двух состояниях, также относительно эффективна.

### 2.3.3. Нецифровые символы в представлении чисел

В позиционной системе счисления по основанию  $N$  с помощью цифр можно записать любое число из  $\mathbb{N}_0$ . Но, как было сказано выше, для представления некоторых данных недостаточно неотрицательных целых чисел. В этих случаях используются формы записи, включающие дополнительные символы, не являющиеся цифрами.

#### Знак

В частности, отрицательные числа маркируются нецифровым символом «—» (минус), после которого записывается абсолютная величина (модуль) числа в используемой системой счисления. Соответственно, перед значением положительного числа может быть поставлен символ  $+$  (плюс), но он часто опускается:

$$\begin{aligned} -1_{10} &= -1_2 = -1_{12} \\ +2_{10} &= +10_2 = +2_{12} = 2_{12} \\ -11_{10} &= -1011_2 = -A_{12} \end{aligned} \tag{2.8}$$

## Дробная черта

Простые дроби  $\frac{m}{n}$  представляются своими числителем и знаменателем, разделёнными горизонтальной чертой:

$$\begin{aligned} \left(\frac{3}{4}\right)_{10} &= \left(\frac{11}{100}\right)_2 = \left(\frac{3}{4}\right)_{12} \\ \left(\frac{12}{5}\right)_{10} &= \left(\frac{1100}{101}\right)_2 = \left(\frac{10}{5}\right)_{12} \\ \left(\frac{100}{49}\right)_{10} &= \left(\frac{1100100}{110001}\right)_2 = \left(\frac{84}{41}\right)_{12} \end{aligned} \quad (2.9)$$

Но не все вещественные числа можно представить в виде отношения двух натуральных; кроме того, работать с таким представлением не всегда удобно.

## Разделитель дробной части

Более универсальным является расширение позиционного представления на случай вещественных чисел. Для разделения целой и дробной частей **такого представления** также используется нецифровой символ. В российской традиции это запятая, в западной — точка:

$$\begin{aligned} 11,5_{10} &= 1011,1_2 = \text{B},6_{12} \\ 0,25_{10} &= 0,01_2 = 0,3_{12} \end{aligned} \quad (2.10)$$

Рассмотрим это представление подробнее.

### 2.3.4. Позиционное представление вещественных чисел

Хотя определение позиционного представления по основанию  $N$  (2.2) изначально давалось для натуральных чисел, его можно расширить, введя отрицательные степени  $\frac{1}{N}$ ,  $\frac{1}{N^2}$  и так далее:

$$\begin{aligned} X &= x_k \cdot N^k + \dots + x_1 \cdot N^1 + x_0 \cdot N^0 + x_{-1} \cdot N^{-1} + \dots + x_{-\ell} \cdot N^{-\ell} + \dots = \\ &= x_k \cdot N^k + \dots + x_1 \cdot N + x_0 + \frac{x_{-1}}{N} + \dots + \frac{x_{-\ell}}{N^\ell} + \dots = \\ &= \overline{x_k \dots x_1 x_0 x_{-1} \dots x_{-\ell} \dots}_N, \quad 0 \leq x_i < N \text{ — целые} \end{aligned} \quad (2.11)$$

Коэффициенты  $x_i$  (цифры) записываются в порядке убывания степени  $N$ ; между коэффициентами при  $N^0$  и при  $N^{-1}$  ставится запятая. В форме (2.11) ( $N$ -ичной дроби) можно представить любое неотрицательное вещественное число  $X$ , но не всегда это можно сделать с помощью конечного количества знаков (количество отрицательных степеней  $N$  с ненулевыми коэффициентами может быть бесконечным даже для конечных рациональных чисел).

Если для какого-либо  $s$  все коэффициенты  $x_{-\ell}$  для  $\ell > s$  нулевые, позиционная дробь называется конечной (или, что то же самое, числом с конечной дробной частью):

$$\begin{aligned} X &= x_k \cdot N^k + \dots + x_1 \cdot N^1 + x_0 \cdot N^0 + x_{-1} \cdot N^{-1} + \dots + x_{-s} \cdot N^{-s} = \\ &= x_k \cdot N^k + \dots + x_1 \cdot N + x_0 + \frac{x_{-1}}{N} + \dots + \frac{x_{-s}}{N^s} = \\ &= \overline{x_k \dots x_1 x_0, x_{-1} \dots x_{-s} N}, \quad 0 \leq x_i < N \text{ — целые} \end{aligned} \quad (2.12)$$

Число  $X$ , представимое в одной системе счисления как конечная дробь, в другой может оказаться бесконечным.

В случае  $N = 10$  получаем привычные **десятичные дроби**, а запятая (или, в западной традиции, точка), отделяющая целую часть от дробной, называется **десятичным разделителем**.

Сгруппируем члены (2.11):

$$\begin{aligned} X &= \underbrace{(x_k \cdot N^k + \dots + x_1 \cdot N + x_0)}_{[X]} + \underbrace{(x_{-1} \cdot N^{-1} + \dots + x_{-\ell} \cdot N^{-\ell} + \dots)}_{\{X\}} = \\ &= [X] + \{X\} \end{aligned} \quad (2.13)$$

получаем разделение числа  $X$  на целую и дробную часть:  $[X]$  — целое неотрицательное число (сумма целых неотрицательных слагаемых), и  $\{X\} \in [0, 1]$ :

$$\begin{aligned} 0 &= 0 \cdot N^{-1} + \dots + 0 \cdot N^{-\ell} + \dots \leq \{X\} \leq \\ &\leq (N-1) \cdot N^{-1} + \dots + (N-1) \cdot N^{-\ell} + \dots = 1 \end{aligned} \quad (2.14)$$

если рассматривать только конечные дроби, то  $\{X\} \in [0,1)$ .

Для нуля и неотрицательных нецелых чисел представление (2.11) единственно, натуральные могут быть представлены в двух формах — с нулевой дробной частью или с бесконечной дробной частью  $\sum_{\ell=1}^{\infty} \frac{N-1}{N^\ell} = 1$ , например

$$\begin{aligned} 2_{10} &= 2,000 \dots_{10} = 1,999 \dots_{10} = \\ &= 10_2 = 10,000 \dots_2 = 1,111 \dots_2 = \\ &= 2_{12} = 2,000 \dots_{12} = 1,\text{BBB} \dots_{12} \end{aligned} \quad (2.15)$$

канонической (а если есть какие-то ограничения на длину дробной части — единственной) формой записи натуральных чисел является запись с нулевой дробной частью.

Таким образом, для представления числа  $X \in [0, +\infty)$  в каноническом виде (2.11) необходимо разделить его на целую часть  $\lfloor X \rfloor \in \mathbb{N}_0$ , которая затем записывается в позиционной форме (2.2) уже описанным способом, и дробную часть  $\{X\} \in [0,1)$ , которую надо представить в виде:

$$\{X\} = x_{-1} \cdot N^{-1} + \dots + x_{-\ell} \cdot N^{-\ell} + \dots \quad (2.16)$$

после чего представления целой и дробной частей записываются рядом. Для их разделения используется соответствующий нецифровой символ — запятая или (в западной традиции) точка.

### Геометрическая интерпретация

Рассмотрим геометрическую интерпретацию позиционной записи дробной части в виде (2.16). Представим диапазон  $[0,1)$  как полуинтервал на числовой оси (рис. 2.2). Края этого полуинтервала, показанные двумя вертикальными линиями — 0 и 1 — одинаково записываются в любой позиционной системе счисления.

Каждому значению из диапазона  $[0,1)$  соответствует одна точка полуинтервала, причём её положение не зависит от используемой системы счисления. Значение  $\{X\}$ , которое требуется представить в форме (2.16), показано вертикальной линией внутри полуинтервала.

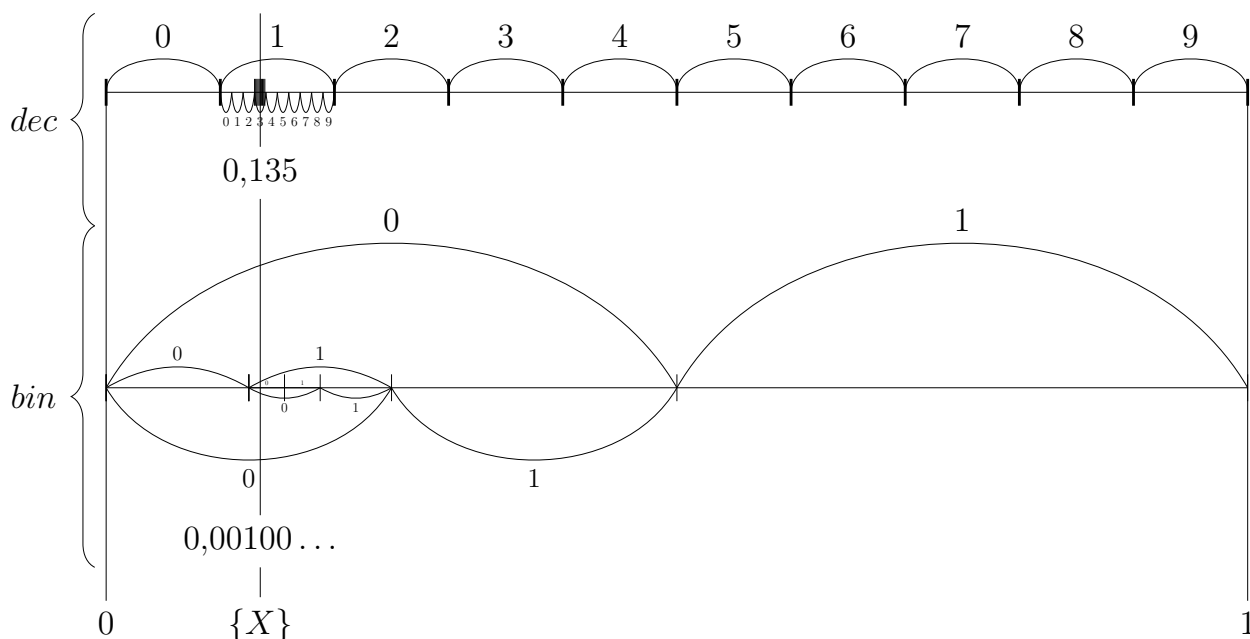


Рис. 2.2. Геометрическая интерпретация позиционного представления дробной части в различных системах счисления

Первое слагаемое (2.16), то есть  $\frac{x-1}{N}$ , можно проиллюстрировать разбиением полуинтервала на  $N$  равных полуинтервалов  $[\frac{n}{N}, \frac{n+1}{N})$ , где  $n$  принимает значения от 0 до  $N-1$ . На рис. 2.2 показано такое разбиение для  $N=10$  (вверху) и  $N=2$  (внизу). Значение  $n$  для каждого полуинтервала записано на дуге, проведённой над этим полуинтервалом. Соответственно, первая цифра в позиционной записи  $\{X\}$  соответствует тому полуинтервалу, которому принадлежит точка  $\{X\}$ . Для значения, показанного на рис. 2.2, это 1 в десятичной системе счисления и 0 — в двоичной.

Вторая цифра получается делением полученного полуинтервала на  $N$  ещё меньших (показаны дугами и цифрами под осью). Для показанного на рисунке значения получаем 3 в десятичной системе счисления и 0 — в двоичной.

Дальнейшее разбиение полуинтервалов (на рисунке показано только для двоичной системы) приводит к десятичной записи  $0,135$  и двоичной  $0,00100 \dots$

Вообще, дробь конечна, если простые делители знаменателя простой дроби входят в число делителей основания системы счисления. Таким образом, чтобы число представлялось конечной двенадцатеричной дробью, знаменатель должен быть произведением произвольного количества двоек и троек; чтобы дробь была конечной в десятичной системе — двоек и пятёрок; а в двоичной системе



конечными будут только те дроби, знаменатель которых является степенью двойки.

### Перевод вещественных чисел между позиционными системами счисления

Процесс перевода вещественного числа  $X \in \mathbb{R}$  в систему счисления по основанию  $N$  включает шесть стадий:

1. Знак числа отделяется от абсолютной величины:

$$X = \pm |X| \quad (2.19)$$

2. Абсолютная величина разделяется на целую и дробную части:

$$|X| = \lfloor |X| \rfloor + \{ |X| \}, \quad (2.20)$$

где  $\lfloor |X| \rfloor \in \mathbb{N}_0$ ,  $\{ |X| \} \in [0; 1)$ .

3. Целая часть абсолютной величины числа раскладывается на сумму неотрицательных степеней основания  $N$ :

$$\lfloor |X| \rfloor = a_0 + a_1 N + a_2 N^2 + \dots + a_k N^k, \quad a_i \in \{0, 1, \dots, N-1\}. \quad (2.21)$$

Для нахождения коэффициентов  $a_i$  используется деление с остатком, как говорилось ранее.

4. Дробная часть раскладывается на сумму отрицательных степеней  $N$  (возможно, бесконечную — на практике в этом случае дробная часть округляется в соответствии с погрешностью числа  $X$ ):

$$\{ |X| \} = \frac{b_1}{N} + \frac{b_2}{N^2} + \frac{b_3}{N^3} + \dots, \quad b_i \in \{0, 1, \dots, N-1\}. \quad (2.22)$$

5. Абсолютная величина числа записывается как последовательность коэффициентов  $a_i$  и  $b_i$ , расположенных по убыванию степени (коэффициенты при отрицательных степенях отделяются запятой):

$$|X| = \overline{a_k \dots a_2 a_1 a_0, b_1 b_2 b_3 \dots} \quad (2.23)$$

6. Перед записью (2.23) ставится знак «+» или «−», определённый на первой стадии (знак «+» может быть опущен).

Рассмотрим процесс перевода дробной части  $\{ |X| \} \in [0, 1)$  в вид (2.22) подробнее. Если цифры целой части получались путём последовательного деления её с остатком на основание системы, то очередную цифру дробной части можно определить *умножением* на  $N$ :

$$\begin{aligned} \{ |X| \} \cdot N &= \overline{0, b_1 b_2 b_3 \dots} \cdot N = \\ &= \left( \frac{b_1}{N} + \frac{b_2}{N^2} + \frac{b_3}{N^3} + \dots \right) \cdot N = b_1 + \frac{b_2}{N} + \frac{b_3}{N^2} + \dots = \\ &= \overline{b_1, b_2 b_3 \dots} = b_1 + \overline{0, b_2 b_3 \dots} \end{aligned} \quad (2.24)$$

Таким образом, после умножения  $\{ |X| \} \in [0, 1)$  на  $N$  получаем значение в диапазоне  $[0, N)$ . Его целая часть —  $b_1$ , первая цифра  $\{ |X| \}$  после запятой. Дробная часть  $\overline{0, b_2 b_3 \dots}$  лежит в диапазоне  $[0, 1)$ . Умножая её на  $N$ , можно найти вторую цифру после запятой  $b_2$  и так далее.

Таким образом, последовательность цифр  $b_i$  можно получить по следующей итерационной схеме:

$$\begin{cases} b_i = \lfloor X_{i-1} \cdot N \rfloor \\ X_i = \{ X_{i-1} \cdot N \} \end{cases}, \quad (2.25)$$

где  $X_0 = \{ |X| \}$ .

В таблице 2.1 показан процесс перевода значения  $X_0 = 0,135_{10}$  в двоичную систему счисления. Порядковый номер действия  $i$  соответствует позиции полученной цифры после запятой. Сама полученная цифра  $b_i$  (целая часть результата действия) показана жирным шрифтом.

На двадцать четвёртом шаге результат совпал с полученным на четвёртом шаге. Так как следующий шаг полностью определяется результатом предыдущего, далее процесс повторится, и получится бесконечная периодическая дробь:

$$0,135_{10} = 0,001(00010100011110101110)_2 \quad (2.26)$$

**Перевод  $0,135_{10}$  в двоичную систему счисления**

Таблица 2.1

$i$	Действие	$i$	Действие	$i$	Действие
1	$0,135 \cdot 2 = 0,270$	9	$0,560 \cdot 2 = 1,120$	17	$0,360 \cdot 2 = 0,720$
2	$0,270 \cdot 2 = 0,540$	10	$0,120 \cdot 2 = 0,240$	18	$0,720 \cdot 2 = 1,440$
3	$0,540 \cdot 2 = 1,080$	11	$0,240 \cdot 2 = 0,480$	19	$0,440 \cdot 2 = 0,880$
4	$0,080 \cdot 2 = 0,160$	12	$0,480 \cdot 2 = 0,960$	20	$0,880 \cdot 2 = 1,760$
5	$0,160 \cdot 2 = 0,320$	13	$0,960 \cdot 2 = 1,920$	21	$0,760 \cdot 2 = 1,520$
6	$0,320 \cdot 2 = 0,640$	14	$0,920 \cdot 2 = 1,840$	22	$0,520 \cdot 2 = 1,040$
7	$0,640 \cdot 2 = 1,280$	15	$0,840 \cdot 2 = 1,680$	23	$0,040 \cdot 2 = 0,080$
8	$0,280 \cdot 2 = 0,560$	16	$0,680 \cdot 2 = 1,360$	24	$0,080 \cdot 2 = 0,160$

Таким образом,  $X_0 = 0,135_{10}$  невозможно точно представить конечной двоичной дробью. Для практической обработки его необходимо округлить.

Как определить, сколько двоичных разрядов достаточно для представления числа? Если значение  $0,135_{10}$  точное, то есть равно  $0,1350000 \dots_{10}$ , то любое конечное количество разрядов будет недостаточным, и погрешность округления будет определяться возможностями вычислителя.

Если, как чаще всего бывает для измеряемых величин, достоверно известны только приведённые цифры, то есть  $X_0 = (0,135 \pm 0,0005)_{10}$ , можно отбросить все те разряды, вес которых  $\frac{1}{2^\ell}$  меньше погрешности. Это разряды, для которых  $2^\ell > \frac{1}{0,0005} = 2000$ , а именно  $\ell \geq 11$ . Так как одиннадцатый разряд нулевой, неоднозначности с округлением не возникает. Таким образом, получаем десять значащих двоичных разрядов:

$$(0,135 \pm 0,0005)_{10} = (0,0010001010 \pm 0,00000000001)_2 \quad (2.27)$$

Последний ноль в двоичной записи также является значащим, так как его вес больше погрешности значения.

Рассмотрим приближённое значение  $\widetilde{X}_0 = 0,0010001010_2$  и переведём его обратно в десятичную систему. Все действия выполняются аналогично приведённым ранее — на каждом шаге выполняется масштабирование в  $10_{10} = 1010_2$  раз (таблица 2.2). Полученная на каждом шаге цифра  $b_i$  выделена жирным шрифтом и для наглядности продублирована в десятичном виде в последнем столбце.

### Перевод $0,0010001010_2$ в десятичную систему счисления

Таблица 2.2

$i$	Действие	$(b_i)_{10}$
1	$0,0010001010 \cdot 1010 = \mathbf{1},01011001$	1
2	$0,01011001 \cdot 1010 = \mathbf{11},0111101$	3
3	$0,0111101 \cdot 1010 = \mathbf{100},110001$	4
4	$0,110001 \cdot 1010 = \mathbf{111},10101$	7
5	$0,10101 \cdot 1010 = \mathbf{110},1001$	6
6	$0,1001 \cdot 1010 = \mathbf{101},101$	5
7	$0,101 \cdot 1010 = \mathbf{110},01$	6
8	$0,01 \cdot 1010 = \mathbf{10},1$	2
9	$0,1 \cdot 1010 = \mathbf{101},0$	5

Технически можно довести вычисления до конца, так как любая конечная двоичная дробь представима конечной десятичной (так как два — делитель десяти). На девятом шаге получаем нулевую дробную часть, то есть  $\widetilde{X}_0 = 0,0010001010_2$  представляется конечной десятичной дробью:

$$\widetilde{X}_0 = 0,0010001010_2 = 0,134765625_{10} \quad (2.28)$$

Но большая часть знаков этой дроби — «мусорные».

Если в записи  $\widetilde{X}_0$  достоверны только указанные знаки, то есть  $\widetilde{X}_0 = 0,0010001010 \pm \frac{1}{2^{11}}$ , то десятичные разряды с весом  $\frac{1}{10^\ell}$ , меньшим погрешности, не определены. Это разряды с  $10^\ell > 2^{11} = 2048$ , то есть  $\ell \geq 4$ . Таким образом, в ответе останутся три значащих десятичных цифры после запятой, а вычисления в таблице 2.2 нужно было прервать после четвёртого шага ( $\widetilde{X}_0 = 0,1347\dots$ ) и округлить до трёх десятичных разрядов. После этого получим исходное значение  $X_0 = 0,135_{10}$ .

## 2.4. Двоичное представление беззнаковых целых чисел

Но да будет слово ваше: «да, да»; «нет, нет»; а что сверх этого, то от лукавого.

*Мф. 5:37*

В настоящее время в вычислительных системах повсеместно используются элементы, которые могут находиться в двух различных состояниях. Соответственно, применяется позиционная система счисления по основанию 2, в которой используется всего две цифры — 0 и 1 (без нецифровых символов, так что отрицательные и вещественные числа невозможно представить привычным образом; работа с ними будет рассмотрена отдельно).

Таким образом, число  $X \in \mathbb{N}_0$  представляется в виде:

$$X = x_0 + x_1 \cdot 2 + \dots + x_k \cdot 2^k = \overline{(x_k \dots x_1 x_0)}_2, \quad x_i \in \{0, 1\} \quad (2.29)$$

например,

$$109_{10} = 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 0 \cdot 16 + 1 \cdot 32 + 1 \cdot 64 = 1101101_2.$$

Каждому двоичному разряду (биту, от binary digit) числа, соответствует одна элементарная ячейка памяти, которая может находиться в одном из двух состояний. Эти состояния обычно также обозначаются как 0 и 1.

Количество таких ячеек в числе не может быть произвольным — оно определяется особенностями вычислительной системы и всегда кратно определённом числу, называемому размером байта. Число минимально возможного размера, соответственно, называется **байтом**. В настоящее время байт обычно содержит 8 разрядов (битов), иногда — 16, другие значения встречаются реже.

Ячейка памяти не может быть пуста — в ней обязательно содержится либо 0, либо 1. В частности, хотя двоичное представление десятичного числа 109 содержит семь значащих цифр (1101101), записать в восьмибитный байт можно только восемь (01101101).

Такая запись называется натуральным двоичным кодом (binary, сокращённо *bin*), так как она, с одной стороны, используется для натуральных чисел (и ещё для нуля), а с другой — является наиболее естественным их представлением.

Добавленный нулевой старший бит соответствует в разложении числа слагаемому  $0 \cdot 128 = 0$ , то есть не влияет на значение числа. Таким образом, ноль в старшем разряде (ведущий ноль) для числа в натуральном двоичном коде является **незначащим** и часто опускается на письме.

Минимальное число, которое можно записать в восьми битах натуральным двоичным кодом — ноль (0000 0000), максимальное — 255 (1111 1111). В общем случае  $N$  битами можно записать числа от 0 до  $2^N - 1$ .

#### 2.4.1. Восьмеричное и шестнадцатеричное представление

Двоичная запись даже относительно небольших чисел выглядит очень громоздко и трудно читается человеком. Обычно для ввода-вывода используется десятичная запись, но в некоторых случаях это неприемлемо.

Для того, чтобы придать числам компактный вид и при этом сохранить их двоичную структуру, используются восьмеричная (octal, *oct*) и шестнадцатеричная (hexadecimal, *hex*) системы счисления. Так как основания этих систем являются степенью двойки (то есть основания исходной системы), нет необходимости в сложных вычислениях.

## Восьмеричное представление натуральных чисел

Рассмотрим двоичную запись некоторого числа в форме (2.29), сгруппируем слагаемые по тройкам (если число разрядов не кратно трём, дополним число справа одним или двумя незначащими нулями) и вынесем общий множитель за скобки:

$$\begin{aligned}
 X &= (\overline{x_k \dots x_1 x_0})_2 = \\
 &= \underbrace{x_0 + x_1 \cdot 2 + x_2 \cdot 2^2}_{+ \dots +} + \underbrace{x_3 \cdot 2^3 + x_4 \cdot 2^4 + x_5 \cdot 2^5}_{+ \dots +} + \dots = \\
 &= (x_0 + x_1 \cdot 2 + x_2 \cdot 2^2) + (x_3 + x_4 \cdot 2 + x_5 \cdot 2^2) \cdot 2^3 + \\
 &\quad + \dots + (x_{3i} + x_{3i+1} \cdot 2 + x_{3i+2} \cdot 2^2) \cdot 2^{3i} + \dots
 \end{aligned} \tag{2.30}$$

Так как двоичные цифры могут принимать только значения 0 и 1, значение внутри каждой скобки целое, неотрицательное и не превышает семи:

$$0 = 0 + 0 \cdot 2 + 0 \cdot 2^2 \leq x_{3i} + x_{3i+1} \cdot 2 + x_{3i+2} \cdot 2^2 \leq 1 + 1 \cdot 2 + 1 \cdot 2^2 = 7 \tag{2.31}$$

коэффициенты при скобках имеют вид  $2^{3i} = (2^3)^i = 8^i$ . Таким образом, получаем позиционную восьмеричную запись:

$$\begin{aligned}
 X &= (x_0 + x_1 \cdot 2 + x_2 \cdot 4) + (x_3 + x_4 \cdot 2 + x_5 \cdot 4) \cdot 8 + \\
 &\quad + \dots + (x_{3i} + x_{3i+1} \cdot 2 + x_{3i+2} \cdot 4) \cdot 8^i + \dots = \\
 &= \tilde{x}_0 + \tilde{x}_1 \cdot 8 + \dots + \tilde{x}_i \cdot 8^i, \quad \tilde{x}_i \in \{0, 1, 2, \dots, 7\}
 \end{aligned} \tag{2.32}$$

где  $\tilde{x}_i = x_{3i} + x_{3i+1} \cdot 2 + x_{3i+2} \cdot 4$ .

На практике для перевода двоичной записи в восьмеричную достаточно разбить разряды на тройки и затем заменить каждую тройку двоичных цифр одной восьмеричной (таблица 2.3):

$$1101010_2 = 001\ 101\ 010_2 = 152_8 \tag{2.33}$$

для обратного преобразования каждая восьмеричная цифра заменяется тремя двоичными:

$$234_8 = 010\,011\,100_2 = 10011100_2 \quad (2.34)$$

### Соответствие двоичных триад восьмеричным цифрам

Таблица 2.3

<i>bin</i>	000	001	010	011	100	101	110	111
<i>oct</i>	0	1	2	3	4	5	6	7

Восьмеричное представление чисел используется, в частности, для записи прав доступа в Unix.

### Шестнадцатеричное представление натуральных чисел

Аналогично предыдущему разделу, для перевода из двоичной системы в шестнадцатеричную двоичные разряды необходимо группировать по четыре, так как  $16 = 2^4$ :

$$\begin{aligned}
 X &= \overline{(x_k \dots x_1 x_0)}_2 = \\
 &= \underbrace{x_0 + x_1 \cdot 2 + x_2 \cdot 2^2 + x_3 \cdot 2^3}_{+ \dots +} + \underbrace{x_4 \cdot 2^4 + x_5 \cdot 2^5 + x_6 \cdot 2^6 + x_7 \cdot 2^7}_{+ \dots +} + \\
 &\quad + \dots + \underbrace{x_{4i} \cdot 2^{4i} + x_{4i+1} \cdot 2^{4i+1} + x_{4i+2} \cdot 2^{4i+2} + x_{4i+3} \cdot 2^{4i+3}}_{+ \dots +} + \dots = \\
 &= (x_0 + x_1 \cdot 2 + x_2 \cdot 2^2 + x_3 \cdot 2^3) + (x_4 + x_5 \cdot 2 + x_6 \cdot 2^2 + x_7 \cdot 2^3) \cdot 2^4 + \\
 &\quad + \dots + (x_{4i} + x_{4i+1} \cdot 2 + x_{4i+2} \cdot 2^2 + x_{4i+3} \cdot 2^3) \cdot 2^{4i} + \dots = \\
 &= (x_0 + x_1 \cdot 2 + x_2 \cdot 4 + x_3 \cdot 8) + (x_4 + x_5 \cdot 2 + x_6 \cdot 4 + x_7 \cdot 8) \cdot 16 + \\
 &\quad + \dots + (x_{4i} + x_{4i+1} \cdot 2 + x_{4i+2} \cdot 4 + x_{4i+3} \cdot 8) \cdot 16^i + \dots = \\
 &= \tilde{x}_0 + \tilde{x}_1 \cdot 16 + \dots + \tilde{x}_i \cdot 16^i, \quad \tilde{x}_i \in \{0, 1, 2, \dots, F\}
 \end{aligned} \quad (2.35)$$

четыре двоичных разряда (тетрада) заменяются одним шестнадцатеричным (таблица 2.4)

$$1101010_2 = 0110\,1010_2 = 6A_{16} \quad (2.36)$$



и наоборот

$$2B3_{16} = 0010\ 1011\ 0011_2 = 001010110011_2 \quad (2.37)$$

### Соответствие двоичных тетрад шестнадцатеричным цифрам

Таблица 2.4

<i>bin</i>	0000	0001	0010	0011	0100	0101	0110	0111
<i>hex</i>	0	1	2	3	4	5	6	7
<i>bin</i>	1000	1001	1010	1011	1100	1101	1110	1111
<i>hex</i>	8	9	A	B	C	D	E	F

Шестнадцатеричное представление используется чаще, так как типичный байт (восемь бит) представляется двумя шестнадцатеричными цифрами. Часто двоичный код разделяют на тетрады просто для читабельности.

Так как восьмеричная и шестнадцатеричная системы счисления обычно используются для более компактной записи двоичного кода (содержимого памяти, в котором не используются нецифровые символы), а не как самостоятельные позиционные системы, они традиционно применяются только для беззнаковых целых чисел. Для более сложных структур данных под шестнадцатеричным представлением подразумевается шестнадцатеричное представление двоичного содержимого памяти, а не самих данных. Так, запись  $-1 = \text{FFFF FFFF}$  обозначает, что значение  $-1$  представляется в памяти двоичным кодом `1111 1111 1111 1111 1111 1111 1111 1111`, что в шестнадцатеричном виде соответствует `FFFF FFFF`.

В частности, функции форматированного ввода-вывода стандартной библиотеки C++ (семейства `scanf` и `printf`) предоставляют два спецификатора для десятичного представления — десятичное знаковое `%d` и десятичное беззнаковое `%u`, но шестнадцатеричное представление возможно только беззнаковое.

## 2.4.2. Беззнаковая арифметика в вычислительных системах

Из-за единообразного представления чисел в позиционных системах счисления арифметические действия во всех таких системах выполняется по одному и тому же алгоритму. Соответственно, сложение, вычитание, умножение и деление натуральных чисел, записанных в двоичной системе, выполняется по привычной нам схеме «в столбик», с одним отличием:  $1 + 1 = 10$ .

Арифметика в вычислительных системах имеет ещё одно отличие от ручных вычислений, кроме основания системы счисления — ограниченность. Производя вычисления вручную, можно потенциально обрабатывать сколь угодно большие числа. Узел ЭВМ, предназначенный для выполнения арифметических действий, имеет фиксированную разрядность. В частности,  $N$ -разрядный сумматор — узел, выполняющий сложение — обрабатывает операнды, состоящие из  $N$  разрядов и формирует результат также длины  $N$ ; и возможна ситуация, когда операнды арифметической операции попадают в допустимый диапазон, а её результат — уже нет.

Большинство современных процессоров семейства x86 имеет наборы команд для  $N = 8_{10}$ ,  $N = 16_{10}$ ,  $N = 32_{10}$  и  $N = 64_{10}$ . Для наглядности все арифметические операции будут рассматриваться на примере  $N = 8_{10}$ .

### Сложение и вычитание

Сложение производится поразрядно, начиная с младшего разряда. Если сумма младших разрядов равна или превышает 10, возникает **перенос в старший разряд**.

Рассмотрим сложение двух чисел, представленных в двоичном виде. При сложении младших разрядов получаем  $1 + 1 = 10$ , то есть младший разряд суммы равен 0, а к более старшему разряду добавляется единица переноса (перенос показан мелким шрифтом над первым слагаемым). Соответственно, для второго разряда получаем уже  $1 + 1 + 1 = 11$  — единица в соответствующем разряде

суммы и единица переноса — и так далее:

$$\begin{array}{r}
 \phantom{+} \overset{1}{1} \\
 + 1010011 \\
 \hline
 \phantom{+} 1011 \\
 \phantom{+} 1011110
 \end{array} \quad (83_{10} + 11_{10} = 94_{10}) \quad (2.38)$$

В скобках показано десятичное представление слагаемых и суммы.

Как уже было сказано, одной из особенностей арифметики вычислительных систем является ограниченный диапазон представимых чисел. Все операции в ЭВМ выполняются над числами фиксированной длины  $N$  (в данном разделе рассматривается случай  $N = 8_{10}$ ).

В этом случае пример (2.38) корректнее было бы записать в виде

$$\begin{array}{r}
 \phantom{+} \overset{1}{1} \\
 + 01010011 \\
 \hline
 \phantom{+} 00001011 \\
 \phantom{+} 01011110
 \end{array} \quad (83_{10} + 11_{10} = 94_{10}) \quad (2.39)$$

так как ни один из разрядов числа не может быть пуст. Но такая запись тяжело читается, поэтому незначащие нули часто опускают.

Пусть необходимо прибавить к максимальному представимому числу (для восьми разрядов это  $1111\,1111 = 255_{10}$ ) единицу. Сумма младших разрядов слагаемых даст ноль в младшем разряде и единицу переноса ( $1 + 1 = 10$ ); сумма единицы переноса и вторых разрядов слагаемых — ноль во втором разряде и единицу переноса ( $1 + 1 + 0 = 10$ ) и так далее. Сумма старших (восьмых) разрядов тоже также даст ноль в старшем (восьмом) разряде результата и единицу переноса в девятый разряд результата ( $1 + 1 + 0 = 10$ ), так что в неограниченной арифметике (в частности, при ручном расчёте) получилось бы

$$\begin{array}{r}
 \phantom{+} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\
 + 11111111 \\
 \hline
 \phantom{+} 1 \\
 \phantom{+} 100000000
 \end{array} \quad (255_{10} + 1_{10} = 256_{10}) \quad (2.40)$$

но у восьмибитного результата нет девятого разряда — поэтому фактический результат будет равен нулю. Такая ситуация называется **беззнаковым пере-**

**полнением.** Бит переноса в несуществующий разряд результата сохраняется в специальной ячейке, называемой **флагом переноса** ( $CF$  — Carry flag).

$$\begin{array}{r}
 \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\
 + 11111111 \\
 \hline
 \phantom{00000000} 1
 \end{array} \quad (255_{10} + 1_{10} = 0_{10}) \quad CF = 1 \quad (2.41)$$

В общем случае в  $N$ -битной арифметике  $(2^N - 1) + 1 = 0$ , при этом  $CF = 1$ .

Флаг переноса не является частью числа, куда записывается результат. Конструктивно ячейка  $CF$  принадлежит сумматору и, если подряд выполняется несколько операций, каждая из них будет перезаписывать  $CF$  новым значением.

Если размер суммы не превышает  $N$  разрядов, всё делается аналогично неограниченной арифметике и флаг переноса  $CF$  равен нулю.

$$\begin{array}{r}
 \overset{1}{0}1010011 \\
 + \overset{1}{0}1001011 \\
 \hline
 10011110
 \end{array} \quad (83_{10} + 75_{10} = 158_{10}) \quad CF = 0 \quad (2.42)$$

Одного разряда для флага переноса достаточно. Даже при сложении двух максимально возможных восьмибитных беззнаковых значений возникает перенос в девятый, но не в десятый разряд:

$$\begin{array}{r}
 \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\
 + 11111111 \\
 + 11111111 \\
 \hline
 11111110
 \end{array} \quad (255_{10} + 255_{10} = 254_{10}) \quad CF = 1 \quad (2.43)$$

Таким образом, в ЭВМ реализована циклическая двоичная арифметика: при сложении операндов  $a$  и  $b$  разрядности  $N$  результат фактически равен  $(a + b) \bmod 2^N$ . Программист может определить корректность результата, анализируя флаг переноса из старшего разряда  $CF$ .

Вычитание выполняется, как и сложение, поразрядно. При необходимости выполняется заём из старшего разряда:

$$\begin{array}{r}
 \overset{\cdot}{1} \overset{\cdot}{0} \overset{\cdot}{0} 0011 \\
 - 1001 \\
 \hline
 111010
 \end{array} \quad (67_{10} - 9_{10} = 58_{10}) \quad (2.44)$$

В случае, когда уменьшаемое меньше вычитаемого, возможен заём из несуществующего девятого разряда. Такая ситуация также отмечается единичным значением флага переноса  $CF$ :

$$\begin{array}{r} \overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot} \\ - \overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot} \\ \hline \overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot} \end{array} \quad (9_{10} - 67_{10} = 198_{10}) \quad CF = 1 \quad (2.45)$$

Как можно заметить, в неограниченной арифметике  $58_{10} + 198_{10} = 256_{10}$ .

Таким образом, вычитание, реализованное в ЭВМ, также циклическое: разность  $a$  и  $b$  разрядности  $N$  равна  $(a - b) \bmod 2^N$ , где под знаком « $-$ » подразумевается вычитание в неограниченной арифметике. Здесь остаток  $(a - b) \bmod 2^N$  всегда неотрицателен (то есть вычисляется по правилам математики, а не C++). Корректность результата можно определить, анализируя флаг  $CF$ , показывающий также заём в старший разряд.

В частности,

$$\begin{array}{r} \overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot} \\ - \overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot} \\ \hline \overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot}\overset{\cdot}{\cdot} \end{array} \quad (0_{10} - 1_{10} = 255_{10}) \quad CF = 1 \quad (2.46)$$

Циклическое сложение (сложение по модулю  $2^N$ ) коммутативно ( $a + b = b + a$ ) и ассоциативно ( $a + (b + c) = (a + b) + c$ ). Циклическое вычитание связано с ним так же, как и обычное вычитание с обычным (неограниченным) сложением. Таким образом, циклическое вычитание антикоммутативно ( $a - b = -(b - a)$ ) и требует смены знаков при раскрытии скобок ( $a - (b - c) = (a - b) + c$ ).

### Умножение и деление

При умножении двух чисел первый множитель последовательно умножается на каждый разряд второго множителя и сдвигается влево на соответствующее количество разрядов (что соответствует умножению на 10, то есть на два). Затем

результаты складываются:

$$\begin{array}{r}
 \times 1010011 \\
 1011 \\
 \hline
 1010011 \\
 1010011 \\
 0000000 \\
 1010011 \\
 \hline
 1110010001
 \end{array}
 \quad (83_{10} \cdot 11_{10} = 913_{10}) \quad (2.47)$$

Как видно из (2.47), разрядность произведения может превысить разрядность множителей более чем на один бит. Поэтому большинство команд умножения процессоров семейства x86, работающие с операндами размера  $N$ , возвращают результат размера  $2N$ .

Действительно, рассмотрим умножение двух максимально возможных восьмибитных беззнаковых значений.

$$\begin{array}{r}
 \times 1111\ 1111 \\
 1111\ 1111 \\
 \hline
 1111\ 1111 \\
 1\ 1111\ 111 \\
 11\ 1111\ 11 \\
 111\ 1111\ 1 \\
 1111\ 1111 \\
 1\ 1111\ 111 \\
 11\ 1111\ 11 \\
 111\ 1111\ 1 \\
 \hline
 1111\ 1110\ 0000\ 0001
 \end{array}
 \quad (255_{10} \cdot 255_{10} = 65025_{10}) \quad (2.48)$$

В умножении столбиком, как правило, переносы между разрядами не записываются, а запоминаются, но в данном примере ввиду его громоздкости перенос показан мелким шрифтом в следующей строке после множителей. Для наглядности столбцы, соответствующие тетрадам произведения, отделены пробелами.

Таким образом, для произведения двух восьмибитных множителей достаточно шестнадцати разрядов.



## 2.5. Представление отрицательных чисел

Если вы уже открываете мне лицевой счёт, то хоть ведите его правильно. Заведите дебет, заведите кредит. В дебет не забудьте внести шестьдесят тысяч рублей, которые вы мне должны, а в кредит — жилет. Сальдо в мою пользу — пятьдесят девять тысяч девятьсот девяносто два рубля. Ещё можно жить.

*Илья Ильф, Евгений Петров.*

*Двенадцать стульев*

Натуральный двоичный код позволяет кодировать только неотрицательные целые числа. При этом в некоторых задачах необходимы целые числа со знаком. В ячейках памяти нельзя непосредственно сохранить знак числа, так что знаковые и, в частности, отрицательные числа необходимо представить в виде комбинации нулей и единиц.

Рассмотрим различные способы кодирования знаковых чисел на восьми разрядах. Всего в восьмиразрядную ячейку можно записать 256 различных комбинаций нулей и единиц. Каждой из этих комбинаций соответствует некоторое неотрицательное число  $u$  в натуральном двоичном коде.

Для кодирования знаковых чисел необходимо поставить в соответствие каждой комбинации новое число (декодированное значение  $x$ ), причём около половины из этих чисел должны быть отрицательными. Это можно сделать различными способами.

К представлению знаковых чисел можно сформулировать следующие пожелания:

- код должен позволять представить с помощью  $N$  битов все целые числа некоторого диапазона  $x \in [\nu_1, \nu_2]$ . Представимый диапазон  $[\nu_1, \nu_2]$  должен



- включать ноль и примерно равное количество положительных и отрицательных чисел;
- представление неотрицательных чисел должно совпадать с их натуральным двоичным кодом;
  - должен существовать простой способ различения положительных и отрицательных чисел;
  - сложение и вычитание должно выполняться с помощью того же сумматора, что и сложение и вычитание неотрицательных чисел.

Три наиболее известных способа представления знаковых чисел на примере восьми разрядов представлены в таблицах 2.5 и 2.6.

Таблица 2.5 показывает кодирование знаковых чисел  $x$  различными способами. Первый столбец содержит десятичное представление знакового числа  $x$ , следующая группа из трёх столбцов показывает двоичный код  $x$  в виде величины со знаком, шестнадцатеричное представление этого кода, а также десятичное представление кода (так как это представление кода, а не самого числа  $x$ , оно беззнаковое). Аналогично, следующая группа из трёх столбцов содержит код с избытком (представлен избыток 128) и его шестнадцатеричное и десятичное представления, третья группа — дополнительный код.

Таблица 2.6 показывает декодирование беззнакового кода  $u$ . Первые три столбца содержат код  $u$  в десятичном, шестнадцатеричном и двоичном виде, четвёртый — значение, получаемое при декодировании  $u$  как величины со знаком, пятый — при декодировании как кода с избытком 128, шестой — как дополнительного кода.

Для кодирования целых знаковых чисел как самостоятельных величин в ЭВМ используется дополнительный код, соответствующих всем перечисленным выше критериям. Представление в виде величины со знаком или кода с избытком используются в кодировании компонент вещественных чисел.

**Различные способы представления знаковых чисел**  
(кодирование)

Таблица 2.5

Значение	Код $u$								
$x_{10}$	Величина со знаком			Код с избытком 128			Дополнительный код (дополнение до 2)		
	$u_2$	$u_{16}$	$u_{10}$	$u_2$	$u_{16}$	$u_{10}$	$u_2$	$u_{16}$	$u_{10}$
−128	отсутствует			0000 0000	00	0	1000 0000	80	128
−127	1111 1111	FF	255	0000 0001	01	1	1000 0001	81	129
−126	1111 1110	FE	254	0000 0010	02	2	1000 0010	82	130
−125	1111 1101	FD	253	0000 0011	03	3	1000 0011	83	131
−124	1111 1100	FC	252	0000 0100	04	4	1000 0100	84	132
...	...	...	...	...	...	...	...	...	...
−4	1000 0100	84	132	0111 1100	7C	124	1111 1100	FC	252
−3	1000 0011	83	131	0111 1101	7D	125	1111 1101	FD	253
−2	1000 0010	82	130	0111 1110	7E	126	1111 1110	FE	254
−1	1000 0001	81	129	0111 1111	7F	127	1111 1111	FF	255
0	1000 0000	80	128	1000 0000	80	128	0000 0000	00	0
	0000 0000	00	0						
+1	0000 0001	01	1	1000 0001	81	129	0000 0001	01	1
+2	0000 0010	02	2	1000 0010	82	130	0000 0010	02	2
+3	0000 0011	03	3	1000 0011	83	131	0000 0011	03	3
...	...	...	...	...	...	...	...	...	...
+123	0111 1011	7B	123	1111 1011	FB	251	0111 1011	7B	123
+124	0111 1100	7C	124	1111 1100	FC	252	0111 1100	7C	124
+125	0111 1101	7D	125	1111 1101	FD	253	0111 1101	7D	125
+126	0111 1110	7E	126	1111 1110	FE	254	0111 1110	7E	126
+127	0111 1111	7F	127	1111 1111	FF	255	0111 1111	7F	127

**Различные способы представления знаковых чисел**  
(декодирование)

Таблица 2.6

Код $u$			Декодированное значение $x$		
$u_{10}$	$u_{16}$	$u_2$	Величина со знаком	Код с избытком 128	Дополнительный код (дополнение до 2)
0	00	0000 0000	+0	−128	0
1	01	0000 0001	+1	−127	+1
2	02	0000 0010	+2	−126	+2
3	03	0000 0011	+3	−125	+3
4	04	0000 0100	+4	−124	+4
5	05	0000 0101	+5	−123	+5
6	06	0000 0110	+6	−122	+6
...	...	...	...	...	...
125	7D	0111 1101	+125	−3	+125
126	7E	0111 1110	+126	−2	+126
127	7F	0111 1111	+127	−1	+127
128	80	1000 0000	−0	0	−128
129	81	1000 0001	−1	+1	−127
130	82	1000 0010	−2	+2	−126
...	...	...	...	...	...
249	F9	1111 1001	−121	+121	−7
250	FA	1111 1010	−122	+122	−6
251	FB	1111 1011	−123	+123	−5
252	FC	1111 1100	−124	+124	−4
253	FD	1111 1101	−125	+125	−3
254	FE	1111 1110	−126	+126	−2
255	FF	1111 1111	−127	+127	−1

### 2.5.1. Величина со знаком

Наиболее очевидный способ кодирования чисел со знаком — выделить один бит (обычно старший) для хранения знака, а в оставшихся хранить абсолютную величину (модуль) числа (столбец «Величина со знаком» таблицы 2.6). Такой код легко читается человеком и для неотрицательных чисел совпадает с натуральным. Код в виде величины со знаком из  $N$  разрядов позволяет представить числа в диапазоне  $[-2^{N-1} + 1, 2^{N-1} - 1]$  — всего  $2^N - 1$  значений, хотя  $N$  разрядов вмещают  $2^N$  различных двоичных кодов.

Это связано с тем, что данный код включает два нуля:  $+0$ , совпадающий с беззнаковым нулём, и  $-0$  — с единичным знаковым битом и нулевым модулем.

Сложение таких чисел с использованием беззнакового сумматора требует большого числа дополнительных действий (в случае слагаемых одного знака модули будут складываться, для разных знаков — вычитаться).

Напротив, умножение и деление отрицательных чисел, представленных в виде величины со знаком, выполняется как беззнаковое умножение или деление величин, дополненное сложением знаков по модулю 2 (xor). Но, так как способ кодирования отрицательных чисел сложился в ранний период развития вычислительной техники, когда аппаратно были реализованы только сложение и вычитание (умножение и деление выполнялись подпрограммами), это достоинство оказалось несущественным.

Соответственно, код в виде величины со знаком не используется для целых чисел. Идея раздельного кодирования знака и абсолютной величины используется при кодировании вещественных чисел с плавающей запятой.

### 2.5.2. Код с избытком

Также для сопоставления знаковых чисел беззнаковым кодам можно задать некоторую константу  $\xi$  и поставить в соответствие каждому знаковому числу  $x$  беззнаковое значение  $u = x + \xi$ . После этого к полученному значению  $u$  применяется натуральное двоичное кодирование. Подобный код называется

кодом с избытком  $\xi$ . Значение  $x$  по коду  $u$ , соответственно, можно найти как  $x = u - \xi$ .

Беззнаковым значение  $u = x + \xi$  будет только для  $x \geq -\xi$ ; соответственно, числа  $x < -\xi$  невозможно закодировать подобным образом. Верхняя граница определяется не только величиной избытка  $\xi$ , но и количеством разрядов  $N$ , отведённых для кода  $u$ . Таким образом, код с избытком  $\xi$  позволяет представить  $N$  разрядами числа в диапазоне  $[-\xi, 2^N - \xi - 1]$ . В отличие от других описанных способов кодирования знаковых чисел, диапазон представимых чисел, в зависимости от значения  $\xi$ , может быть несимметричным и даже может не включать нуля. В частности, запись года двумя цифрами — код с избытком  $\xi = -2000$ .

В пятом столбце таблицы 2.6 показан код с избытком 128. В данном коде присутствует только один ноль, но его код не равен 0000 0000; кроме того, положительные числа кодируются не натуральным кодом.

Для сложения и вычитания чисел, представленных в коде с избытком, можно воспользоваться беззнаковым сумматором, но понадобится коррекция полученного результата. В частности, сложение двух чисел с избытком  $\xi$  даст результат с избытком  $2\xi$ , так что необходимо вычесть  $\xi$ . Таким образом, сложение или вычитание таких чисел требует двух операций сложения/вычитания.

Код с избытком используется для представления порядка вещественных чисел с плавающей запятой, а также в специальной аппаратуре или для передачи данных по каналам связи, если диапазон данных невелик, но заведомо несимметричен относительно нуля (в частности, год или температура в помещении в градусах Цельсия).

### 2.5.3. Дополнительный код

Для того, чтобы записывать ноль и положительное число  $x$  натуральным двоичным (прямым) кодом и при этом иметь возможность пользоваться для знаковых чисел беззнаковым сумматором без коррекции результата, необходимо записывать отрицательное число  $-x$  тем кодом, который получается в результате

беззнакового вычитания  $0 - x$  (с учётом цикличности сложения и вычитания в ЭВМ беззнаковое представление этого кода  $2^N - x$ ).

Такой код называется дополнительным (или дополнением до двух) и представлен в последнем столбце таблицы 2.6. В таблице видно, что с помощью восьми двоичных разрядов можно представить:

- одно значение нуля ( $0 = -0 = 0000\ 0000$ );
- положительные значения от 1 до 127, представленные в натуральном двоичном коде, которым соответствуют коды от 0000 0001 до 0111 1111;
- соответствующие им отрицательные значения от  $-1$  до  $-127$  — коды от 1111 1111 ( $0 - 1$ ) до 1000 0001 ( $0 - 127$ ).

Старший бит называется знаковым, так как у нуля и положительных чисел он равен нулю, у отрицательных — единице. Соответственно, код 1000 0000, который можно в принципе трактовать и как 128, и как  $0 - 128 = -128$ , считается кодом отрицательного числа  $-128$ .

Таким образом, дополнительный код позволяет представить с помощью  $N$  разрядов целые числа в диапазоне  $[-2^{N-1}, 2^{N-1} - 1]$ .

Дополнительный код неотрицательных чисел совпадает с прямым (натуральным).

Для перевода отрицательного числа  $-x$  на практике используется следующая схема. Преобразуем  $0 - x$  с учётом ассоциативности и коммутативности циклического сложения и соответствующих свойств вычитания:

$$-x = 0 - x = (-1 - x) + 1. \quad (2.51)$$

С учётом того, что дополнительный код  $-1$  состоит из единиц во всех разрядах,  $-1 - x$  — это инверсия всех битов  $x$ . Соответственно, дополнительный код  $-x$  может быть рассчитан как  $\neg x + 1$ , где  $\neg x$  — побитовое отрицание (инверсия битов) натурального двоичного представления абсолютной величины числа  $x$ .

Именно в дополнительном коде представлены отрицательные числа в современных вычислительных системах. При этом нет способа, анализируя двоичный код, в частности, 1111 1111, понять, кодирует ли он беззнаковое число 255 или знаковое  $-1$  (или и вовсе что-то иное). Программист должен сам помнить, что именно было записано в данную ячейку, и применять соответствующие команды для обработки и вывода. При программировании на языке высокого уровня (в частности, C++) данную информацию хранит компилятор, но при присваивании переменных различного типа значения могут интерпретироваться по-разному.

#### 2.5.4. Знаковая арифметика в вычислительных системах

В большинстве вычислительных систем целые знаковые числа представляются дополнительным кодом, чтобы иметь возможность складывать и вычитать их теми же командами, что и беззнаковые. Рассмотрим подробнее действия над такими числами.

##### Сложение и вычитание

Дополнительный код построен таким образом, что операции с беззнаковым переполнением получают смысл.

В частности, примеры (2.41) и (2.43), демонстрирующие переполнение при беззнаковом сложении, являются корректными операциями сложения, если рассматривать операнды и результат как знаковые:

$$\begin{array}{rcl}
 \begin{array}{r}
 11111111 \\
 +11111111 \\
 \hline
 1 \\
 00000000
 \end{array} & -1_{10} + 1_{10} = & 0_{10} \\
 \begin{array}{r}
 11111111 \\
 +11111111 \\
 \hline
 11111110
 \end{array} & -1_{10} + (-1_{10}) = & -2_{10}
 \end{array} \tag{2.52}$$

Корректными станут и операции вычитания большего числа из меньшего в (2.45) и (2.46):

$$\begin{array}{r}
 \begin{array}{r}
 \overset{\cdot\cdot}{0}0000\overset{\cdot\cdot}{1}001 \\
 - \overset{\cdot\cdot}{0}10000\overset{\cdot\cdot}{1}1 \\
 \hline
 11000110
 \end{array}
 \end{array}
 \quad 9_{10} - 67_{10} = -58_{10}$$

$$\begin{array}{r}
 \begin{array}{r}
 \overset{\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot}{0}00000000 \\
 - \underset{1}{1} \\
 \hline
 11111111
 \end{array}
 \end{array}
 \quad 0_{10} - 1_{10} = -1_{10}$$
(2.53)

Операции (2.38) и (2.44), где и операнды, и результат неотрицательны и не превышают  $127_{10}$ , корректны и при знаковой, и при беззнаковой трактовке.

Если беззнаковая операция корректна (переполнения не было), но результат превышает максимально возможное знаковое положительное число (для восьми бит  $127_{10}$ ), то при знаковой интерпретации результат перестаёт быть правильным. Например, в (2.42)

$$\begin{array}{r}
 \begin{array}{r}
 \overset{1}{0}10100\overset{1}{1}1 \\
 + \overset{1}{0}10010\overset{1}{1}1 \\
 \hline
 10011110
 \end{array}
 \end{array}
 \quad 83_{10} + 75_{10} = -98_{10}$$
(2.54)

В (2.54), в отличие от (2.52) и (2.53), есть перенос в старший (знаковый) разряд, но нет переноса из него в несуществующий девятый разряд.

Возможна ситуация, когда, наоборот, сумма двух отрицательных чисел имеет ноль в знаковом разряде, **и, соответственно, интерпретируется как положительное число:**

$$\begin{array}{r}
 \begin{array}{r}
 \overset{1}{1}00100\overset{1}{1}1 \\
 + \overset{1}{1}10010\overset{1}{1}1 \\
 \hline
 01011110
 \end{array}
 \end{array}
 \quad -109_{10} + (-53_{10}) = +94_{10}$$
(2.55)

Здесь не было переноса в знаковый разряд, но есть перенос из него.

Подобная ситуация называется **знаковым переполнением**.



Классические случаи знакового переполнения — добавление единицы к максимально возможному положительному числу:

$$\begin{array}{r}
 \begin{array}{cccccccc}
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 + & & & & & & & 1 \\
 \hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}
 \end{array}
 \quad 127_{10} + 1_{10} = -128_{10} \quad (2.56)$$

и вычитание единицы из минимально возможного отрицательного:

$$\begin{array}{r}
 \begin{array}{cccccccc}
 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 - & & & & & & & 1 \\
 \hline
 & 0 & 1 & 1 & 1 & 1 & 1 & 1
 \end{array}
 \end{array}
 \quad -128_{10} - 1_{10} = +127_{10} \quad (2.57)$$

Соответственно, если беззнаковое переполнение — это циклический переход через ноль, то знаковое переполнение для  $N$  бит — циклический переход между  $+2^{N-1} - 1$  и  $-2^{N-1}$  (код этого числа соответствует беззнаковому значению  $2^{N-1}$ ). Для восьми бит это переход от  $+127_{10}$  к  $-128_{10} = 128_{10}$  или наоборот. Именно этот переход и демонстрирует перенос/заём в знаковый разряд, который не компенсируется переносом/заёмом из него.

Таким образом, сложение и вычитание знаковых чисел, представленных в дополнительном коде, может выполняться таким же образом, как и беззнаковых чисел. При этом для проверки корректности результата нужно анализировать не перенос/заём в несуществующий разряд (флаг переноса  $CF$ ), а знаковое переполнение — комбинацию переноса/заёма в знаковый разряд и переноса/заёма из знакового разряда в несуществующий.

Наличие знакового переполнения также отражается специальным флагом — **флагом переполнения** ( $OF$ , Overflow flag).

Соответственно, команды сложения и вычитания не разделяются на знаковые и беззнаковые. По результатам выполнения устанавливается значение как флага переноса (беззнакового переполнения)  $CF$ , так и флага знакового переполнения  $OF$ .

## Умножение и деление

Умножение отрицательных чисел по беззнаковой схеме, показанное, в частности, в (2.48), явно некорректно — результат получается также отрицательным. Это связано с тем, что при сложении частичных произведений подразумевалось, что пустые ячейки в записи умножения «столбиком» — это пропущенные незначащие нули. Но ведущий ноль в дополнительном коде обозначает положительное число, поэтому во всех пустых ячейках необходимо разместить единицы, как показано в (2.58). Для того, чтобы получить все значащие цифры произведения

$$\begin{array}{r}
 \times \dots 111\ 1111\ 1111 \\
 \dots 111\ 1111\ 1111 \\
 \hline
 \begin{array}{cccccccc}
 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
 \dots 111\ 1111\ 1111 \\
 \dots 111\ 1111\ 111 \\
 \dots 111\ 1111\ 11 \\
 \dots 111\ 1111\ 1 \\
 \dots 111\ 1111 \\
 \dots 111\ 111 \\
 \dots 111\ 11 \\
 \dots 111\ 1 \\
 \dots 111 \\
 \dots 11 \\
 \dots 1 \\
 \hline
 \dots 000\ 0000\ 0001
 \end{array}
 \end{array}
 \quad (-1_{10}) \cdot (-1_{10}) = 1_{10} \quad (2.58)$$

(как и для беззнаковых чисел, максимальное их количество равно  $2N$ ), можно не рассматривать бесконечное количество ведущих единиц — достаточно ограничиться размером произведения:

В общем случае знаковые множители дополняются до  $2N$  разрядов знаковым битом (см. раздел 2.7.2). После этого они перемножаются уже без учёта знака и переносов за пределы разрядной сетки, как в (2.59).

Таким образом, если для умножения вычисляются все  $2N$  бит (а для деления — делимое имеет размер  $2N$ ), то знаковое и беззнаковое умножение и деле-

$$\begin{array}{r} \times \begin{array}{cccc} 1111 & 1111 & 1111 & 1111 \\ 1111 & 1111 & 1111 & 1111 \\ \hline \text{EDCB} & \text{A 9 8 7} & \text{6 5 4 3} & \text{2 1} \\ 1111 & 1111 & 1111 & 1111 \\ 1111 & 1111 & 1111 & 111 \\ \dots & & & \\ 11 & & & \\ 1 & & & \\ \hline 0000 & 0000 & 0000 & 0001 \end{array} \end{array} \quad (-1_{10}) \cdot (-1_{10}) = 1_{10} \quad (2.59)$$

ние выполняются по различным алгоритмам и, соответственно, для них должны быть предусмотрены разные команды. Младшие  $N$  бит произведения двух  $N$ -разрядных чисел одинаковы и для беззнакового, и для знакового умножения [74].

Знаковые  $N$ -битные сложение и умножение, так же как и беззнаковые, коммутативны и ассоциативны.

## 2.6. Альтернативная арифметика

— Сегодня нам на доукомплектование прибыло 28 танков. Их нужно распределить по 7 ротам. Я посчитал, на каждую роту получается по 13 танков.

...

— Смотри, лейтенант, ты получаешь 13 танков. 3 танка отдаёшь в 3 взвода, а 1 остаётся тебе. Всё.

## Фольклор

В предыдущих разделах рассматривались позиционное двоичное представление беззнаковых и знаковых целых чисел ограниченной разрядности и циклическая (по модулю  $2^N$ ) арифметика над такими числами. Но для некоторых задач удобнее использовать другое представление или другой способ обработки.

Следует помнить, что как невозможно по представлению в памяти отличить знаковое число от беззнакового, нельзя отличить их и от двоично-десятичных чисел, и от вектора остатков модулярного представления, и т. д. Программист сам должен отслеживать тип данных и применять соответствующие команды.

### 2.6.1. Двоично-десятичная арифметика

Двоично-десятичное представление (binary-coded decimal, BCD) беззнаковых целых чисел — это десятичное позиционное представление, в котором каждая десятичная цифра записана двоичным кодом. Если для записи используется  $n$  десятичных цифр, представимый диапазон чисел —  $[0, 10^n - 1]$ .

Существует множество вариантов как для размера кода десятичных цифр, так и для сопоставления различных кодовых комбинаций цифрам.

Для записи десятичных цифр, принимающих десять различных значений, недостаточно трёх бит (с их помощью можно задать восемь кодовых комбинаций), а четырёх (шестнадцать комбинаций) хватает с избытком. Таким образом, размер двоичного кода десятичной цифры в принципе может принимать значения от четырёх до бесконечности, но на практике используются всего два размера:

- четыре бита (двоичная тетрада) — для максимальной компактности записи;
- байт используемой системы — для упрощения поразрядных десятичных арифметических действий.

Если размер цифры равен байту, значение обычно записано в младшей тетраде байта, а все остальные (старшие) биты равны нулю. Таким образом, достаточно рассмотреть четырёхбитные двоично-десятичные коды.

Так как десятичных цифр десять, а четырёхбитных кодовых комбинаций шестнадцать, то либо шесть комбинаций будут недопустимыми (не будут обозначать никакую цифру), либо некоторые цифры будут кодироваться неоднозначно — двумя и более комбинациями.

Виды представления десятичных цифр (двоично-десятичные коды) делятся на две основные группы.

1. **Взвешенные** коды, когда значение десятичной цифры  $d$  вычисляется по битам  $b_3, b_2, b_1, b_0$  тетрады с использованием постоянных весов:

$$d = \sum_{i=0}^3 q_i \cdot b_i = q_0 \cdot b_0 + q_1 \cdot b_1 + q_2 \cdot b_2 + q_3 \cdot b_3. \quad (2.60)$$

Взвешенный код обычно обозначается своими весами  $q_3q_2q_1q_0$  (может также использоваться обозначение  $q_3-q_2-q_1-q_0$  или  $q_3, q_2, q_1, q_0$ ).

Натуральный двоичный код цифр (также называемый кодом прямого замещения или кодом 8421), для которого значение десятичной цифры  $d$  равно двоичному значению тетрады  $\overline{b_3b_2b_1b_0}$ , является частным случаем взвешенного кода. Такой код легко читается человеком и переводится в двоичный, но при выполнении сложения и вычитания при помощи двоичного сумматора сложно выделить десятичный перенос. Тетрады  $1010_2-1111_2$ , соответствующие значениям  $10_{10}-15_{10}$ , не являются корректными двоично-десятичными цифрами в коде прямого замещения.

Хотя код прямого замещения наиболее популярен в ЭВМ (BCD без уточнения обычно означает именно код прямого замещения), на практике используются и другие взвешенные коды, в частности, код Айкена—Эмери 2421. Веса также могут быть отрицательными.

Основным недостатком взвешенных кодов является то, что, если при передаче будет искажён один из разрядов с большим по модулю весом (в частности,  $\pm 8$  или  $\pm 7$ ), ошибка будет значительно больше, чем при искажении разряда с малым по модулю весом. С этой точки зрения лучше применять невзвешенный код, у которого ошибки, вызванные помехами, были бы одинаковыми для любого разряда.

2. **Невзвешенные** коды — значение десятичной цифры не может быть представлено в виде (2.60). Перечислим некоторые невзвешенные двоично-десятичные коды:

- код с избытком 3 (код  $8421 + 3$ )

$$d = b_0 + 2 \cdot b_1 + 4 \cdot b_2 + 8 \cdot b_3 + 3 \quad (2.61)$$

позволяет относительно просто осуществлять коррекцию после сложения или вычитания двоичным сумматором, в частности, десятичный перенос равен двоичному переносу из тетрады [41];

- код Грея (двоичный рефлексный, или двоичный отражённый код), для которого инверсия любого одного бита изменяет значение на  $\pm 1$  [79].

Процессоры семейства x86 содержат набор команд, облегчающих арифметические действия над цифрами двоично-десятичных чисел в коде прямого замещения. Эти команды доступны при разрядности кода до 32 включительно. В 64-разрядном режиме они исключены, так что коррекцию при операциях с двоично-десятичными цифрами необходимо осуществлять программно [70].

Двоично-десятичные числа, поддерживаемые командами x86, делятся на две разновидности по размеру цифры:

- упакованные — каждый байт содержит две десятичные цифры, представленные в коде прямого замещения; например,  $12_{10}$  кодируется одним байтом 0001 0010;
- неупакованные — байт содержит одну цифру в коде прямого замещения, так что  $12_{10}$  кодируется двумя байтами 0000 0010 0000 0001.

Упакованные числа складываются и вычитаются побайтово (по две десятичные цифры) в два этапа. Сначала выполняется двоичное сложение или вычитание байтов, затем — соответствующая команда коррекции.

Цифры неупакованных чисел можно не только складывать и вычитать, но и умножать и делить. При делении одной цифрой (то есть одним байтом) записываются делитель и частное, делимое состоит из двух десятичных цифр, остаток не вычисляется. Обработка также выполняется в два этапа — сначала двоичная операция над байтом, затем команда коррекции.

Действия над двоично-десятичными числами, содержащими несколько цифр, реализуются программно. Соответственно, длина таких чисел потенциально не ограничена.

В 32-разрядном режиме x86 доступен полный набор команд для обработки неупакованных двоично-десятичных цифр, сложения и вычитания упакованных пар цифр (две четырёхбитных цифры в восьмибитном байте), а также преобразований между упакованной и неупакованной формами. В 64-разрядном режиме многие из этих команд исключены.

Кроме того, математический сопроцессор FPU включает команды для импорта и экспорта 80-битных двоично-десятичных чисел со знаком.

### 2.6.2. Модулярная арифметика

Модулярная арифметика основана на представлении целого неотрицательного числа  $X$  в виде последовательности остатков от деления  $X$  на набор взаимно простых чисел  $p_1, p_2, \dots, p_n$ :

$$X = (x_1, x_2, \dots, x_n), \quad (2.62)$$

где

$$\begin{aligned} x_1 &= X \bmod p_1 \\ x_2 &= X \bmod p_2 \\ \dots & \\ x_n &= X \bmod p_n \end{aligned}, \quad \begin{aligned} &p_1, p_2, \dots, p_n \in \mathbb{N}, \\ &\forall i \neq j : \text{НОД}(p_i, p_j) = 1. \end{aligned}$$

Согласно китайской теореме об остатках, такое представление единственно для любого целого  $X \in [0, P)$ , где  $P = p_1 \cdot p_2 \cdot \dots \cdot p_n$ .

Для вычисления суммы (или произведения) двух чисел, представленных подобным образом, достаточно сложить (или перемножить) каждую пару соответствующих остатков.

Так, пусть

$$\begin{cases} X = (x_1, x_2, \dots, x_n) \\ Y = (y_1, y_2, \dots, y_n) \end{cases}, \quad (2.63)$$

тогда

$$\begin{aligned} X + Y &= ((x_1 + y_1) \bmod p_1, (x_2 + y_2) \bmod p_2, \dots, (x_n + y_n) \bmod p_n) \\ X - Y &= ((x_1 - y_1) \bmod p_1, (x_2 - y_2) \bmod p_2, \dots, (x_n - y_n) \bmod p_n) \\ X \cdot Y &= ((x_1 \cdot y_1) \bmod p_1, (x_2 \cdot y_2) \bmod p_2, \dots, (x_n \cdot y_n) \bmod p_n) \end{aligned} \quad (2.64)$$

Здесь остаток от деления ( $\bmod$ ) вычисляется по правилам математики, а не C/C++ — он неотрицателен (то есть  $(-1) \bmod 7 = 6$ ).

При этом действия с различными парами остатков можно выполнять параллельно, так как действия выполняются независимо друг от друга. При использовании небольших  $p_1, p_2, \dots, p_n$  возможно вместо вычисления результата операции воспользоваться предварительно вычисленной таблицей. Таким образом, любая операция при соответствующей конвейеризации будет выполняться за один машинный такт.

Подобная арифметика в целом циклична по модулю  $P$ , так что, в частности,  $(P - 1) + 1 = 0$ ; ассоциативна и коммутативна.

Некоторые операции выполняются сложнее, чем в позиционных системах счисления. Такие операции называются **немодульными**. Это, прежде всего, **сравнение**, то есть установление порядка (установление равенства — модульная операция), и все операции, в реализации которых оно используется:

- контроль переполнения;
- деление;
- квадратный корень и т. п.

Из-за немодульности сравнения расширение модулярного представления на все целые числа возможно только в виде значения со знаком. При использовании для отрицательных чисел кода со смещением  $x \rightarrow x + \xi$  или дополнительного



кода  $(-x) \rightarrow P - x$  нет простого способа отличить отрицательное число от положительного. На вещественные числа модулярное представление естественным путём не расширяется.

Модулярная арифметика позволяет реализовать очень быстрые вычисления над неотрицательными целыми числами, а также контролировать корректность результата в процессе вычислений, что позволяет обнаруживать аппаратные ошибки.

### 2.6.3. Арифметика с насыщением

В предыдущих разделах рассматривалась циклическая арифметика, то есть, если обрабатываемые числа находятся в диапазоне  $[Min, Max]$ , то  $Max + 1 = Min$  и  $Min - 1 = Max$ . Это основная используемая в ЭВМ схема.

Для некоторых задач в случае выхода за границы допустимого диапазона удобнее схема с насыщением — если «правильный» результат операции превышает наибольшее представимое значение  $Max$ , то он считается равным  $Max$ , если он меньше  $Min$ , то, соответственно, формируется результат  $Min$ .

Таким образом,  $Max + 1 = Max$  и  $Min - 1 = Min$ .

Арифметика с насыщением **коммутативна, но** неассоциативна, в частности

$$(Max - 1) - (Max - 1) = 0,$$

но

$$Max - (1 + Max) + 1 = Max - Max + 1 = 1.$$

Процессоры семейства x86 поддерживают арифметику с насыщением в рамках команд расширения MMX.

Арифметика чисел с плавающей запятой (раздел 2.8.2) не является арифметикой с насыщением.

## 2.7. Битовые операции

Итак, хотя более простая категория исторически могла существовать раньше более конкретной, она в своём полном интенсивном и экстенсивном развитии может быть присуща как раз более сложной форме общества, в то время как более конкретная категория была полнее развита при менее развитой форме общества.

*К. Маркс. Капитал*

Битовые операции — операции, производимые над цепочками битов.

Существует три основных класса битовых операций:

- поразрядные операции (not, and, or, xor);
- операции расширения (увеличения разрядности);
- сдвиги.

### 2.7.1. Поразрядные операции

Поразрядные операции применяются к каждому разряду операнда (для унарных операций) или к каждой паре соответствующих разрядов операндов (для бинарных операций) независимо от соседних разрядов.

Так как один двоичный разряд может принимать только два значения, которые можно трактовать как «ложь» и «истина», набор поразрядных операций обычно включает все логические операции (таблица 2.7).

Для поразрядных операций применяются различные обозначения, наиболее популярные из них показаны в шапке таблицы 2.7. В последующих примерах используются обозначения, принятые в языке C++ (!, &, |, ^).

Результат в каждом разряде рассчитывается независимо от соседних (отсутствует перенос между разрядами). Разрядность результата соответствует разрядности операндов, для бинарных операций разрядность обоих операндов

## Логические операции над разрядами

Таблица 2.7

$a$	$b$	$\neg a$ (not, !)	$a \wedge b$ (and, &)	$a \vee b$ (or,  )	$a \oplus b$ (xor, ^)
0	0	1	0	0	0
0	1		0	1	1
1	0	0	0	1	1
1	1		1	1	0

должна быть одинаковой. Поразрядные операции не могут быть знаковыми или беззнаковыми: обработка старшего бита не отличается от остальных.

Таким образом, поразрядное отрицание (битовая инверсия, дополнение до единицы) — это унарная операция, где к каждому разряду единственного операнда применяется логическое отрицание:

$$!2_{10} = !0000\,0010 = 1111\,1101 \quad (2.65)$$

Полученное значение можно трактовать как  $-3_{10}$ , если интерпретировать результат как знаковый, либо как  $253_{10}$ , если интерпретировать его как беззнаковый. Ни одно из этих двух значений здесь не будет «правильным» или «ошибочным».

Если трактовать операнды как знаковые и представленные в дополнительном коде,  $\forall x : !x = -x - 1$ .

При этом необходимо помнить, что если операнд  $2_{10}$  имеет бóльшую разрядность (и, соответственно, включает больше ведущих нулей), то и результат будет содержать больше ведущих единиц. Соответственно, в случае разрядности  $N$  значение  $!2_{10}$  в знаковой интерпретации будет иметь значение  $-3_{10}$ , а в беззнаковой —  $(2^N - 3)_{10}$ .

Поразрядное «и» (конъюнкция) — бинарная операция, к каждой паре разрядов операндов применяется логическое «и»:

$$3_{10} \& 5_{10} = 0000\,0011 \& 0000\,0101 = 0000\,0001 = 1_{10} \quad (2.66)$$

разряд результата равен нулю, если хотя бы один операнд содержит ноль в соответствующем разряде.

При помощи поразрядного «и» можно получить неотрицательный остаток от деления целого числа  $x$  на  $2^n$ . В этом случае второй операнд («маска») состоит из  $n$  единиц:  $x \bmod 2^n = x \ \& \ (2^n - 1)$ .

$$189_{10} \bmod 8_{10} = 189_{10} \ \& \ 7_{10} = 1011\ 1101 \ \& \ 0000\ 0111 = 0000\ 0101 = 5_{10} \quad (2.67)$$

Поразрядное «или» (дизъюнкция) — бинарная операция, к каждой паре разрядов операндов применяется логическое «или»:

$$3_{10} \mid 5_{10} = 0000\ 0011 \mid 0000\ 0101 = 0000\ 0111 = 7_{10} \quad (2.68)$$

разряд результата равен единице, если хотя бы один операнд содержит единицу в соответствующем разряде.

Поразрядное исключающее «или» (также называется сложением по модулю два, но, в отличие от арифметической операции сложения, нет переноса между разрядами) — бинарная операция, к каждой паре разрядов операндов применяется исключающее «или»:

$$3_{10} \wedge 5_{10} = 0000\ 0011 \wedge 0000\ 0101 = 1111\ 0110 = -10_{10} = 246_{10} \quad (2.69)$$

разряд результата равен единице, если один и только один операнд содержит единицу в соответствующем разряде.

Так как для одного разряда  $\forall b \in \{0, 1\} : 1 \oplus b = \neg b$ , для чисел любой разрядности верно  $\forall x : (-1) \wedge x = !x$ .

Поразрядные логические операции используются как в алгоритмах криптографии, так и, в некоторых ситуациях, для ускорения арифметических вычислений. Последнее возможно, только если один из операндов является константой специфического вида.

### 2.7.2. Расширение целых чисел

Часто необходимо увеличить разрядность числа, сохранив его значение. В C++ подобное происходит, в частности, при присваивании переменных разного размера:

```
1 int i;  
2 short int s = -1;  
3 i = s;
```

Операция увеличения разрядности называется расширением  $n$ -разрядного числа  $x$  до  $m$  разрядов ( $m > n$ ).

Младшие  $n$  разрядов результата совпадают с расширяемым значением  $x$ , старшие  $m - n$  должны быть как-то инициализированы.

Существует две операции расширения, по-разному инициализирующие расширяемую часть:

- беззнаковое расширение — расширяемая часть заполняется нулями (такая операция сохраняет значение беззнаковой интерпретации  $x$ );
- знаковое расширение — расширяемая часть заполняется значением знакового бита (сохраняет значение знаковой интерпретации  $x$ ).

В таблице 2.8 показаны примеры знакового и беззнакового расширения восьмибитных чисел до шестнадцати бит.

Каждая строка таблицы соответствует одному значению (цепочке битов). Во втором столбце показано его двоичное представление, в третьем — шестнадцатеричное, в четвёртом — десятичное представление его знаковой интерпретации, в пятом — десятичное представление беззнаковой интерпретации.

Строки сгруппированы по три — восьмибитное значение  $x$ , его беззнаковое расширение до шестнадцати бит и знаковое расширение до того же размера.

Видно, что для неотрицательных чисел знаковое и беззнаковое расширение выполняется одинаково.

## Знаковое и беззнаковое расширение

Таблица 2.8

Значение	<i>bin</i>	<i>hex</i>	<i>dec</i> <sub>знак</sub>	<i>dec</i> <sub>беззн</sub>
$x$ (8 бит)	0000 0000	00	0	0
$x \xrightarrow{\text{беззнаковое}} 16 \text{ бит}$	0000 0000 0000 0000	0000	0	0
$x \xrightarrow{\text{знаковое}} 16 \text{ бит}$	0000 0000 0000 0000	0000	0	0
$x$ (8 бит)	0000 0001	01	1	1
$x \xrightarrow{\text{беззнаковое}} 16 \text{ бит}$	0000 0000 0000 0001	0001	1	1
$x \xrightarrow{\text{знаковое}} 16 \text{ бит}$	0000 0000 0000 0001	0001	1	1
$x$ (8 бит)	0000 1111	0F	15	15
$x \xrightarrow{\text{беззнаковое}} 16 \text{ бит}$	0000 0000 0000 1111	000F	15	15
$x \xrightarrow{\text{знаковое}} 16 \text{ бит}$	0000 0000 0000 1111	000F	15	15
$x$ (8 бит)	1000 0000	80	−128	128
$x \xrightarrow{\text{беззнаковое}} 16 \text{ бит}$	0000 0000 1000 0000	0080	128	128
$x \xrightarrow{\text{знаковое}} 16 \text{ бит}$	1111 1111 1000 0000	FF80	−128	65 408
$x$ (8 бит)	1111 1111	FF	−1	255
$x \xrightarrow{\text{беззнаковое}} 16 \text{ бит}$	0000 0000 1111 1111	00FF	255	255
$x \xrightarrow{\text{знаковое}} 16 \text{ бит}$	1111 1111 1111 1111	FFFF	−1	65 535

### 2.7.3. Битовые сдвиги

Битовые сдвиги — семейство бинарных операций с несимметричными операндами. Один из операндов представляет собой цепочку битов, второй — неотрицательное целое число — величину сдвига. Цепочка битов смещается вправо или влево на указанное количество битов. Разрядность результата равна разрядности операнда-цепочки. Кроме явных операндов и результата, в сдвигах используется флаг переноса *CF*.

Сдвиг на  $n$  разрядов эквивалентен повторённому  $n$  сдвигу на один разряд, соответственно, для простоты ниже рассматривается только сдвиги на один бит.

Как было сказано выше, сдвиги различаются направлением — вправо (в сторону младших разрядов) или влево (в сторону старших). Кроме того, при сдвиге крайний с одной стороны разряд выдвигается за пределы разрядной сетки (его значение попадает в ячейку флага переноса  $CF$ ), а крайняя ячейка с другой стороны освобождается.

В зависимости от способа заполнения освободившейся ячейки различаются следующие типы сдвига:

- циклические.

Ниже различные виды сдвигов рассматриваются на примере восьмибитных значений.

### Логический и арифметический сдвиг вправо

В случае логического сдвига вправо (в сторону младших разрядов) освободившийся старший разряд инициализируется нулём (рис. 2.3, а). В случае арифметического сдвига вправо — копией знакового бита (рис. 2.3, б).

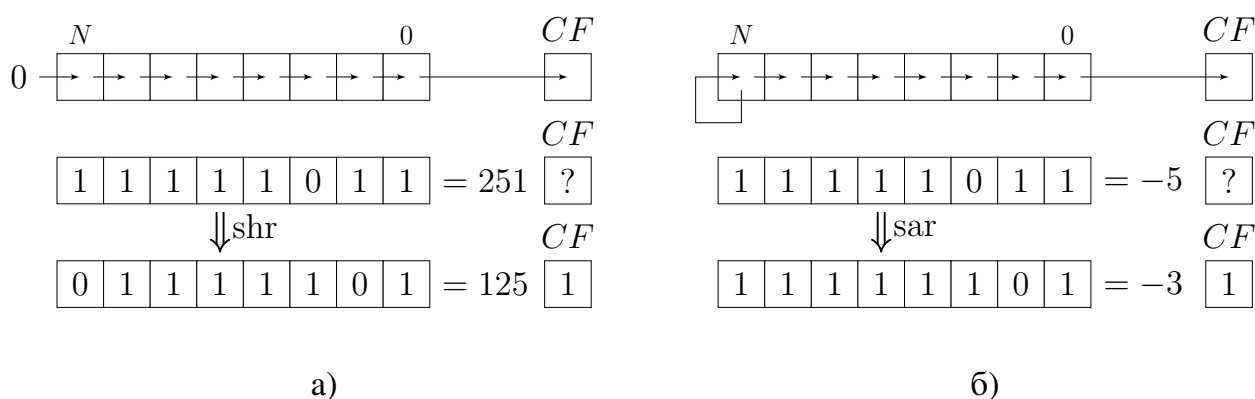


Рис. 2.3. Беззнаковый (а) и знаковый (б) сдвиги вправо

Логический сдвиг вправо соответствует беззнаковому делению на 2 с остатком, арифметический — знаковому. Знаковое деление при этом выполняется по правилам математики, подразумевающим неотрицательный остаток даже при

отрицательном делимом, то есть при делении минус единицы на 2 частное равно минус единице, а остаток — плюс единице.

Остаток как для знакового, так и для беззнакового деления на 2 сдвигом на один бит равен биту  $CF$  (вышедшему за разрядную сетку младшему биту исходного числа).

Сдвиг вправо на  $n$  разрядов соответствует делению на  $2^n$ , причём в случае знакового (арифметического) сдвига отрицательных чисел также выполняется «математическое» деление, подразумевающее неотрицательный остаток.

Во многих языках программирования (в частности, в C/C++) считается, что при делении знаковых чисел остаток может быть отрицательным, то есть при делении минус единицы на 2 частное будет равно нулю, а остаток — минус единице. Команды знакового деления ЭВМ реализуют именно эту схему.

Результат «программистского» деления отрицательного числа  $x$  на  $2^n$  в общем случае не равен результату «математического» (подразумевающего неотрицательный остаток и рассчитываемого сдвигом  $x \text{ sar } n$ ). Они совпадают только в том случае, когда  $x$  делится на  $2^n$  нацело. Для всех остальных  $x < 0$  результат «программистского» деления будет на единицу больше.

Таким образом, чтобы получить при помощи арифметического сдвига значение «программистского» частного, необходимо до сдвига прибавить к отрицательному делимому значение  $2^n - 1$ . Для положительного делимого или нуля коррекция не требуется.

### **Логический и арифметический сдвиг влево**

В случае сдвига влево (в сторону старших разрядов) освободившийся младший бит инициализируется нулём (рис. 2.4).

Арифметический и логический сдвиги влево не различаются. Сдвиг влево на один бит эквивалентен умножению на 2 (так как разрядность результата равна разрядности исходной цепочки, неважно — знаковому или нет). Коррекция операндов не требуется.



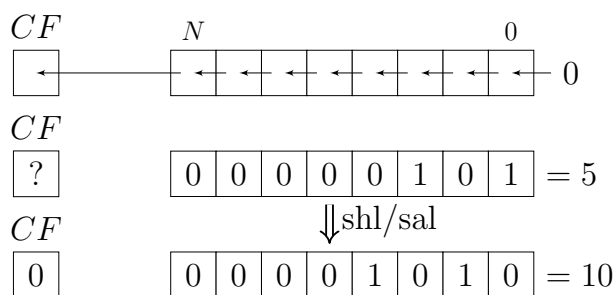


Рис. 2.4. Знаковый (беззнаковый) сдвиг влево

Битовые сдвиги выполняются гораздо быстрее, чем умножение и деление с помощью специализированных команд. Соответственно, умножение и деление на специальные константные значения часто выполняются оптимизирующими компиляторами при помощи сдвигов или комбинации сдвигов и сложения.

### Циклические сдвиги

В случае простого циклического сдвига освободившаяся ячейка с одной стороны замещается разрядом, вышедшим с другой стороны за разрядную сетку (рис. 2.5, а) и б).

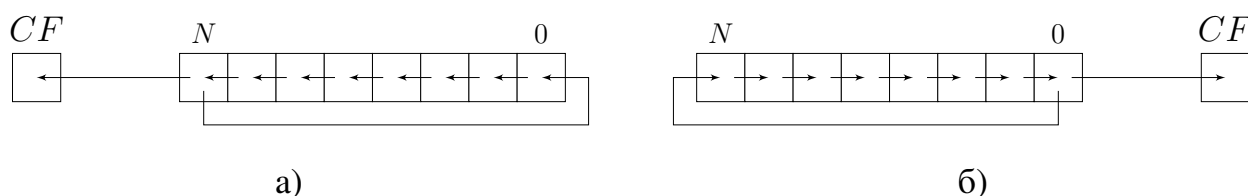


Рис. 2.5. Простой циклический сдвиг: а) влево, б) вправо

В случае циклического сдвига через флаг переноса освободившаяся ячейка инициализируется значением флага переноса  $CF$ , а ячейка  $CF$  замещается разрядом, вышедшим за разрядную сетку (рис. 2.6, а) и б). Таким образом, результат зависит не только от операндов, но и от текущего значения  $CF$ .

Если простой циклический сдвиг  $N$ -разрядного числа циклически перемещает разряды  $N$ -битной цепочки, то сдвиг через флаг переноса —  $(N + 1)$ -битной цепочки (операнд +  $CF$ ).

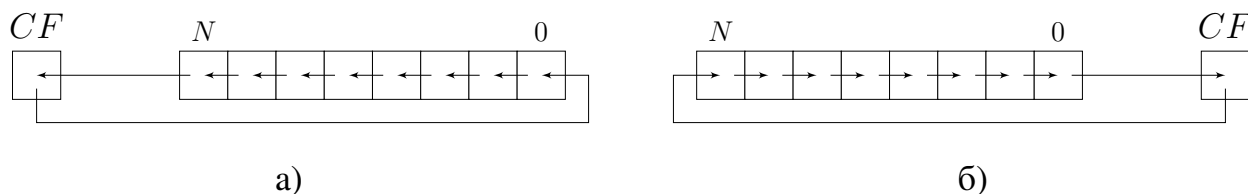


Рис. 2.6. Циклический сдвиг через флаг переноса: а) влево,  
б) вправо

Циклические сдвиги не эквивалентны какой-либо арифметической операции. Они широко используются в криптографии.

## 2.8. Представление вещественных чисел

Опять скажу: никто не обнимет необъятного!

*К. П. Прутков.*

*Плоды раздумья. Мысли и афоризмы*

Какой бы код ни использовался, с помощью  $N$  битов  $b_{N-1} \dots b_1 b_0$  можно получить только  $2^N$  разных кодовых комбинаций и, соответственно, представить не более  $2^N$  значений. При этом, в отличие от целых чисел, вещественных значений в любом диапазоне  $[\nu_1, \nu_2] \subseteq \mathbb{R}$  бесконечно много (причём эта бесконечность несчётна, то есть количество вещественных чисел, помещающихся на любом интервале, превышает общее число существующих целых чисел).

Таким образом, для того, чтобы закодировать вещественные числа, необходимо не только ограничить допустимый диапазон, но и проредить его внутреннюю часть. Большую часть вещественных чисел описываемого диапазона  $[\nu_1, \nu_2] \subseteq \mathbb{R}$  невозможно точно представить в ЭВМ.

Представление вещественных чисел основано на описанной в разделе 2.3.4 позиционной двоичной записи.

### 2.8.1. Представление вещественных чисел с фиксированной запятой

Представим вещественное число  $X$  в виде бесконечной двоичной дроби (2.11), округлим до  $n$  разряда после запятой, обозначив результат округления  $\check{X}$ :

$$\begin{aligned} X \approx \check{X} &= x_k \cdot 2^k + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0 + x_{-1} \cdot 2^{-1} + \dots + x_{-n} \cdot 2^{-n} = \\ &= x_k \cdot 2^k + \dots + x_1 \cdot 2 + x_0 + \frac{x_{-1}}{2} + \dots + \frac{x_{-n}}{2^n} = \\ &= \overline{x_k \dots x_1 x_0, x_{-1} \dots x_{-n}}, \quad 0 \leq x_i < 2 \text{ — целые} \end{aligned} \quad (2.70)$$

и последовательно запишем цифры  $x_k, \dots, x_1, x_0, x_{-1}, \dots, x_{-n}$  в биты  $b_{N-1}, \dots, b_1, b_0$ . Таким образом, из  $N$  битов, отведённых под представление числа, для дробной части используется  $n$ , для целой остаётся  $k = N - n$ .

$$\begin{aligned} X \approx \check{X} &= b_{N-1} \cdot 2^{N-n-1} + \dots + b_{n+1} \cdot 2^1 + b_n \cdot 2^0 + b_{n-1} \cdot 2^{-1} + \dots + b_0 \cdot 2^{-n} = \\ &= b_{N-1} \cdot 2^{N-n-1} + \dots + b_{n+1} \cdot 2 + b_n + \frac{b_{n-1}}{2} + \dots + \frac{b_0}{2^n} = \\ &= \overline{b_{N-1} \dots b_n, b_{n-1} \dots b_0} \end{aligned} \quad (2.71)$$

Двоичная запятая, отделяющая дробную часть от целой, всегда расположена между разрядами  $n$  и  $n - 1$ . Соответственно, такой способ кодирования называется представлением с фиксированной запятой.

Целая часть числа может принимать значения от 0 до  $2^{N-n} - 1$ , дробная — от  $0 = \overline{0,00 \dots 00}$  до  $1 - \frac{1}{2^n} = \overline{0,11 \dots 11}$  (всего  $2^n$  значений). Таким образом, можно представить числа в диапазоне  $[0, 2^{N-n} - \frac{1}{2^n}]$ . На каждом полуинтервале  $[\nu, \nu + 1)$  находится  $2^n$  чисел, представимых в виде с фиксированной запятой с  $n$ -битной дробной частью.

При округлении вещественного числа  $X \in [0, 2^{N-n} - \frac{1}{2^n}]$  до  $n$  двоичных разрядов после запятой абсолютная погрешность округления не превышает  $\frac{1}{2^n}$ :

$$|X - \check{X}| \leq \frac{1}{2^n} \quad (2.72)$$

относительная погрешность увеличивается при уменьшении абсолютной величины числа:

$$\left| \frac{X - \check{X}}{X} \right| \leq \frac{1}{2^n \cdot |X|} \quad (2.73)$$

Нетрудно заметить, что при умножении (2.71) на  $2^n$  получим целое число:

$$\begin{aligned} 2^n \cdot \check{X} &= 2^n \cdot \overline{b_{N-1} \dots b_n b_{n-1} \dots b_0} = \\ &= 2^n \cdot \left( b_{N-1} \cdot 2^{N-n-1} + \dots + b_{n+1} \cdot 2 + b_n + \frac{b_{n-1}}{2} + \dots + \frac{b_0}{2^n} \right) = \\ &= b_{N-1} \cdot 2^{N-1} + \dots + b_{n+1} \cdot 2^{n+1} + b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \dots + b_0 \cdot 2^0 = \\ &= \overline{b_{N-1} \dots b_n b_{n-1} \dots b_0} = x, \quad x \in \{0, 1, \dots, 2^N - 1\} \end{aligned} \quad (2.74)$$

то есть существует взаимно однозначное соответствие между числами с фиксированной запятой и целыми числами в диапазоне  $[0, 2^N - 1]$ :

$$\check{X} = \frac{x}{2^n} \quad (2.75)$$

причём  $\check{X}$  и  $x$  записываются одной и той же комбинацией бит.

В некоторых вычислительных машинах работа с числами с фиксированной запятой реализована аппаратно. При этом отрицательные числа часто кодируются как значение со знаком.

Если специального набора команд нет, арифметика с фиксированной запятой может быть реализована программно с использованием целочисленных команд. Сложение и вычитание чисел с фиксированной запятой можно выполнять целочисленным сумматором без коррекции результата:

$$\check{X} + \check{Y} = \frac{x}{2^n} + \frac{y}{2^n} = \frac{x + y}{2^n} \quad (2.76)$$

После целочисленного умножения требуется коррекция в виде деления результата на  $2^n$ :

$$\check{X} \cdot \check{Y} = \frac{x}{2^n} \cdot \frac{y}{2^n} = \frac{x \cdot y}{2^{2n}} = \frac{\frac{x \cdot y}{2^n}}{2^n} \quad (2.77)$$

после целочисленного деления потребовалась бы коррекция в виде умножения результата на  $2^n$ , если бы этот результат был вещественным:

$$\frac{\check{X}}{\check{Y}} = \frac{\frac{x}{2^n}}{\frac{y}{2^n}} = \frac{x}{y} = \frac{\frac{x}{y} \cdot 2^n}{2^n} \quad (2.78)$$

но коррекция целочисленного частного умножением даст нулевые значения младших  $n$  бит, то есть дробной части числа с фиксированной запятой, что неправильно. Поэтому необходимо проводить коррекцию перед делением:

$$\frac{\check{X}}{\check{Y}} = \frac{\frac{x \cdot 2^n}{y}}{2^n} \quad (2.79)$$

Умножение и деление на  $2^n$  может быть выполнено при помощи битовых сдвигов, что практически не замедлит работу.

Таким образом, представление с фиксированной запятой позволяет представить числа в малом диапазоне с ограниченной абсолютной погрешностью и позволяет использовать для арифметических действий над вещественными числами быстрые целочисленные и логические операции.

### 2.8.2. Представление вещественных чисел с плавающей запятой

В основе представления вещественных чисел с плавающей запятой лежит экспоненциальный (научный) формат:

$$X = N^p \cdot \mu \quad (2.80)$$

где  $N$  — основание системы счисления (в современных ЭВМ  $N = 2$ , стандарт арифметики с плавающей точкой IEEE 754 [13, 14, 80] описывает также случай  $N = 10$ ),  $\mu$  называется мантиссой числа  $X$ , целое число  $p$  — порядком (иногда из-за английского *exponent* используется термин «экспонента», но он не принят в отечественной литературе).

В настоящее время чаще всего, кроме порядка и мантиссы, отделяется ещё и знак числа:

$$X = (-1)^s \cdot N^p \cdot \mu, \quad s \in \{0, 1\}, p \in \mathbb{Z}, \mu \geq 0 \quad (2.81)$$

В форме (2.81) можно представить любое конечное вещественное число, но не единственным способом:

$$512,12 = 512,12 \cdot 10^0 = 51212 \cdot 10^{-2} = 51,212 \cdot 10^1 = 0,051212 \cdot 10^4$$

Представление (2.81) называется нормализованным, если  $0,1_N \leq \mu < 1$ :

$$X = (-1)^s \cdot N^p \cdot \mu, \quad s \in \{0, 1\}, p \in \mathbb{Z}, 0,1_N \leq \mu < 1 \quad (2.82)$$

В нормализованной форме (2.82) можно представить любое конечное вещественное число, кроме нуля, причём единственным образом. В частности,  $512,12 = 0,51212 \cdot 10^3$ . В двоичной системе счисления то же самое число и его нормализованное экспоненциальное представление записываются как  $1000000000,0001111 \dots = 0,100000000000001111 \dots \cdot 2^9$ .

Найдём порядок нормализованного представления числа  $X$ :

$$\log_N |X| = \log_N (N^p \cdot \mu) = p + \log_N \mu \quad (2.83)$$

так что  $p = \log_N |X| - \log_N \mu$ . Так как  $0,1_N \leq \mu < 1$ , то  $-1 \leq \log_N \mu < 0$ :

$$\log_N |X| < p \leq \log_N |X| + 1 \quad (2.84)$$

с учётом того, что  $p \in \mathbb{Z}$ , получаем

$$p = \left\lfloor \log_N |X| + 1 \right\rfloor \quad (2.85)$$

Если записать мантиссу нормализованного представления числа  $X$  в позиционной форме, получим

$$\mu = \overline{0, m_1 m_2 m_3 m_4 \dots}, \quad m_1 \neq 0. \quad (2.86)$$

где  $m_i \in \{0, 1, \dots, N - 1\}$  — цифры. Если для представления с фиксированной запятой до определённого знака округляется дробная часть числа  $X$  (что даёт ограниченную абсолютную погрешность), то для представления с плавающей запятой до определённой длины округляется мантисса (что приводит к ограниченной относительной погрешности округления).

### Структура двоичного числа с плавающей запятой согласно IEEE 754

Представим вещественное число  $X \neq 0$  в нормализованной двоичной форме (2.81)

$$X = (-1)^s \cdot 2^p \cdot \mu, \quad s \in \{0, 1\}, p \in \mathbb{Z}, 0,1_2 \leq \mu < 1 \quad (2.87)$$

Запишем мантиссу в позиционной двоичной форме  $\mu = 0,\overline{m_1 m_2 m_3 m_4 \dots}$ , где  $m_i$  — двоичные цифры, 0 или 1. Так как представление (2.87) нормализовано,  $m_1 \neq 0$ . В двоичной системе если  $m_1$  не равна нулю, то она равна единице, то есть  $\mu = 0,1\overline{m_2 m_3 m_4 \dots}$

$$X = (-1)^s \cdot 2^p \cdot \overline{0,1 m_2 m_3 m_4 \dots}, \quad s \in \{0, 1\}, p \in \mathbb{Z} \quad (2.88)$$

Запишем в память компоненты такого представления (рис. 2.7). Знак  $s$  занимает старший бит.

Следующие  $w$  бит занимает порядок  $p$ . Порядок представляется кодом с избытком, то есть после знакового бита следует натуральный код значения  $p + \xi$ , где  $\xi = 2^{w-1} - 2$  постоянно для формата. Значение  $p + \xi$  называется смещённым порядком.

Минимально представимое значение смещённого порядка кодируется строкой из одних нулей, максимально представимое — строкой из единиц. Оба они считаются специальными — смещённому порядку 000...000 соответствуют нули и денормализованные числа, 111...111 — бесконечности, неопределённость и нечисла.

Таким образом, минимальное допустимое значение смещённого порядка нормализованного числа кодируется как 000...001, а собственно порядок равен

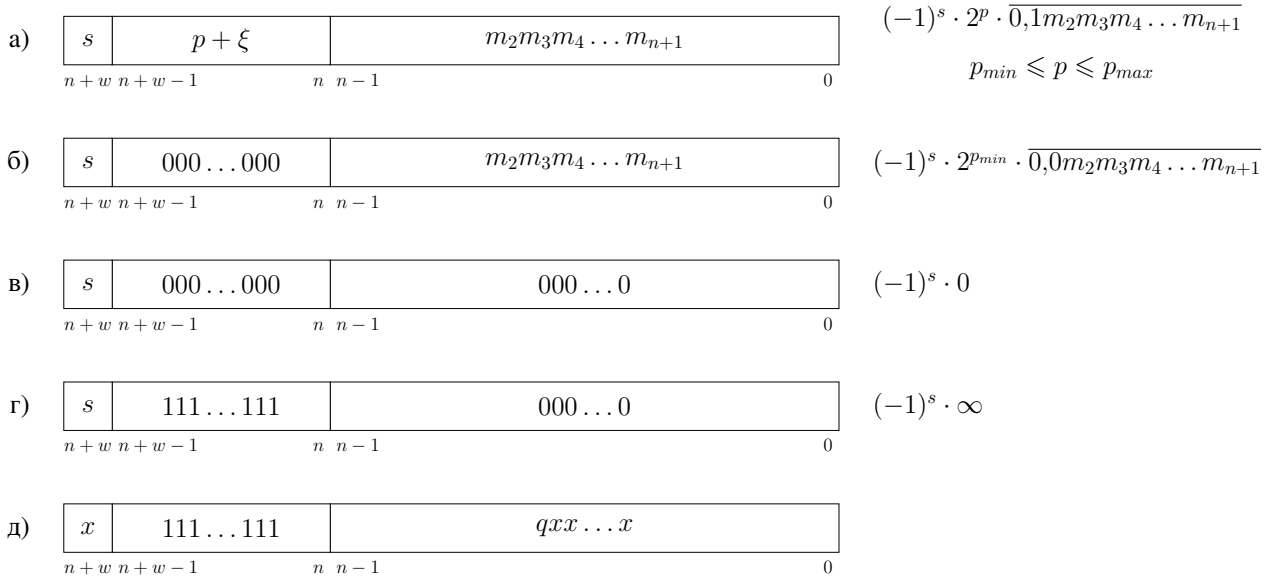


Рис. 2.7. Структура числа с плавающей запятой согласно стандарту IEEE 754: а) нормализованное число, б) денормализованное, в) ноль, г) бесконечность, д) неопределённость или нечисло

$p_{min} = 1 - \xi = 3 - 2^{w-1}$ , максимальное — кодируется как  $111 \dots 110$ , соответственно максимальный порядок числа  $p_{max} = 2^w - 2 - \xi = 2^{w-1}$ .

Последние  $n$  бит числа с плавающей запятой занимает округлённая мантийса.

Так как для нормализованного числа старший бит мантийсы  $m_1$  всегда равен единице, его не имеет смысла хранить. Соответственно,  $n$  бит мантийсы хранят разряды от  $m_2$  до  $m_{n+1}$ . В поле смещённого порядка нормализованного числа записывается натуральный двоичный код  $p + \xi$  (рис. 2.7, а).

Если порядок числа  $X$  слишком мал ( $p < p_{min}$ ), число представляют в виде:

$$X = (-1)^s \cdot 2^{p_{min}} \cdot \mu, \quad s \in \{0, 1\}, \quad 0 \leq \mu < 0,1_2 \quad (2.89)$$

и называют денормализованным.

Старший бит  $m_1$  мантийсы денормализованного числа всегда равен нулю ( $\mu = \overline{0,0m_2m_3m_4 \dots}$ ), так что его тоже не имеет смысла хранить. В поле



мантиссы записываются разряды от  $m_2$  до  $m_{n+1}$ . В поле смещённого порядка денормализованного числа записывается специальное значение  $000 \dots 000$  (рис. 2.7, б).

Если смещённый порядок равен  $000 \dots 000$  и при этом все биты мантиссы равны нулю, получаем значение  $\pm 0$  (рис. 2.7, в):

$$X = (-1)^s \cdot 2^{p_{min}} \cdot \mu, \quad s \in \{0, 1\}, \quad \mu = \overline{0,0000 \dots 0} \quad (2.90)$$

Нули считаются не денормализованными, а специальными значениями, хотя и могут быть декодированы по формуле (2.89).

Если смещённый порядок состоит только из единиц (равен  $111 \dots 111$ ), а поле мантиссы — только из нулей, получаем специальное значение бесконечности (рис. 2.7, г).

В зависимости от поля знака, существуют два значения бесконечности и два нуля, так что  $\frac{1}{+0} = +\infty$  и  $\frac{1}{-0} = -\infty$ . На рис. 2.7, в) и г) это показано как  $(-1)^s \cdot 0$  и  $(-1)^s \cdot \infty$  соответственно.

Если смещённый порядок равен  $111 \dots 111$ , а поле мантиссы содержит не только нули, получаем так называемые нечисла (рис. 2.7, д). Нечисла не имеют знака, бит  $s$  игнорируется.

Если при этом старший сохраняемый бит мантиссы ( $q$  на рис. 2.7, д) равен единице, это так называемое тихое нечисло, или вещественная неопределённость (получаемая, в частности, как  $\frac{0}{0}$ ). Если  $q = 0$ , нечисло называется сигнальным и не может быть результатом вещественной операции.

### **Форматы двоичных чисел с плавающей запятой согласно IEEE 754**

Стандарт IEEE 754-1985 описывает два двоичных формата с плавающей запятой — 32-битный формат одинарной точности и 64-битный формат двойной точности. В IEEE 754-2008 были добавлены 16- и 128-битный двоичный форматы (от названий в новой версии формата отказались), а также описана общая формула  $k$ -битного двоичного формата для  $k \geq 128$  (таблица 2.9).

## Стандартные двоичные форматы с плавающей запятой

Таблица 2.9

Общая длина $n + w + 1$ , бит	16	32	64	128	$k \geq 128$
Длина кода знака $s$ , бит	1	1	1	1	1
Длина кода порядка $w$ , бит	5	8	11	15	$\text{round}(4 \cdot \log_2 k) - 13$
Длина кода мантииссы $n$ , бит	10	23	52	112	$k - w - 1$

Также IEEE 754-2008 описывает возможность расширения стандартных форматов с увеличением как точности мантииссы, так и диапазона порядка.

Кроме двоичных, IEEE 754-2008 описывает два десятичных формата длины 64 и 128 бит, а также формулы для десятичного формата длины  $k = 32\kappa$ . Десятичные форматы IEEE 754-2008 имеют более сложную структуру, чем двоичные. Для экономии памяти в мантииссе для цифр  $m_2, m_3, m_4 \dots$  используется кодирование троек десятичных цифр группами по десять бит (так как число кодовых комбинаций  $2^{10} = 1024 > 10^3$ , это возможно), а код порядка соединён с кодом старшей цифры  $m_1 \neq 0$ .

## Структура нестандартного числа FPU x87

В математическом сопроцессоре x87, входящем в состав процессоров линейки x86, используется нестандартный формат вещественных чисел (рис. 2.8), так как первый подобный сопроцессор был выпущен задолго до первой редакции стандарта IEEE 754.

Порядок этого формата занимает  $w = 15$  бит, мантиисса —  $n = 64$  бита. Общий размер числа  $k = n + w + 1 = 80$  бит.

Формат сопроцессора x87 отличается от стандартных не только разрядностью полей (IEEE 754 предусматривает расширенные форматы с увеличенной разрядностью), но и тем, что мантиисса включает старший бит (единицу для нормализованных чисел и ноль для денормализованных). Благодаря этому, кроме

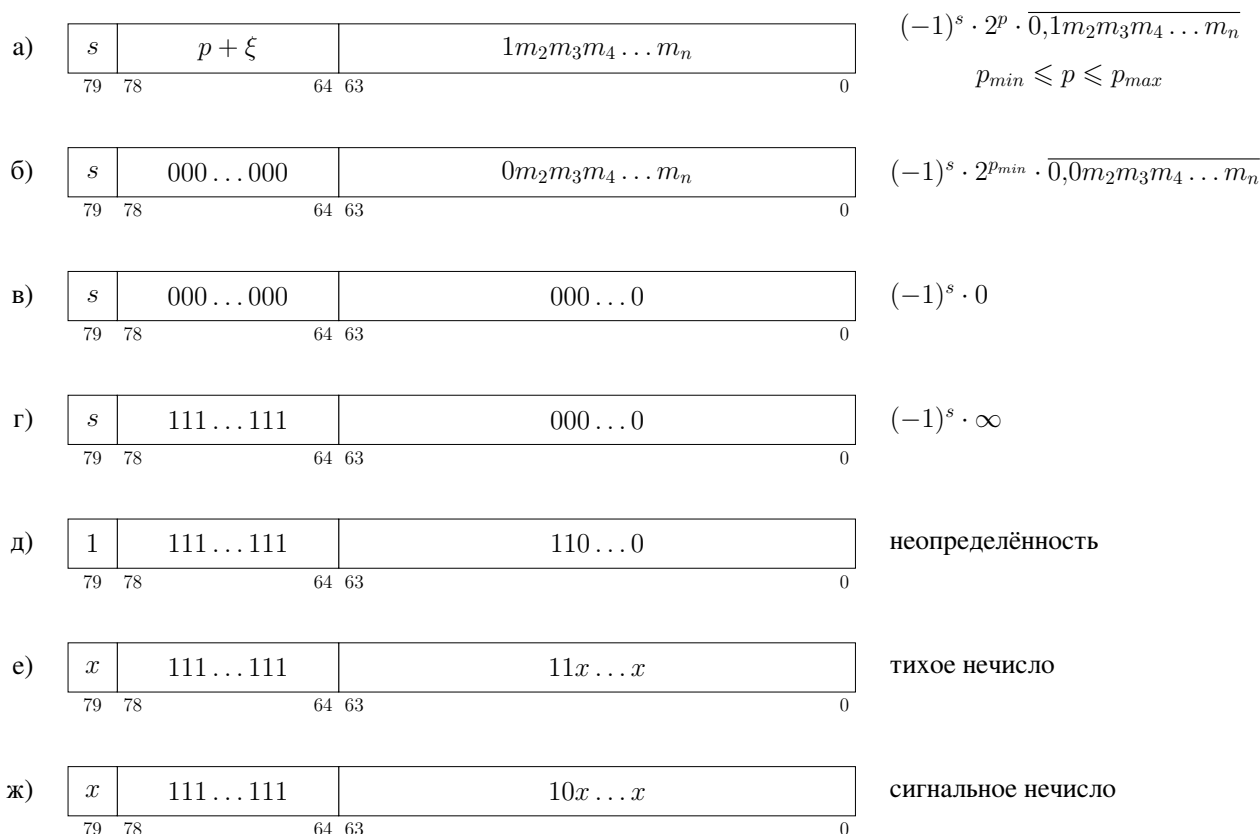


Рис. 2.8. Структура внутреннего представления чисел в FPU

х87: а) нормализованное число, б) денормализованное, в) ноль,  
 г) бесконечность, д) вещественная неопределённость,  
 е) тихое нечисло, ж) сигнальное нечисло

тихих и сигнальных нечисел, возможны недопустимые значения, для которых старший бит мантиссы не соответствует порядку.

Также сопроцессор х87 различает несколько видов тихих нечисел (рис. 2.8, е). Только один из них является вещественной неопределённостью (рис. 2.8, д).

Недопустимыми считаются значения, не соответствующие ни одному из шаблонов рис. 2.8, а-ж). Для порядка, состоящего из всех нулей, недопустим единичный старший бит мантиссы (рис. 2.9, а). Для  $p_{min} \leq p \leq p_{max}$  недопустим нулевой старший бит мантиссы (рис. 2.9, б). Для порядка, состоящего из всех единиц, недопустима ненулевая мантисса с нулевым старшим битом (рис. 2.9, в).

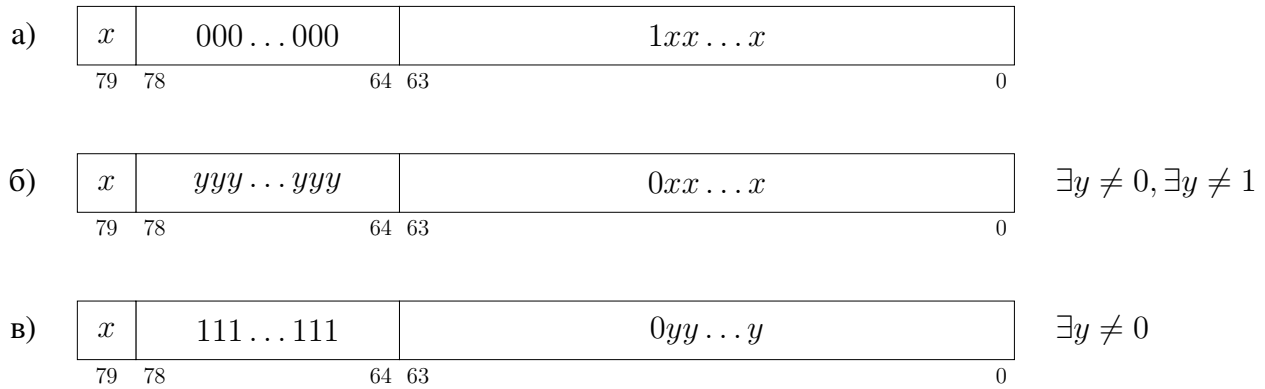


Рис. 2.9. Недопустимые значения в FPU x87:

а) с нулевым порядком, б) с порядком  $p_{min} \leq p \leq p_{max}$ ,

в) с порядком, состоящим из единиц

Возможен экспорт из описанного нестандартного представления в стандартные форматы одинарной и двойной точности.

### Точность двоичных чисел с плавающей запятой

Как и в случае фиксированной запятой, формат с плавающей точкой не может описать все вещественные значения в заданном диапазоне.

При сохранении вещественного числа  $X$  в формате с плавающей запятой оно округляется до ближайшего представимого числа  $\check{X}$ .

Если  $\check{X} = (-1)^s \cdot 2^p \cdot \mu$  нормализовано, то абсолютная погрешность округления не превышает веса младшего разряда мантииссы:

$$\left| X - \check{X} \right| \leq \frac{2^p}{2^{n+1}} = 2^{p-n-1}. \quad (2.91)$$

Эта величина зависит от порядка числа  $p$ , а также от количества бит в мантииссе  $n$  ( $n$  постоянно для формата), и превышает единицу при  $p > n + 1$ .

Таким образом, с ростом абсолютной величины  $X$  (и, соответственно, порядка  $p$ ) абсолютная погрешность округления катастрофически растёт.

Относительная погрешность округления до нормализованного  $\check{X}$  может быть оценена сверху как  $\frac{1}{2^n}$ :

$$\begin{aligned} \left| \frac{X - \check{X}}{X} \right| &\leq \frac{2^p}{2^{n+1} \cdot |X|} \approx \frac{2^p}{2^{n+1} \cdot |\check{X}|} = \frac{2^p}{2^{n+1} \cdot 2^p \cdot \mu} = \\ &= \frac{1}{2^{n+1} \cdot \mu} \leq \frac{1}{2^{n+1} \cdot 0,1_2} = \frac{1}{2^n}. \end{aligned} \quad (2.92)$$

Оценка на самом деле точная, так как хотя настоящая мантисса  $X$  не равна  $\mu$ , она также больше или равна  $0,1_2$ .

Если  $X$  невозможно представить в нормализованном виде, абсолютная погрешность округления не превышает  $\frac{2^{p_{min}}}{2^{n+1}} = 2^{p_{min}-n-1}$ , относительная погрешность будет расти при уменьшении абсолютной величины  $X$ , так как мантисса денормализованного числа может быть сколь угодно близка к нулю.

### Нормализованное представление в отечественной и зарубежной традиции

В зарубежных источниках (2.81) называется нормализованным в случае  $1 \leq \mu < 10_N$  [16]. Там считается, что мантисса нормализованного числа включает целую часть (хотя суть от этого не меняется). Действительно, пусть

$$X = (-1)^s \cdot N^p \cdot \mu, \quad s \in \{0, 1\}, \quad p \in \mathbb{Z}, \quad 0,1_N \leq \mu < 1, \quad (2.93)$$

то есть  $\mu = \overline{0, m_1 m_2 m_3 m_4 \dots}$ ,  $m_1 \neq 0$ . Пусть также

$$X = (-1)^{\tilde{s}} \cdot N^{\tilde{p}} \cdot \tilde{\mu}, \quad \tilde{s} \in \{0, 1\}, \quad \tilde{p} \in \mathbb{Z}, \quad 1 \leq \mu < 10_N \quad (2.94)$$

Тогда

$$\begin{cases} \tilde{s} &= s \\ \tilde{p} &= p - 1 \\ \tilde{\mu} &= N \cdot \mu = 10_N \cdot \mu = \overline{m_1 m_2 m_3 m_4 \dots} \end{cases} \quad (2.95)$$

Так как двоичная/десятичная запятая является нецифровым символом, она не может быть записана в память, а только *подразумевается* на той или иной позиции, двоичное представление мантиссы в формах (2.93) и (2.94) *полностью совпадает*.

Порядок  $\tilde{p}$  формы (2.94), соответственно, записывается с избытком  $\tilde{\xi} = \xi + 1$ . Одна из основных особенностей кода со смещением — невозможность «визуально» определить ноль, так что полученный код (смещённый порядок) также *полностью совпадает* для форм (2.93) и (2.94).

Таким образом, двоичное представление одинаковых чисел (как нормализуемых в формате расширенной точности, так и денормализованных) одинаково и не зависит от формы нормализованного представления.

В отдельных источниках мантисса и вовсе рассматривается как целое беззнаковое число  $\overline{m_1 m_2 m_3 m_4 \dots m_n}$  [80]. Такая трактовка также допустима и равносильна (2.93) и (2.94) при соответствующей коррекции порядка и смещения.

## Контрольные вопросы

1. Чем различаются качественные и количественные данные?
2. Какие числа называются натуральными?
3. Какие числа называются неотрицательными целыми?
4. Какие нецифровые символы используются в представлении чисел?
5. Какие способы представления беззнаковых целых чисел используются в ЭВМ?
6. Какие способы представления знаковых целых чисел используются в ЭВМ?
7. Какие логические и битовые операции вы знаете?
8. Какие способы представления вещественных чисел используются в ЭВМ?
9. Как выглядит нормализованное представление вещественного числа?

## Глава 3

# Архитектура команд семейства x86

Знающий сокровенное и явное, силён, мудр.

*Коран. 64.18*

Обозначением x86 описывают целый класс вычислительных систем, включающий уже практически не используемую шестнадцатибитную архитектуру, тридцатидвухбитную архитектуру IA-32 (i386–i686), шестидесятичетырехбитную x86-64 (amd64, Intel 64 или IA-32e).

Все модели этого многочисленного семейства совместимы между собой на уровне архитектуры команд, то есть все современные процессоры в определённом режиме могут выполнять программы, написанные для более старых.

В данной главе рассматриваются режимы работы x86-совместимых процессоров, доступные регистры, флаги, режимы адресации, а также структура команды и вытекающие из неё ограничения.

### 3.1. Развитие линейки x86 и режимы работы

В отношении деятельности опыт, по-видимому, ничем не отличается от искусства; мало того, мы видим, что имеющие опыт преуспевают больше, нежели те, кто обладает отвлечённым знанием, но не имеет опыта.

*Аристотель. Метафизика*

В данном пособии рассматривается тридцатидвухбитный и шестидесятичетырехбитный режимы работы, как более простые для прикладного программиро-

вания и более распространённые. Подробное описание качественно отличного от них шестнадцатибитного режима можно найти, в частности, у Питера Абеля [26].

Тем не менее, так как многие особенности архитектуры x86 обусловлены исторически и поддерживаются для совместимости, необходимо сделать краткий экскурс в историю данной линейки.

### 3.1.1. История семейства x86

Архитектура x86 основана на архитектуре четырёхразрядного микропроцессора Intel 4004. Так как микросхема 4004 была разработана для настольного калькулятора, в ней не были реализованы многие механизмы, давно и успешно применявшиеся в более ранних компьютерах, в частности, аппаратная трансляция адресов. Позже была выпущена улучшенная версия 4004 — процессор 4040, а на его основе был разработан восьмиразрядный 8008, включавший два восьмибитных регистра общего назначения  $a$  и  $b$ . В улучшенном 8080 их было уже семь —  $a, b, c, d, e, h, l$ , объединявшиеся в три пары  $bc, de, hl$ . Шина адреса в 8080 была шестнадцатиразрядной (то есть можно было адресовать до  $2^{16}$  байт, или 64 килобайта), адрес задавался парой регистров  $hl$ .

### Шестнадцатибитные процессоры

Процессор 8086 — родоначальник семейства x86 — был шестнадцатибитным, из-за чего 16 бит при программировании для x86 обычно называют **словом**. Он включал четыре шестнадцатибитных регистра общего назначения  $ax, bx, cx, dx$ , каждый из которых фактически был парой восьмибитных (в частности,  $ax = ah : al$ ), и четыре неделимых шестнадцатибитных регистра  $bp, sp, si, di$ . У каждого из них было и специальное назначение:  $A$  — accumulator (неявный аргумент большинства команд),  $C$  — counter (счётчик),  $D$  — data (данные),  $B$  — base (базовый регистр). В отличие от 32-разрядного режима, невозможно было использовать в косвенной адресации любые регистры. Для задания адреса в памяти использовались только три бита поля  $R/M$  и поле смещения (раздел 3.6.2). Базовыми



могли быть только *bx* и *bp* (base pointer), индексными — только *si* и *di* (source index и destination index). Масштабирование индекса не использовалось.

Шина адреса при этом была двадцатиразрядной. Для того, чтобы адресовать  $2^{20}$  байт (один мегабайт) памяти шестнадцатибитными адресами, была введена **сегментная модель памяти**. Полный адрес складывался из шестнадцатибитного адреса и значения специального сегментного регистра, умноженного на 16.

Область памяти, адресуемая с помощью одного сегментного регистра, называлась сегментом. Сегмент занимал  $2^{16}$  байт, то есть 64 килобайта; разные сегменты могли пересекаться или полностью совпадать. В 8086 было четыре сегментных регистра, соответственно в программе использовалось четыре сегмента:

- *cs* (code segment) — сегмент кода; значение регистра *cs* добавлялось к адресам команд;
- *ds* (data segment) — сегмент данных, его значение добавлялось к адресам статических переменных;
- *es* (extra segment) — дополнительный сегмент данных, иногда там располагалась куча;
- *ss* (stack segment) — сегмент стека, добавлялся к адресам в стеке.

Добавляемый сегментный регистр определялся процессором автоматически; для данных при необходимости можно было использовать префикс замены сегмента.

Хотя в настоящее время используется плоская модель памяти (сегментные регистры присутствуют, но содержат другую структуру данных — селектор сегмента) по традиции области адресного пространства, где располагаются код, данные, стек и т. д., часто называются сегментами.

Так как четырёх сегментов по 64 килобайта часто не хватало, программисту приходилось изменять значения сегментных регистров во время работы программы для доступа к различным областям памяти. Управлять сегментами приходилось вручную.

В 8086 всё ещё не было механизма трансляции адресов, так что прикладные программы использовали реальные физические адреса ОЗУ; из-за этого режим совместимости с моделью памяти 8086 в более поздних процессорах называется **реальным режимом**. При этом каждой программе реального режима была доступна вся память компьютера, что не позволяло реализовать полноценную многозадачность.

Для 8086 был разработан математический сопроцессор 8087, предназначенный для вычислений с плавающей запятой. Сопроцессор устанавливался в отдельный сокет на материнской плате. Начиная с этой модели, стали выпускаться урезанные варианты процессоров. Так, 8086 с восьмибитной шиной данных получил название 8088.

На основе **8088** был построен компьютер IBM PC, так что большинство последующих процессоров Intel (и неинтеловских x86-совместимых процессоров) совместимы с 8086 на уровне машинного кода. Теоретически любой современный персональный компьютер можно загрузить в специальном режиме совместимости и выполнить программу, написанную для 8086. Практически с этим возникнут трудности, в частности, из-за несоответствия временных характеристик.

Непосредственно следующая модель, 80186, отличалась от 8086 незначительно. В 80286 появилась частичная поддержка **защищённого режима**, когда память разных программ изолирована (защищена) друг от друга за счёт аппаратной трансляции адресов. Шина адреса была увеличена до 24 разрядов.

### **Тридцатидвухбитные и шестидесятичетырёхбитные процессоры**

Полноценная реализация защищённого режима появилась в тридцатидвухбитном процессоре 80386 (часто называемом просто 386). Так как разрядность процессора сравнялась с разрядностью шины адреса, в защищённом режиме 386 используется плоская модель памяти. Количество сегментных регистров возросло до шести. При этом сегментные регистры защищённого режима содержат не часть адреса, а селектор, кодирующий ссылку на запись в специальной таблице

дескрипторов, которая, в свою очередь, задаёт границы сегмента в плоском адресном пространстве и атрибуты защиты.

Современные операционные системы используют именно защищённый режим процессора (либо очень схожий с ним шестидесятичетырёхбитный режим), в котором прикладной программе недоступны многие функции реального. При этом, так как эти функции прозрачно обеспечиваются операционной системой, прикладное программирование заметно упрощено.

Тем не менее, из-за используемого программного обеспечения даже более поздние модели (до Pentium 4) постоянно или часть времени работали в реальном режиме, так что многие учебники ассемблера описывают его наравне с защищённым.

В целом разработка 386 — наиболее существенный шаг в развитии архитектуры семейства x86. В настоящее время «x86» обозначает, как правило, 386-совместимый процессор (такая архитектура обозначается i386 или IA-32). Дальнейшее развитие в основном сводилось к добавлению новых команд, наращиванию параллелизма и увеличению частоты.

В определённый момент четырёх гигабайт памяти, адресуемых 32-битным указателем в плоской модели, оказалось недостаточно. В первую очередь это проявилось на серверах и специализированных высокопроизводительных рабочих станциях. Разработчики ПО и аппаратного обеспечения не стали возрождать неудобную сегментную модель памяти, вместо этого начали продвигаться решения с 64-битными виртуальными адресами.

Результатом совместной разработки Intel и Hewlett Packard стала архитектура IA-64, схожая с суперкомпьютером Эльбрус и свободная от недостатков, унаследованных от калькулятора 4004 и шестнадцатибитного 8086. IA-64 несовместима с набором команд x86. Она не получила популярности в основном из-за недостаточного количества портированного под неё ПО и несовершенства компиляторов, а также дороговизны и некоторых конструктивных недорабо-

ток воплощавших её процессоров Itanium. В мае 2017 г. официально объявили о закрытии этой линейки.

Ведущий конкурент Intel, компания AMD, предложила расширение архитектуры IA-32, увеличивающее разрядность адресов до 64 бит и дающее возможность увеличить разрядность данных с помощью специального префикса *REX*. Именно это расширение, которое в настоящее время поддерживается и процессорами Intel — наиболее популярный способ увеличить адресуемую память персонального компьютера.

Разработанная компанией AMD шестидесятичетырехбитная архитектура x86-64 (также называемая amd64, IA-32e и Intel 64, но не IA-64) не слишком существенно отличается от тридцатидвухбитной x86.

### 3.1.2. Режимы работы процессора

Процессоры тридцатидвухбитной архитектуры x86 (IA-32) поддерживают четыре режима работы (рис. 3.1) [2, 16].

- шестнадцатибитный реальный режим (Real Mode);
- шестнадцатибитный режим виртуального 8086 (Virtual-8086 Mode), используемый тридцатидвухбитными операционными системами для запуска устаревших программ;
- тридцатидвухбитный защищённый режим (Protected Mode);
- режим системного управления (System Management Mode, SMM) — сверхпривилегированный режим, в котором обрабатываются скрытые от операционной системы события.

Процессоры семейства x86-64 добавляют к ним ещё два режима, в совокупности обозначаемые в документации [2] как Long Mode (на рис. 3.1 эти два режима объединены серым овалом вверху рисунка):

- тридцатидвухбитный режим совместимости;
- шестидесятичетырехбитный режим.

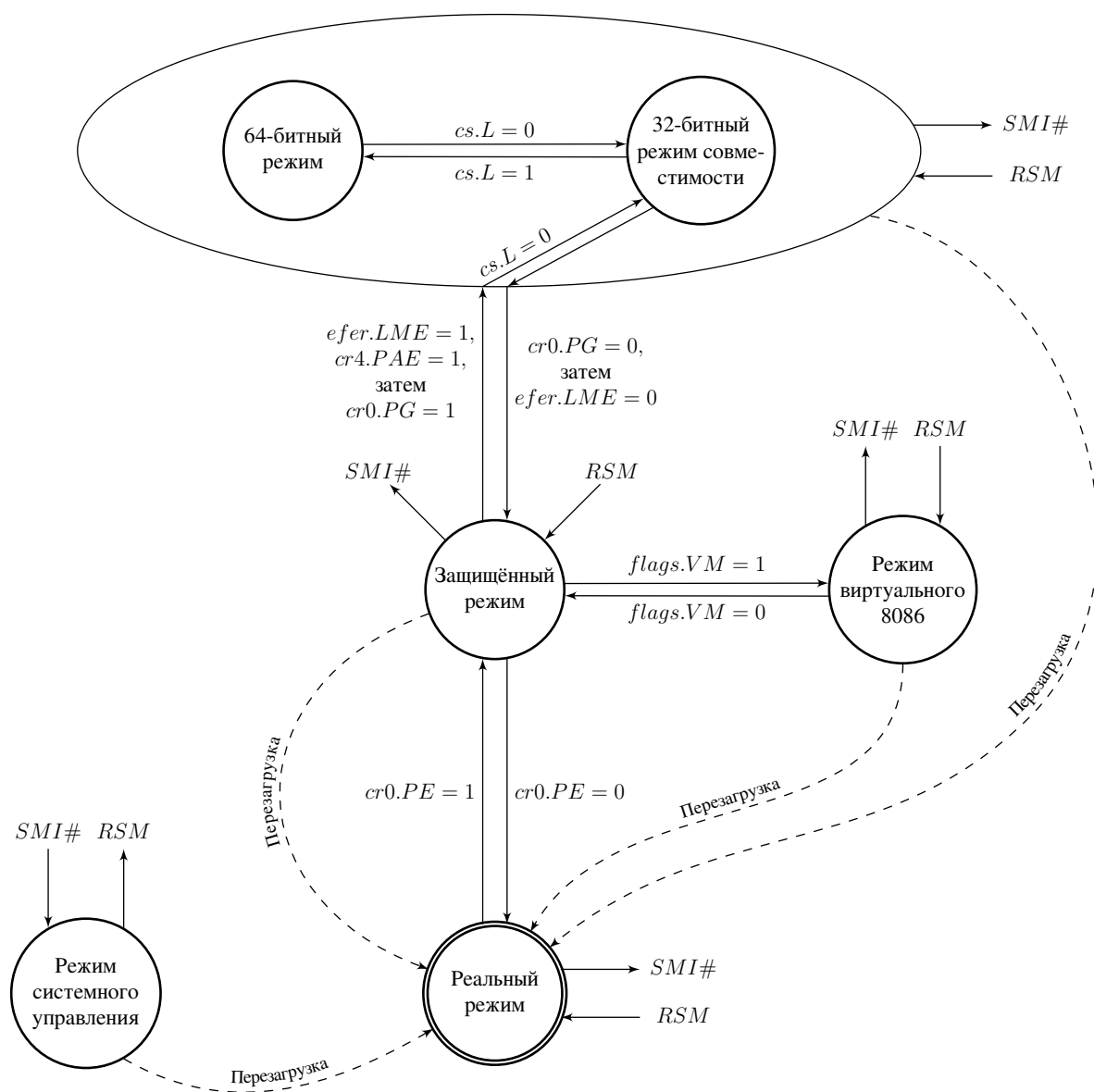


Рис. 3.1. Режимы работы современных процессоров

Шестидесятичетырёхбитный режим использует по умолчанию тридцатидвухбитные данные и смещения и концептуально отличается от тридцатидвухбитного режима гораздо меньше, чем шестнадцатибитный. Соответственно, его иногда называют шестидесятичетырёхбитным расширением защищённого режима, а архитектуру x86-64 обозначают как IA-32e.

### Порядок переключения режимов

При загрузке или сбросе процессор переходит в реальный режим. Из реального режима он может переключиться в защищённый, в котором работают все

тридцатидвухбитные операционные системы. При необходимости выполнения шестнадцатибитного кода процессор может временно переключаться в режим виртуального 8086.

Из тридцатидвухбитного защищённого режима процессор может переключиться в тридцатидвухбитный режим совместимости, а затем в шестидесятичетырёхбитный режим. Эти два режима используются шестидесятичетырёхбитными операционными системами. Таким образом, для выполнения тридцатидвухбитных приложений процессор может временно переключаться обратно в режим совместимости. Выполнить шестнадцатибитное приложение в шестидесятичетырёхбитной операционной системе невозможно — необходимо использовать эмулятор или перезагрузить компьютер под другой операционной системой.

На рис. 3.1 переключения режимов, возможные в процессе работы, показаны сплошными стрелками, переключения по перезагрузке — пунктиром.

Из любого из этих режимов процессор при возникновении соответствующего события может ненадолго перейти в режим системного управления, а после обработки события — вернуться в исходный режим.

В принципе, возможен переход из шестидесятичетырёхбитного режима через режим совместимости в защищённый и из защищённого в реальный, но, как правило, подобной необходимости не возникает и в современных операционных системах она не реализована. Прикладная программа, запущенная в защищённом режиме, не может перевести процессор в реальный (или, соответственно, из шестидесятичетырёхбитного в защищённый).

### 3.2. Сегменты памяти

*Пэ* — прямое движение точки, прочь от неподвижной,  
движение по прямой черте. Отсюда тела, полные пещер,  
рост объёма, занятого телом в трёхмерном мире: пух,  
порох, пушка.

Пара двух точек, разделённая растущим пространством.

*В. Хлебников. Царапина по небу*

В памяти вычислительной машины фон-неймановской архитектуры хранится как код программы, так и данные. Данные (переменные) в программе на языке высокого уровня, в частности, C++, делятся на:

- глобальные, время жизни которых равно времени жизни программы, а имя доступно в любой области программы;
- статические локальные, время жизни которых также равно времени жизни программы, но имя доступно только в ограниченной области;
- локальные — каждый вызов функции порождает новую копию переменной, время жизни которой не превышает времени работы функции и которая доступна только в ограниченной области;
- динамические — память выделяется и освобождается с помощью *new/delete* или *malloc()/free()*.

Различные виды переменных и программы находятся в разных областях диапазона доступных виртуальных адресов — адресного пространства процесса, исторически называемых **сегментами** (сейчас иногда используется также термин «**секция**», чтобы подчеркнуть использование плоской модели памяти и связь областей памяти с секциями исполняемого файла).

На рис. 3.2 представлено возможное распределение виртуальных адресов процесса в тридцатидвухбитной операционной системе GNU/Linux.

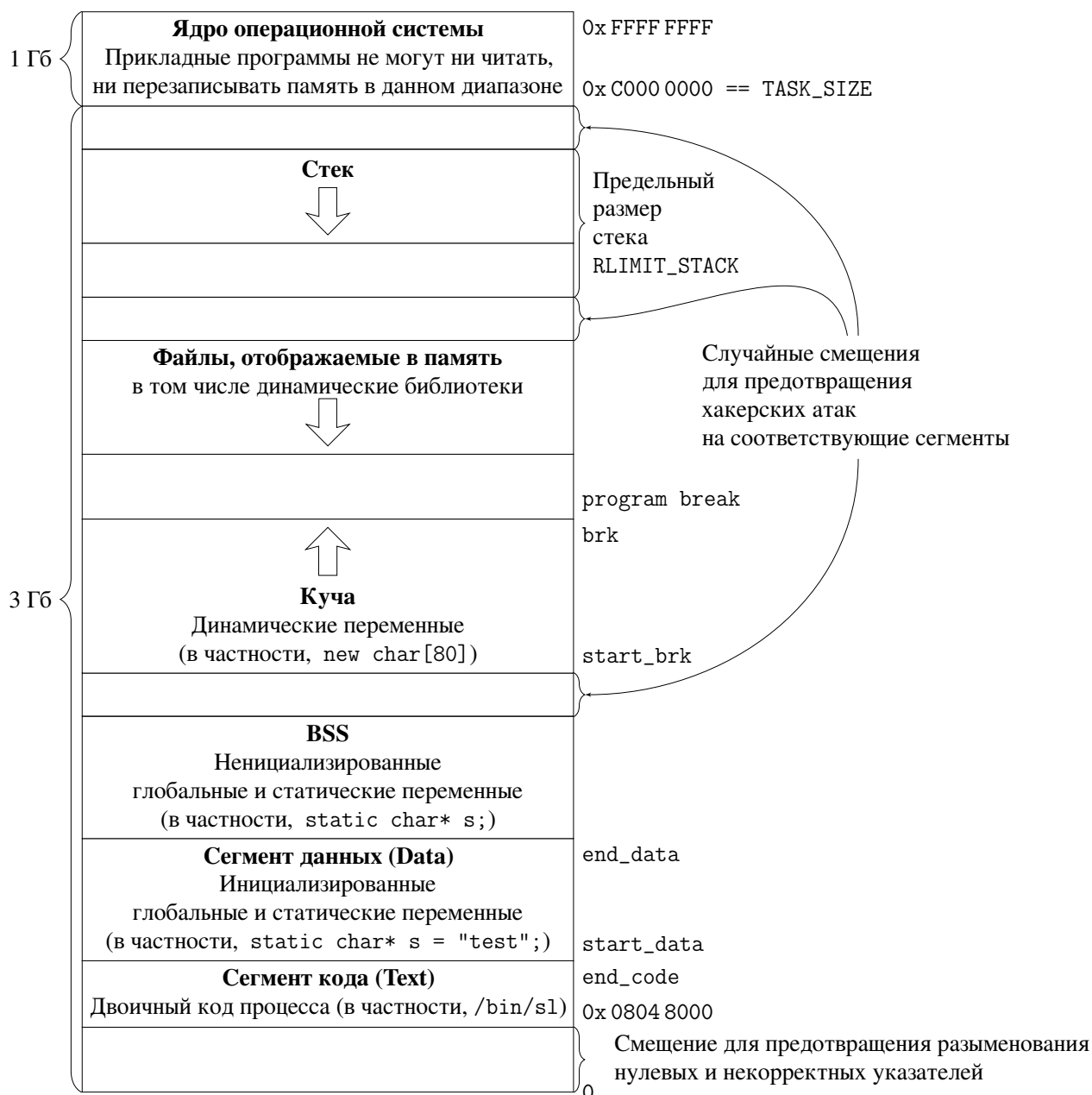


Рис. 3.2. Распределение памяти процесса в тридцатидвухбитной операционной системе GNU/Linux

Конкретное расположение сегментов в адресном пространстве процесса может различаться для различных операционных систем. Состав также может различаться, но основные сегменты — кода, данных, кучи и стека — присутствуют всегда.

Хотя все сегменты располагаются в одном адресном пространстве, они могут иметь разные атрибуты защиты. В частности, сегмент кода для предотвра-



щения вредоносных модификаций доступен только для чтения, но не для записи, а для сегментов, доступных программе на запись (данные, стек, куча) запрещено исполнение.

Нулевой адрес и ближайшие к нему считаются некорректными для выявления и предотвращения ошибок (разыменования переменных, которые указателями не являются).

Кроме того, между динамически растущими сегментами (данных и кучи, кучи и стека, стека и пространства ядра) добавляются «зазоры» случайного размера для затруднения атак на соответствующие области памяти. Размеры этих смещений определяются при загрузке программы в память.

Часть адресного пространства процесса (в тридцатидвухбитных системах по умолчанию 1 Гб, в шестидесятичетырехбитных — 512 Гб) занимает ядро операционной системы [68].

### 3.2.1. Код и статические данные

Код выполняемой программы находится в **сегменте кода**.

Глобальные переменные программы, доступные в любой её точке и статические переменные, отличающиеся от глобальных только областью видимости, расположены в **сегменте данных**. Те глобальные и статические переменные, которые не были инициализированы при объявлении, отделяются в специальный **сегмент BSS**.

Адреса глобальных и статических переменных в программе — неотрицательные целые константы. Для адресов в коде (в частности, функций) возможно задать как фиксированный адрес, так и смещение относительно текущего значения указателя команд *ip*.

Размеры кода программы и переменных, время жизни которых совпадает со временем жизни программы, могут быть определены ещё на этапе загрузки программы в память, поэтому размеры соответствующих областей памяти постоянны.

При этом адрес, по которому могут быть загружены код и статические данные, в принципе может варьироваться (это особенно актуально для разделяемых библиотек). В этом случае требуется каким-то образом сохранить работоспособность программы и доступность данных.

Для кода программы это достигается использованием относительных адресов (так называемых команд ближнего перехода, содержащих не сам адрес, а его смещение относительно текущего значения указателя команд *ip*).

Для данных в тридцатидвухбитном режиме адресация относительно *ip* невозможна. Соответственно, используются два варианта. Первый — коррекция фиксированных адресов в программе при загрузке исполняемого файла в память (это мешает совместному использованию библиотек, так как, если несколько программ попытаются загрузить библиотеку по разным адресам, получится разный код). Второй — копирование текущего значения *ip* в регистр общего назначения обходным путём и ручной расчёт смещений относительно полученного значения.

В шестидесятичетырёхбитном режиме добавлена возможность задавать для данных адрес в виде смещения относительно текущего значения *ip*. Для Mac OS X адресация глобальных и статических переменных относительно *ip* обязательна, для других операционных систем — рекомендуется.

### 3.2.2. Куча

Динамические переменные расположены в **сегменте динамической памяти**, или **куче** (heap). Первоначально программе выделяется определённый объём динамической памяти, из которого средствами языка высокого уровня (*new/new[]/malloc()* для C++) выделяются области памяти под запросы прикладной программы. Распределённые области помечаются в куче как занятые. Если в свободных областях кучи недостаточно памяти для обработки запроса, *new/new[]/malloc()* обращается к операционной системе для расширения кучи. Соответственно, количество корректных адресов сегмента кучи увеличивается.

Когда прикладной программе уже не нужна какая-то динамическая переменная, соответствующую область памяти необходимо освободить. Для этого в C++ используются операторы *delete/delete[]* или функция *free()*, помечающие область как свободную. При этом освобождать область памяти необходимо способом, соответствующим выделению. Таким образом, если память была выделена оператором *new[]*:

```
1 int *p_array = new int[N];
```

освобождать её нужно оператором *delete[]*:

```
1 delete[] p_array;
```

После вызова несоответствующего оператора *delete* *p\_array* будет помечен как свободный только первый элемент массива. В некоторых языках есть механизм автоматического сбора мусора, который освобождает те области памяти, к которым программа уже не обращается, но в C/C++ его нет.

Если память не освобождена, динамические переменные существуют до завершения программы, даже если адрес, по которому можно обратиться к ним, утрачен (подобная ситуация называется утечкой памяти). Таким образом, время жизни динамических переменных фактически определяется программистом.

### 3.2.3. Стек

Локальные переменные подпрограмм находятся в **сегменте стека**, также оптимизирующие компиляторы могут помещать часть целочисленных переменных в регистры общего назначения.

Стек назван так потому, что организован по принципу LIFO (last in, first out) — последним зашёл, первым вышел. Указателем вершины стека служит специальный регистр *sp* — stack pointer. Он содержит адрес начала последнего записанного в стек элемента. Соответственно, адреса локальных переменных в программе отсчитываются относительно вершины стека *sp*.

Команды семейства x86 могут записывать в стек элементы размером от шестнадцати бит, но в GNU/Linux стек **по соглашению** выравнен по *long*, то есть на 32 бита (рис. 3.3).

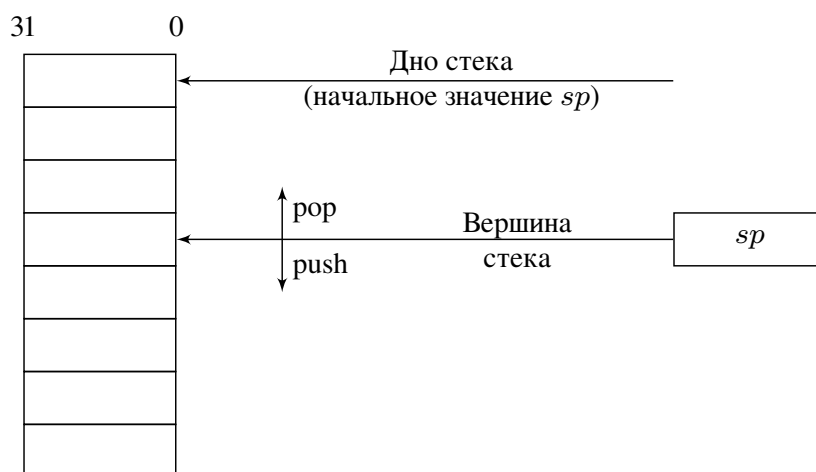


Рис. 3.3. Стек

Стек растёт вниз (в сторону уменьшения адресов). Таким образом, операция помещения элемента в стек (*push*) уменьшает указатель стека *sp*, операция извлечения (*pop*) — увеличивает. Таким образом, с учётом порядка байт Intel *sp* указывает на крайний (с наименьшим адресом) занятый байт стека.

В частности, рассмотрим рекурсивное вычисление факториала небольшого целого числа (листинг 3.1). Это крайне неэффективный способ вычисления, но

**Листинг 3.1.** Рекурсивный вызов функции

```

1 int fact(int n)
2 {
3     int f;
4     if (n <= 2)
5         f = n;
6     else
7         f = n * fact(n-1);
8     return f;
9 }

10 int main(int argc,
11     char *argv[])
12 {
13     int x;
14     x = fact(3);
15     cout << x;
16     return 0;
17 }

```

в учебниках он традиционно рассматривается как наглядный пример рекурсии.

После запуска программы стартовый код запускает головную функцию *main()*. Соответственно, в стеке находятся данные этой функции (рис. 3.4, а): локальная переменная *x*, адрес возврата, показывающий, какой команде будет

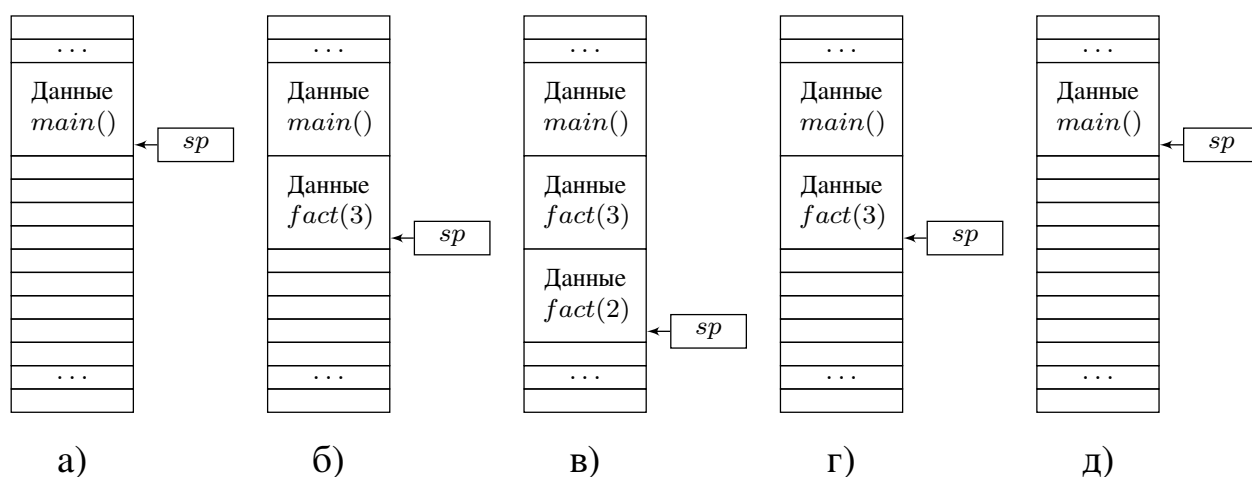


Рис. 3.4. Изменение указателя стека при вызове и возврате из функций

передано управление после возврата из *main()*, а также аргументы *main()* — количество параметров командной строки, переданных при запуске программы и указатель на массив этих параметров. В зависимости от используемого соглашения о вызове (подробнее в разделе 6.2.1), часть или все аргументы функции *main()* могут находиться в регистрах общего назначения; но в тридцатидвух-битном режиме они все передаются через стек.

Размер каждого из параметров, а также общий размер блока локальных переменных, по соглашению о выравнивании должны быть кратны 32 битам.

После вызова функции  $fact(3)$  в стек добавляется ещё один слой данных (рис. 3.4, б): параметр  $n = 3$ , адрес возврата из  $fact(3)$  (в данном случае это адрес команды, записывающей результат в переменную  $x$  в  $main()$ ) и локальная переменная  $f$ . После анализа  $n$  следует рекурсивный вызов  $fact(2)$ , добавляющий в стек новый параметр  $n = 2$ , новый адрес возврата (адрес команды умножения на  $n$  в  $fact(3)$ ) и ещё одна копия локальной переменной  $f$  (данные  $fact(2)$  на рис. 3.4, в). Таким образом, каждому вызову функции  $fact()$  соответствует свой набор параметров и локальных переменных.

После анализа параметра  $n$  следует возврат значения  $n$  из функции  $fact(2)$ . При этом управление передаётся команде по адресу возврата, а сам адрес возврата вместе с локальными переменными и параметрами этой функции удаляется из стека (рис. 3.4, г). Удаление выполняется путём изменения указателя  $sp$ , содержащее памяти при этом не перезаписывается (рис. 3.5, а) и б). Соответственно,

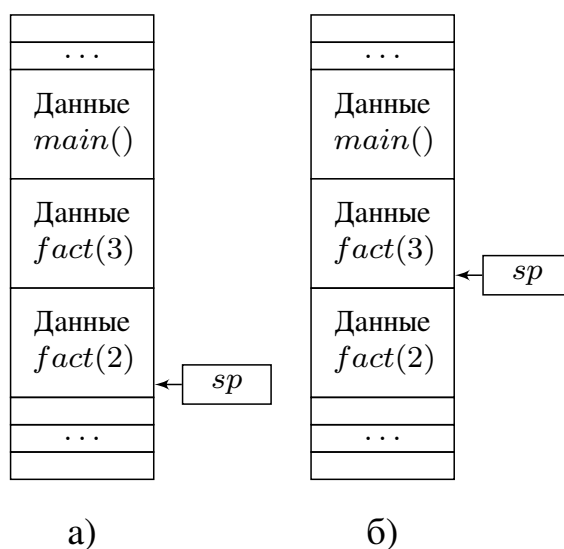


Рис. 3.5. Удаление данных из стека — изменение указателя

в незанятых ячейках стека содержатся «мусорные» данные, поэтому значение неинициализированных локальных переменных непредсказуемо.

После умножения результата  $fact(2)$  на  $n$  происходит возврат в  $main()$  (рис. 3.4, д).

Некоторые модели процессоров, в том числе ранние не-х86 совместимые процессоры Intel, организуют стек вызовов не в памяти, а в специальном наборе регистров. Это ограничивает количество вложенных вызовов функций, зато ускоряет процесс вызова и возврата.

### 3.3. Регистры

*Эр* — точка, прорезавшая, просёкшая поперечную площадь.

*Эр* — реет, рвёт, рассекает преграды, делает русла и рвы.

*В. Хлебников. Зангези*

Процессоры семейства х86 содержат как множество недоступных и ограниченно доступных программисту специальных регистров, так и определённое количество регистров общего назначения, которые можно адресовать на уровне архитектуры команд по номерам, а в программе на ассемблере — явно указанными в коде именами.

В частности, команда безусловного перехода `jmp label`, аналог оператора C++ `goto label`, модифицирует специальный регистр *ip* (указатель команд), но не содержит его имени. Напротив, команда загрузки значения в регистр общего назначения `mov $13, %eax` содержит его имя (*eax*) в явном виде, а соответствующий машинный код `B8 00 00 00 0D` содержит номер регистра *A* ( $B8_{16} = 10111000_2$ , последние три бита `000` задают регистр-приёмник; `00 00 00 0D` — загружаемое тридцатидвухбитное значение  $13_{10} = D_{16}$ ). Если это имя или номер заменить именем или номером другого регистра общего назначения, получим корректную команду загрузки значения в этот регистр.

Граница между специальными регистрами и регистрами общего назначения в наборе команд x86 достаточно размыта. Так как регистров в оригинальном процессоре 8086 было мало, все адресуемые регистры имели ещё и какое-либо специальное назначение. В частности, регистр *A* (*rax/eax/ax/al*) — регистр-аккумулятор. Он используется командами знакового расширения, деления и множеством других команд как неявный аргумент. Со временем специализация адресуемых регистров сглаживается, но многие команды, унаследованные от оригинального набора, обращаются к неявному аргументу в конкретном регистре общего назначения.

Основной набор команд x86 предназначен для обработки целых чисел, так что в регистрах процессора могут находиться целочисленные переменные (адреса, индексы и собственно целые числа). Регистры, доступные различным расширениям набора команд, в частности, команд обработки вещественных чисел, хранят данные соответствующего типа. Такие регистры, как правило, недоступны командам из основного набора и будут рассматриваться отдельно.

### 3.3.1. Регистры общего назначения, доступные в тридцатидвухбитном режиме

Под номер регистра в команде (её структура более подробно рассматривается в разделе 3.6.2) отведено всего три бита [17, 36], так что регистров общего назначения в тридцатидвухбитном режиме x86 доступно только восемь (рис. 3.6).

В некоторых источниках к регистрам общего назначения относят только четыре регистра — *A*, *B*, *C* и *D* (на рис. 3.6 показаны слева). Каждый из них конструктивно имеет размер машинного слова (сейчас, как правило, 64 бита), но в тридцатидвухбитном режиме доступны только младшие 32. Разные их части называются разными именами.

В частности, младший байт регистра *A* обозначается *al* (low), следующий байт — *ah* (high). Пара однобайтовых регистров *ah* : *al* составляет младшие 16 бит регистра — *ax* (для шестнадцатибитного 8086 это означало eXtended). Млад-



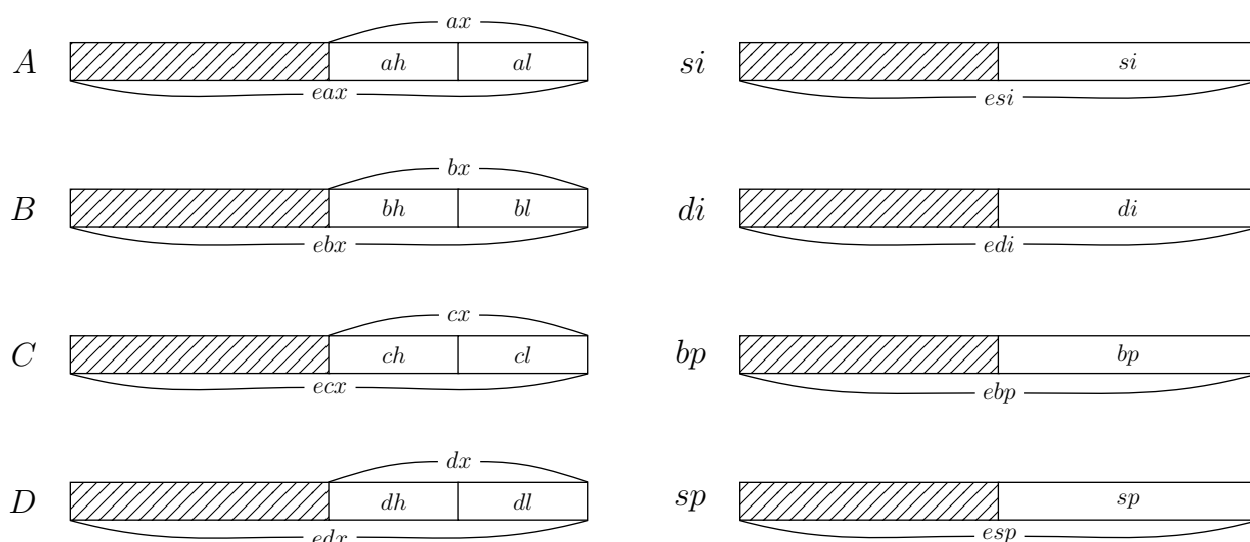


Рис. 3.6. Регистры общего назначения в тридцатидвухбитном режиме

шие 32 бита (максимально доступный в тридцатидвухбитном режиме размер регистра) обозначаются как *eax*, доступные только в шестидесятичетырехбитном режиме 64 бита — *rax*. Для краткости будем использовать однобуквенное обозначение регистра, когда его разрядность может быть любой или совпадает с разрядностью системы, в частности, *A* вместо *rax/eax/ax/al*.

Также имена и номера существуют для регистров *si*, *di*, *bp* и указателя вершины стека *sp*, которые иногда также причисляют к регистрам общего назначения (на рис. 3.6 справа). Эти имена соответствуют младшим 16 битам регистров. Их 32-битные варианты называются соответственно *esi*, *edi*, *ebp* и *esp*, 64-битные — *rsi*, *rdi*, *rbp* и *rsp*. Младшие байты этих регистров не имеют имён в тридцатидвухбитном режиме.

Для краткости будем использовать оригинальное имя регистра, когда его разрядность совпадает с разрядностью системы, например, *sp* вместо *rsp/esp/sp*. Это не вызовет путаницы, так как шестнадцатиразрядный код сейчас практически не используется.

Хотя *sp* можно адресовать как регистр общего назначения, использовать его иначе, чем как указатель вершины стека, категорически не рекомендуется. Кроме того, как будет показано в разделе 3.6.3, возможности адресации *sp* ограничены. Таким образом, будем считать регистрами общего назначения следующие семь — *A*, *B*, *C*, *D*, *si*, *di* и *bp*.

### 3.3.2. Регистры общего назначения, доступные

#### в шестидесятичетырехбитном режиме

В шестидесятичетырехбитном режиме доступны все описанные выше регистры. При этом для регистров общего назначения, доступных в тридцатидвухбитном режиме, используются те же имена. Для шестидесятичетырехбитных регистров имена соответствуют тридцатидвухбитным вариантам, но вместо префикса *e* используется префикс *r* (*rax*, *rdi* и т. д.).

Кроме того, в 64-разрядном режиме может быть использован специальный префикс *REX* (расширения регистров, подробнее в разделе 3.6.5), который добавляет ещё один бит к номерам регистров в команде, так что можно адресовать ещё восемь регистров общего назначения *r8*–*r15* (рис. 3.7). В 32-разрядном режиме они недоступны.

Младшие части регистров *r8*–*r15* имеют имена *r8b*–*r15b* (размер этих регистров равен одному байту — 8 бит), *r8w*–*r15w* (размер равен слову — 16 бит), *r8d*–*r15d* (размер равен двойному слову — 32 бита). Также префикс *REX* позволяет адресовать младшие байты регистров *si*, *di*, *bp* и *sp* — они имеют имена *sil*, *dil*, *bpl* и *spl* [16] и доступны наравне с *al*–*dl* и *r8b*–*r15b*.

Старшие байты младшего слова (разряды 8–15, аналогично *ah*–*dh*) не имеют собственных имён ни для каких регистров, кроме *A*–*D*. В некоторых источниках сказано, что *ah*–*dh* в шестидесятичетырехбитном режиме недоступны. Это не совсем так. Регистры *ah*–*dh* доступны, но только в командах без префикса *REX*.

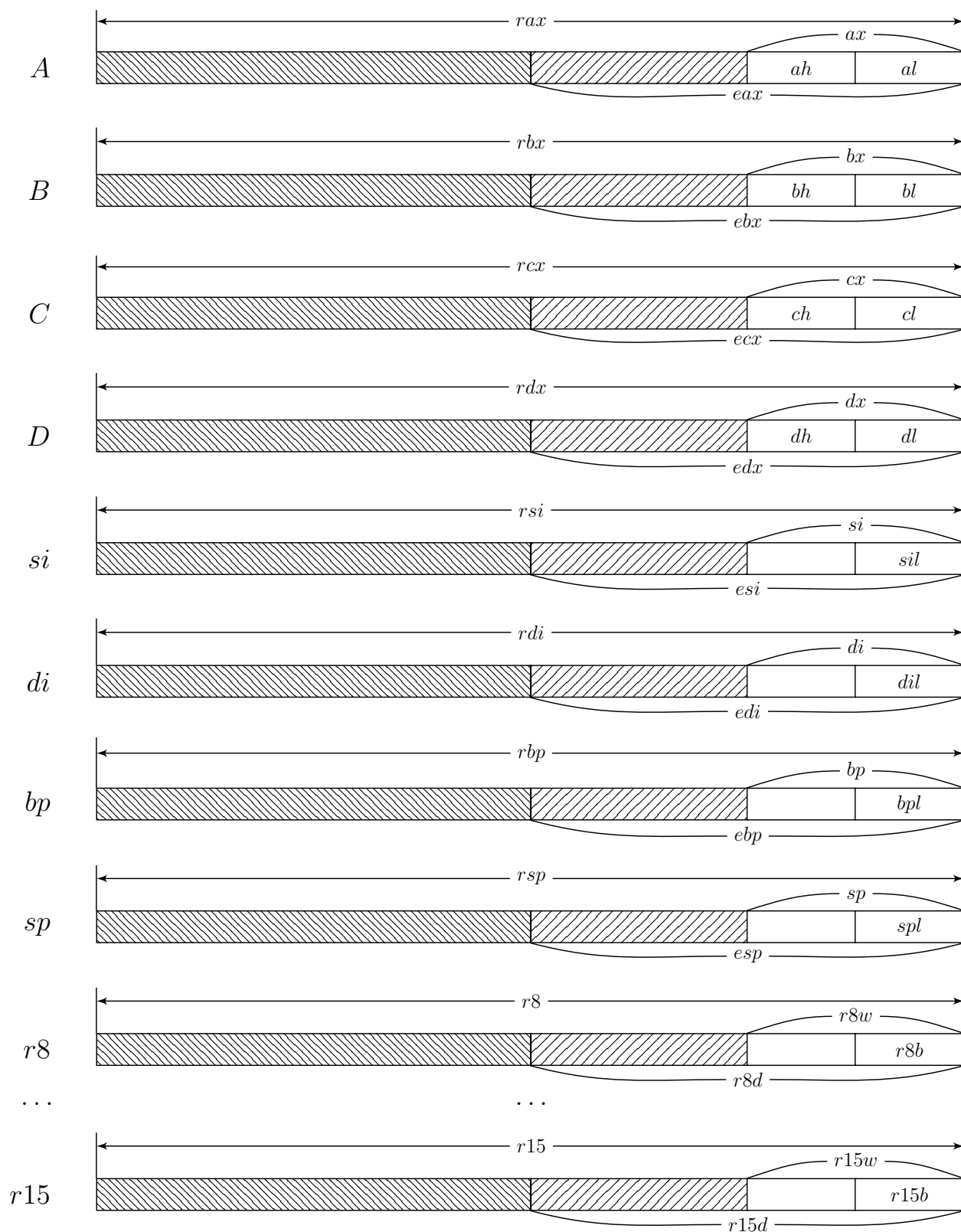


Рис. 3.7. Регистры общего назначения  
в шестидесятичетырехбитном режиме

### 3.3.3. Специальные регистры и регистры расширений

Из специальных регистров следует отметить регистры состояния и управления: уже упоминавшийся указатель команды *ip* (в 32- и 64-битных системах иногда называется *eip* и *rip* соответственно) и регистр флагов *flags* (*eflags/rflags*). Разряды регистра флагов либо показывают те или иные характеристики последней операции процессора (флаги состояния), либо влияют на выполнение команд (управляющие флаги).

Современные процессоры семейства x86, кроме основного набора команд, поддерживают несколько расширений. Для них реализованы несколько групп регистров общего назначения, доступных в командах соответствующих наборов:

- восемь 80-разрядных регистров FPU x87 ( $r_0 - r_7$ ) могут быть использованы как командами FPU как  $st(0) - st(7)$ , хранящие числа с плавающей запятой, так и командами расширения MMX как 64-разрядные регистры  $mm0 - mm7$ , ( $mm0 - mm7$  — мантиссы  $r_0 - r_7$ );
- восемь 128-разрядных регистров расширения SSE, или XMM ( $xmm0 - xmm7$ ). Каждый из них предназначен для хранения вектора вещественных чисел одинарной точности, а не длинного 128-битного числа. В 64-битных системах количество XMM-регистров, как и количество регистров общего назначения, увеличено до шестнадцати ( $xmm0 - xmm15$ ). Расширение AVX (YMM) вдвое увеличило их разрядность — до 256-разрядных  $ymm0 - ymm15$ , недавно появившееся AVX-512 (ZMM) — до 512-разрядных  $zmm0 - zmm31$  (в 32-битных системах доступны только первые восемь). При этом регистры  $ymm_i$  — младшие половины регистров  $zmm_i$ , а  $xmm_i$ , соответственно, — младшие половины  $ymm_i$ . Регистры ZMM есть не во всех современных процессорах.

Также расширения могут иметь свои специальные регистры, в частности, регистры флагов. Например, FPU имеет обширный набор специальных регистров, так

как изначально команды набора FPU выполнялись отдельным устройством — математическим сопроцессором.

### 3.4. Математический сопроцессор (FPU x87)

У вас свой вероустав, у меня свой вероустав.

*Коран. 109.6*

Математический сопроцессор (Floating Point Unit, FPU) — устройство для обработки числовых данных в формате с плавающей точкой. Первый математический сопроцессор для линейки x86 — FPU 8087 — был выпущен в 1980 году. Он представлял собой отдельную микросхему, устанавливаемую в специальный сокет на системной плате. Взаимодействие с основным процессором выполнялось в основном через оперативную память.

Начиная с процессора i486DX математический сопроцессор интегрирован в процессор. При этом сопроцессор долгое время (вплоть до линейки микропроцессоров Atom) имел почти независимое ядро, так что обработка целых чисел CPU и вещественных FPU могла выполняться параллельно. Из-за этого в системе команд была введена команда ожидания завершения работы сопроцессора, а многие команды управления сопроцессором реализованы в двух вариантах — с ожиданием и без. В современных процессорах FPU настолько плотно интегрирован с ядром CPU, что их параллельная работа невозможна и ожидание не требуется.

#### 3.4.1. Регистры FPU

FPU x87 предоставляет восемь 80-разрядных регистров для хранения данных и шесть вспомогательных регистров [5, 16]. При обращении к ним в GAS надо указывать тот же префикс %, что и для регистров основного процессора (CPU).



Рис. 3.8. Регистры FPU

Восемь регистров данных, согласно документации Intel [16], носят имена  $r_0 - r_7$ , но обратиться к ним по этим именам невозможно. Они образуют стек с плавающей вершиной, построенный по принципу кольцевого буфера. К регистру, находящемуся сейчас в вершине стека, можно обратиться как к  $st(0)$ ; если стек содержит более одного элемента, то к более глубоким элементам можно обращаться по именам  $st(1)$ ,  $st(2)$  и так далее до  $st(7)$  [33]. Регистры данных сопроцессора хранят вещественные числа в 80-битном расширенном формате. Мантисса занимает 64 бита, порядок — 15 бит, под знак отводится один бит.

Шестнадцатибитный регистр (слово) тегов  $tw$  (Tag Word, также используется сокращение  $twr$  — Tag Word Register) хранит состояние регистров данных. Каждому регистру  $r_0 - r_7$  соответствует два бита слова тегов (рис. 3.9):

- 00 — в соответствующем регистре корректное ненулевое значение;
- 01 — в регистре ноль;

- в) 10 — в регистре специальное значение: некорректное значение (*nan* или значение, не соответствующее формату вещественного числа с расширенной точностью), бесконечность или денормализованное число;
- г) 11 — регистр пуст.

<i>tag</i> (7)	<i>tag</i> (6)	<i>tag</i> (5)	<i>tag</i> (4)	<i>tag</i> (3)	<i>tag</i> (2)	<i>tag</i> (1)	<i>tag</i> (0)
15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

Рис. 3.9. Слово тегов FPU

Если регистр  $r_i$  помечен в слове тегов как пустой, его значение при этом может быть каким угодно — попытка чтения из него приведёт к ошибке стека.

Флаги математического сопроцессора разбиты на два шестнадцатитбитных регистра (рис. 3.10) — управляющие флаги составляют управляющее слово *cw* (Control Word, также *cwr*), флаги состояния сгруппированы в слово состояния — *sw* (Status Word, также *swr*).

Слово состояния <i>sw</i>													
<i>B</i>	<i>C3</i>	<i>top</i>	<i>C2</i>	<i>C1</i>	<i>C0</i>	<i>ES</i>	<i>SF</i>	<i>PE</i>	<i>UE</i>	<i>OE</i>	<i>ZE</i>	<i>DE</i>	<i>IE</i>
15 14	13	11	10	9	8	7	6	5	4	3	2	1	0

Управляющее слово <i>cw</i>													
		<i>RC</i>	<i>PC</i>			<i>PM</i>	<i>UM</i>	<i>OM</i>	<i>ZM</i>	<i>DM</i>	<i>IM</i>		
15	12	11	10	9	8	7	6	5	4	3	2	1	0

Рис. 3.10. Слово состояния и управляющее слово FPU

Управляющее слово содержит шесть масок исключений ( $IM-PM$ ), поле управления точностью  $PC$ , и поле управления округлением  $RC$ .

Слово состояния отображает текущее состояние сопроцессора после выполнения последней команды. Младший байт слова состояния включает семь флагов, показывающих корректность операций ( $IE-SF$ ) и флаг  $ES$ , показывающий, что сбой не только был, но и привёл к прерыванию. Старший байт включает

флаги  $C0$ – $C3$ , хранящие **признаки последней операции FPU (в частности, устаревшие команды сравнения чисел помещают в них результат сравнения)**, а также трёхбитный текущий номер вершины стека  $top$ . Последний бит  $B$  в настоящее время не используется.

Таким образом, стек сопроцессора организован с помощью восьми регистров данных  $r_0$  –  $r_7$ , соответствующих восьми полей слова тегов  $tag(0)$ – $tag(7)$  и поля  $top$  слова состояния. Вершина стека  $st(0)$  находится в регистре  $r_{top}$ , обозначение  $st(1)$  получает следующий регистр  $r_{top+1}$  и так далее. За  $r_7$  по принципу кольцевого буфера следует  $r_0$ . На рис. 3.11 показаны соотношения между физическими  $r_i$  и логическими  $st(i)$  именами регистров данных сопроцессора при различных значениях номера вершины стека  $top$ .

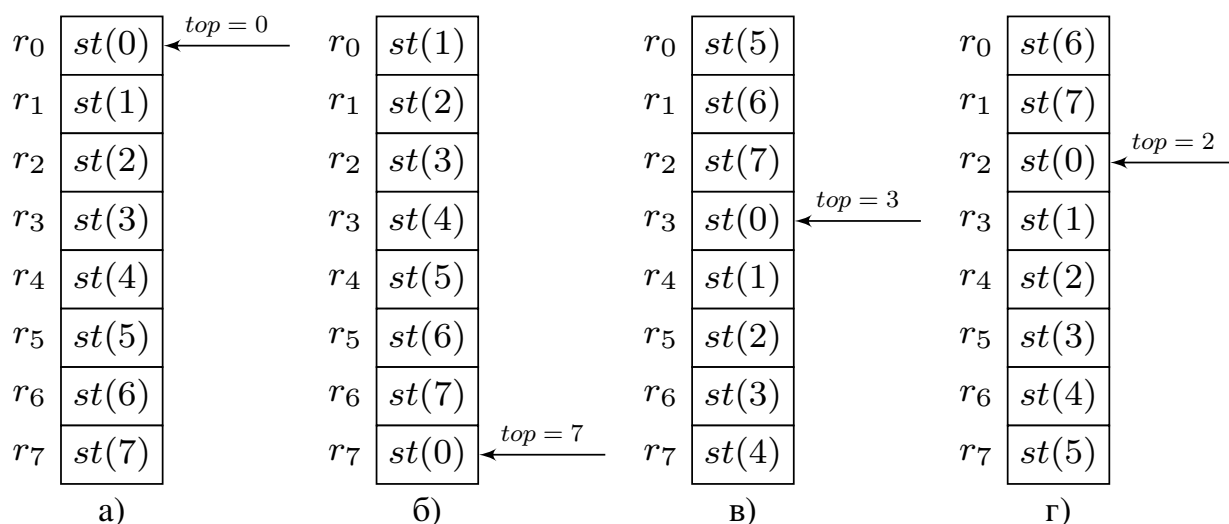


Рис. 3.11. Стек FPU

Положение дна стека определяется словом тегов  $tw$  (первый пустой регистр).

После инициализации стек пуст. После завершения вычислений (перед выходом из функции или ассемблерной вставки) его также необходимо оставить пустым. Если функция возвращает вещественное значение через стек сопроцессора, в стеке не должно остаться ничего, кроме возвращаемого значения.



Для вычислений хотя бы один операнд должен быть загружен в стек сопроцессора.

Два 48-битных регистра указателей (на последнюю команду — FPU Instruction Pointer, *fip*, в некоторых источниках также *ipr* [51] и последний загруженный операнд Data (Operand) Pointer, *fdp*, также *dpr*), а также десятибитный регистр кода операции последней неуправляющей команды (FPU Opcode Register, *fop*) используются в обработке исключений для определения места сбоя.

### 3.4.2. Исключения FPU

Во время работы сопроцессора возможны ситуации, когда по какой-то причине невозможно корректно выполнить требуемые вычисления. Подобные ситуации называются исключительными ситуациями, или просто исключениями FPU.

Рассмотрим исключения FPU подробнее. Любое из них приводит к общему исключению недействительной операции (**#I**).

**#I** — недействительная операция (Invalid operation). Может быть стековой ошибкой **#IS** или недопустимой арифметической операцией **#IA**.

#### Стековая ошибка

**#IS** — стековая ошибка (Stack Fault) — попытка записи в полностью заполненный стек или чтения из пустой ячейки стека FPU.

#### Недопустимые арифметические операции

Недействительной арифметической операцией (**#IA**) считается операция, проводимая над некорректными аргументами. В этом случае может также возникнуть одна из следующих пяти ситуаций.

**#D** — денормализованный операнд (Denormalized operand) — выполнение арифметической операции над денормализованным числом или загрузка такого числа в стек FPU.

**#Z** — деление на ноль (Zero Divide) — деление на ноль.

**#O** — переполнение порядка (Overflow) — порядок результата выходит за максимально допустимое значение.

Для команд выгрузки из стека `f*st` переполнение возможно в том случае, если размер порядка приёмника недостаточен.

**#U** — антипереполнение, или исчезновение порядка (Underflow) — порядок результата выходит за минимально допустимое значение (денормализованный результат).

Для команд выгрузки из стека `f*st` антипереполнение возможно в том случае, если выгружаемое значение слишком близко к нулю и не может быть корректно представлено в формате приёмника.

**#P** — неточный результат (Precision) — результат невозможно точно представить в формате назначения (например,  $\frac{1}{3}$ ,  $\sqrt{2}$ ).

Команды вычисления трансцендентных функций (`fsin`, `fcos`, `fsincos`, `fptan`, `fpatan`, `f2xm1`, `fyl2x`, `fyl2xp1`) всегда приводят к неточному результату.

## Маски исключений

Если в языках высокого уровня термин «исключение» подразумевает прерывание нормального хода программы и переход к обработчику, то FPU на некоторые (арифметические) исключения может реагировать двояко: помещать на место результата специальное значение (**вещественную неопределённость**) или инициировать прерывание вычислений.

Поведением FPU управляют шесть масок исключений ( $IM-PM$ ), расположенных в первых шести битах управляющего слова `sw`. На тех же местах в слове состояния `sw` располагаются соответствующие флаги  $IE-PE$ .

Если бит маски установлен в единицу, то соответствующее исключительная ситуация не вызывает прерывания выполнения программы (то есть того, что обычно и называется в языке высокого уровня исключением). Такое исключение называется замаскированным.

Стековую ошибку замаскировать невозможно.

### Флаги сопроцессора

Математический сопроцессор имеет собственный регистр флагов — слово состояния *sw*. Аналогично *flags*, биты слова состояния сопроцессора представляют те или иные характеристики последней операции сопроцессора [16, 71].

На рис. 3.10 показано расположение семи флагов ошибок разных видов, флага суммарной ошибки и флагов *C0–C3*, куда, в частности, помещают результат устаревшие команды сравнения. Команды вещественной арифметики не выставляют флаги *C0–C3* аналогично командам сравнения, но могут использовать эти биты иначе.

Первые семь битов слова состояния соответствуют исключениям FPU. Каждой исключительной ситуации соответствует свой флаг ошибки, который устанавливается в единицу при возникновении этой исключительной ситуации.

**IE (бит 0)** — флаг недействительной операции.

Устанавливается в единицу при выполнении недопустимой стековой (в этом случае устанавливается также флаг *SF*) или арифметической операции. В последнем случае могут быть установлены также флаги *DE*, *ZE*, *OE*, *UE* или *PE*.

**DE (бит 1)** — флаг денормализованного операнда.

**ZE (бит 2)** — флаг деления на ноль.

**OE (бит 3)** — флаг переполнения порядка.

**UE (бит 4)** — флаг антипереполнения, или исчезновения порядка.

**PE (бит 5)** — флаг неточного результата.

**SF (бит 6)** — флаг стековой ошибки.

Также по результатам операции выставляется флаг суммарной ошибки, которому не соответствует ни одно из исключений.

**ES (бит 7)** — флаг суммарной ошибки (Error Summary Status). Он равен единице, если возникает хотя бы одно незамаскированное исключение.

В некоторых источниках говорится, что  $ES$  равен единице в том случае, когда в разрядах 0...6 есть хотя бы одна единица [27]. Это в общем случае неверно. Если какое-то исключение замаскировано,  $ES$  не дублирует состояние соответствующего флага.

В частности, в C++ деление на ноль не должно приводить к прерыванию работы программы, поэтому соответствующее исключение при настройке сопроцессора стартовым кодом маскируется.

Соответственно, при попытке деления единицы на ноль, как можно убедиться при помощи отладчика, результат принимает специальное значение  $inf$  ( $+\infty$ ), устанавливается флаг  $ZE$ , но флаг  $ES$  не устанавливается.

### 3.5. Флаги

Мечтой увенчанный язык  
Плохой товарищ, где нет чисел,  
К числа жезлу наш ум привык.

*В. Хлебников. Двух юных слышу разговор...*

Во время выполнения многих команд формируется не только результат в виде числа, но и те или иные признаки результата (в частности, корректен ли он) — флаги состояния. Флаг занимает один бит и считается установленным, когда он равен 1, и сброшенным, когда равен 0.

В частности, как было сказано в разделе 2.5, при сложении и вычитании целых беззнаковых чисел ограниченной разрядности может образоваться бит переноса/заёма из старшего разряда, который сохраняется процессором в особой ячейке — флаге переноса  $CF$ . При сложении и вычитании знаковых чисел формируется флаг переполнения  $OF$ . Как  $CF$ , так и  $OF$  являются флагами состояния.

Аналогично флагам состояния, однобитовые переменные, не отражающие признаков результата последней операции, но влияющие на выполнение некоторых команд, называются управляющими флагами. Некоторые флаги состояния или управляющие флаги доступны только операционной системе и, соответственно, называются системными.

Флаги и некоторые системные переменные часто объединяются в специальный регистр — регистр флагов. Процессоры семейства x86 исторически имеют два регистра флагов — собственно регистр флагов *flags*, связанный с командами основного набора и слово состояния FPU *sw*, связанное с командами математического сопроцессора FPU.

### 3.5.1. Флаги основного процессора

Процессоры семейства x86 объединяют ячейку *CF* и подобные ей биты, показывающие те или иные свойства последней целочисленной арифметической операции — флаги состояния — в специальный регистр флагов *flags* (таблица 3.1).

Кроме флагов состояния, регистр флагов включает один бит, не отражающий выполнение последней операции, но влияющий на выполнение некоторых команд (управляющий флаг направления *DF*), а также несколько битов, недоступных прикладным программам (системные флаги) [18, 71, 78]. Часть битов зарезервирована и не используется сейчас как флаги (зарезервированный бит может иметь как произвольное, так и фиксированное значение).

Доступные прикладным программам флаги состояния в основном сосредоточены в младших восьми разрядах *flags*, поэтому многие команды сохранения/восстановления регистра флагов оперируют только с младшим байтом. Старшие восемь бит содержат один флаг состояния *OF*, управляющий флаг *DF* и несколько системных. В тридцатидвухбитном регистре *eflags* в старших шестнадцати битах добавлено ещё шесть системных флагов; старшие тридцать два бита шестидесятичетырехбитного *rflags* не используются.

Регистр флагов *flags*

Таблица 3.1

flags				
0	CF	Carry Flag	Флаг переноса (беззнакового переполнения)	Состояние
1	1	—	Зарезервирован	
2	PF	Parity Flag	Флаг чётности	Состояние
3	0	—	Зарезервирован	
4	AF	Auxiliary Carry Flag	Вспомогательный флаг переноса	Состояние
5	0	—	Зарезервирован	
6	ZF	Zero Flag	Флаг нуля	Состояние
7	SF	Sign Flag	Флаг знака	Состояние
8	TF	Trap Flag	Флаг трассировки	Системный
9	IF	Interrupt Enable Flag	Флаг разрешения прерываний	Системный
10	DF	Direction Flag	Флаг направления	Управляющий
11	OF	Overflow Flag	Флаг знакового переполнения	Состояние
12–13	IOPL	I/O Privilege Level	Уровень приоритета ввода-вывода	Системный
14	NT	Nested Task	Флаг вложенности задач	Системный
15	0	—	Зарезервирован	
eflags				
16	RF	Resume Flag	Флаг возобновления	Системный
17	VM	Virtual-8086 Mode	Режим виртуального процессора 8086	Системный
18	AC	Alignment Check	Проверка выравнивания	Системный
19	VIF	Virtual Interrupt Flag	Виртуальный флаг разрешения прерывания	Системный
20	VIP	Virtual Interrupt Pending	Ожидающее виртуальное прерывание	Системный
21	ID	ID Flag	Проверка на доступность инструкции CPUID	Системный
22–31		—	Зарезервированы	

Регистр *flags* не может быть явно указан как операнд команды, но является неявным результатом большинства арифметических команд и неявным операндом условных команд.

**Флаги состояния**

Флаги состояния отображают результаты целочисленных арифметических операций (сложения и вычитания; ограниченно умножения и поразрядных логи-

ческих операций и пр.); этими флагами являются биты 0, 2, 4, 6, 7 и 11 регистра *flags*.

### **CF (бит 0)**

Флаг переноса (Carry Flag = CF). Устанавливается, если происходит перенос из старшего разряда результата за пределы разрядной сетки при сложении или заём в старший разряд из несуществующего (выходящего за пределы операнда, воображаемого) разряда при вычитании, таким образом, этот флаг показывает переполнение при выполнении беззнаковых арифметических операций.

Флаг *CF* часто используется и для других целей, тогда его значение не связано с беззнаковым переполнением. Так, этот бит используется командами сдвига — именно в него выдвигается «лишний» бит, командами извлечения бита — для хранения извлечённого значения и многими другими.

### **PF (бит 2)**

Флаг чётности (Parity Flag = PF). Устанавливается, если младший байт результата команды содержит чётное число единиц, иначе — сбрасывается.

Флаг чётности использовался для подсчёта контрольных сумм.

### **AF (бит 4)**

Флаг коррекции (Adjust Flag = AF). Устанавливается, если арифметическая операция производит перенос (заём) из младшей тетрады младшего байта, т. е. из бита 3 в старшую тетраду при сложении (вычитании). Используется только для двоично-десятичной (BCD — Binary-Coded Decimal) арифметики, которая оперирует исключительно младшими байтами.

### **ZF (бит 6)**

Флаг нуля (Zero Flag = ZF). Устанавливается, если результат операции — нуль, иначе — сбрасывается.

**SF (бит 7)**

Флаг знака (Sign Flag = SF). Всегда равен значению старшего бита результата. Этот бит интерпретируется как знаковый в некоторых арифметических операциях (0/1 — число положительное/отрицательное).

**OF (бит 11)**

Флаг переполнения (Overflow Flag = OF). Устанавливается, если при знаковой интерпретации результат операции не помещается в операнд (слишком большое положительное или слишком маленькое для отрицательных знаковых чисел); иначе — сбрасывается. При сложении этот флаг устанавливается в 1, если происходит перенос в старший бит и нет переноса из старшего бита (то есть сумма положительных чисел даёт результат, интерпретируемый как отрицательный), или имеется перенос из старшего бита, но отсутствует перенос в него (сумма отрицательных чисел положительна); в противном случае, флаг *OF* устанавливается в 0. При вычитании он устанавливается в 1, когда возникает заём из старшего бита, но заём в старший бит отсутствует, либо имеется заём в старший бит, но отсутствует заём из него.

Флаг переполнения сигнализирует о потере старшего бита результата в связи с переполнением разрядной сетки при работе со знаковыми числами.

Этот флаг используется командами знаковой целочисленной арифметики.

**Знаковые и беззнаковые команды**

Флаги состояния используются командами целочисленной арифметики трёх типов — знаковой, беззнаковой и (в тридцатидвухбитном режиме) двоично-десятичной BCD, а также командами условного перехода (ветвления) и условного присваивания.

При переполнении индикатором является:

- для знаковой арифметики — флаг *OF*,
- для беззнаковой арифметики — флаг *CF*.



Так как беззнаковые и представленные в дополнительном коде знаковые числа складываются с помощью одного и того же сумматора и одной командой, этот сумматор на всякий случай формирует при сложении и вычитании и  $OF$ , и  $CF$ , и остальные флаги состояния. Выбор для анализа того флага, который соответствует реальному типу операндов — ответственность программиста.

### 3.5.2. Флаги FPU

Математический сопроцессор FPU включает собственный регистр флагов — слово состояния  $sw$  (раздел 3.4.2). Старший байт  $sw$  может быть загружен в младший байт флагов основного процессора (таблица 3.2).

#### Загрузка состояния FPU в регистр флагов

Таблица 3.2

sw			
0	IE	Недействительная операция	
1	DE	Денормализованный операнд	
2	ZE	Деление на ноль	
3	OE	Переполнение	
4	UE	Антипереполнение	
5	PE	Неточный результат	
6	SF	Стековая ошибка	
7	ES	Бит суммарной ошибки	
flags, fnstsw + sahf			
8	C0	0	CF   Carry Flag
9	C1	игнорируется	
10	C2	2	PF   Parity Flag
11	TOP	Указатель вершины стека сопроцессора	игнорируется
12			4   AF   Auxiliary Carry Flag
13			игнорируется
14	C3	6	ZF   Zero Flag
15	B	7	SF   Sign Flag

Биты  $C0$ ,  $C2$ ,  $C3$  помещаются соответственно во флаги  $CF$ ,  $PF$ ,  $ZF$ ; бит  $C1$  игнорируется. Другие флаги младшего байта  $flags$  получают фактически неопределённое значение.

## 3.6. Структура команды и методы адресации

Это не костыль — это уже экзоскелет.

*Программистский фольклор*

Набор команд процессоров семейства x86 имеет тип CISC. Он создавался в условиях жёсткой экономии памяти, так что команды имеют максимально компактную (и, соответственно, сложную для декодирования и понимания) структуру и используют разнообразные методы адресации операндов.

Необходимость программирования в машинных кодах возникает крайне редко, но знание формата команды помогает лучше представлять себе возможности и ограничения архитектуры.

### 3.6.1. Методы адресации

Адрес операнда в машинной команде может быть задан одним из следующих способов [53]. Они называются методами адресации операндов.

1. **Неявная** адресация. Местоположение операнда фиксировано и определяется кодом операции.
2. **Непосредственная** адресация. Операнд — константа, которая включается непосредственно в команду.
3. **Прямая абсолютная** (прямая) адресация. Операнд — переменная в памяти по фиксированному адресу (глобальная или статическая). Этот адрес включается непосредственно в команду.
4. **Прямая относительная** адресация. В команде содержится смещение, которое прибавляется к значению указателя команд *ip*.

Этот режим используется в командах передачи управления и позволяет загружать код в память по произвольному адресу без нарушения корректности переходов.

Шестидесятичетырёхбитное расширение позволяет использовать прямую относительную адресацию также и для данных.

5. **Регистровая** адресация. Операнд находится в регистре общего назначения. В команду включается номер этого регистра.
6. **Косвенно-регистровая** (косвенная) адресация. Операнд — переменная в памяти и её адрес находится в регистре (регистрах) общего назначения.

Иногда выделяют следующие виды косвенной адресации:

- базовая адресация — адрес операнда в регистре (базовом регистре, базе);
- базовая адресация со смещением — адрес операнда вычисляется как сумма базового регистра и константы (смещения);
- базово-индексная адресация — адрес операнда вычисляется как сумма двух регистров — базового регистра и индексного;
- базово-индексная адресация со смещением — адрес операнда вычисляется как сумма двух регистров и константы.

Если необходимо обработать значение, адрес которого получается более сложным образом (в частности, переменная в памяти, указатель на которую также находится в памяти), этот адрес надо вычислить отдельно и поместить в регистр.

### 3.6.2. Структура команды

Команды процессоров семейства x86 имеют переменную длину. Структура команды показана на рис. 3.12.

Все поля, кроме кода операции, необязательны [17, 36].

Команда может предваряться одним или несколькими префиксами, изменяющими её поведение. Из префиксов x86 следует отметить префикс изменения размера операнда 0x66 и префикс изменения размера адреса 0x67. Для тридцатидвухбитного режима (и его шестидесятичетырёхбитного расширения) они уменьшают разрядность операнда или адреса соответственно до 16 бит, для

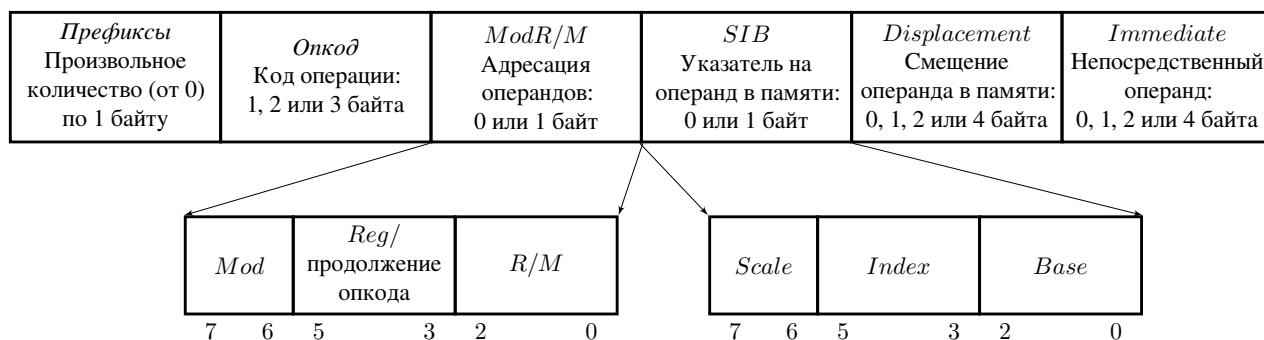


Рис. 3.12. Структура команды в архитектуре x86

шестнадцатибитного — повышают до 32. Восемьбитные варианты команд, как правило, представлены отдельными опкодами.

Далее идёт код операции (опкод), занимающий один, два или три байта (и, может быть, ещё три бита в байте *Mod R/M*). Следующий байт, *Mod R/M*, согласно [17], задаёт адресацию операндов. За ним следует необязательный байт *SIB*, уточняющий расположение операнда в памяти, если такой есть и для него используется косвенная базово-индексная адресация. Каждый из байтов *Mod R/M* и *SIB* состоит из трёх полей.

Поле *Displacement* содержит смещение адреса при косвенной адресации. Оно трактуется как знаковое тридцатидвухбитное (при использовании понижающего разрядность адреса префикса 0x67 — шестнадцатибитное) или восьмибитное число.

Поле *Immediate* — непосредственное значение («магическое число», включённое непосредственно в команду). Оно присутствует, если опкод соответствует операции с непосредственным операндом.

### 3.6.3. Общие для тридцатидвухбитного и шестидесятичетырёхбитного режимов сведения об адресации

Адресация в системе команд x86 частично задаётся опкодом. Это относится не только к неявной адресации (что логично), но и к непосредственной. Более того, для самых компактных команд номер операнда-регистра также может

быть указан внутри опкода. Так, тот вариант команды `mov`, который записывает непосредственное значение в регистр, состоит из однобайтового опкода и поля *Immediate*; при этом собственно код операции занимает пять старших бит опкода, а последние три бита задают регистр-приёмник.

При этом, если опкодом задаётся использование **только одного операнда или** двух операндов, один из которых непосредственный, а второй адресуется полем *R/M*, то в освободившееся поле *Reg* байта *Mod R/M* может быть записана часть опкода. **В частности, однобайтовый опкод FF, требующий поля *ModR/M* (в зависимости от значений *Mod* и *R/M* также возможны *SIB* и *Displacement*) соответствует, в зависимости от поля *Reg*, четырём различным однооперандным командам: `inc`, `dec`, `call` и `push`. Таким образом, полный код операции в этом случае занимает не восемь, а одиннадцать бит и записывается в двух полях.**

Кроме непосредственного, команда может иметь два операнда, расположение которых задаётся байтом *Mod R/M* и при необходимости уточняется байтом *SIB* и полем *Displacement*.

Поле *Reg* байта *Mod R/M* содержит трёхбитовый номер операнда-регистра. Является ли он источником или приёмником, определяет специальный бит опкода — поле направления.

Поле *R/M* в зависимости от поля *Mod* может как содержать номер второго операнда-регистра, так и определять адрес операнда в памяти. При необходимости совместно с полями *R/M* и *Mod* используются также *SIB* и *Displacement* (таблица 3.3).

При *Mod* = 11 поле *R/M* — номер второго операнда-регистра. Остальные три возможных значения *Mod* соответствуют косвенной адресации, то есть операнд находится в памяти по некоторому адресу.

Адресация операнда при помощи полей *Mod* и *R/M*

Таблица 3.3

<i>Mod</i>	<i>Displacement</i>	<i>R/M</i>	<i>SIB</i>	Операнд
11	—	<i>Reg2</i>	—	<i>Reg2</i>
01	<i>Disp8</i>	$Base \neq 100$	—	$*(Base + Disp8)$
		100	$(Scale, Index, Base)$ 100	$*(Base + 2^{Scale} \cdot Index + Disp8)$
		100	$(Scale, 100, Base)$	$*(Base + Disp8)$
10	<i>Disp32</i>	$Base \neq 100$	—	$*(Base + Disp32)$
		100	$(Scale, Index, Base)$ 100	$*(Base + 2^{Scale} \cdot Index + Disp32)$
		100	$(Scale, 100, Base)$	$*(Base + Disp32)$
00	—	$Base \notin \{100, 101\}$	—	$*(Base)$
		100	$(Scale, Index, Base)$ 100 101	$*(Base + 2^{Scale} \cdot Index)$
	<i>Disp32</i>	100	$(Scale, Index, 101)$ 100	$*(2^{Scale} \cdot Index + Disp32)$
		100	$(Scale, 100, 101)$	$*(Disp32)$
		101	—	зависит от разрядности режима
				32-битный: $*(Disp32)$ 64-битный: $*(rip + Disp32)$

Адрес операнда в памяти в общем случае может включать до четырёх компонент:

$$\text{Адрес} = Base + 2^{Scale} \cdot Index + Displacement \quad (3.1)$$

где масштаб  $Scale \in [0, 3]$  — константа,  $Base$  и  $Index$  — регистры.

Каждая из компонент может быть опущена; таким образом, прямая адресация в x86 является частным случаем косвенной.

Значения поля *Mod* 01 и 10 показывают наличие поля *Displacement* различной длины (при  $Mod = 01$  — восьмибитное, при  $Mod = 10$  — тридцатидвух-

битное), содержащего смещение. При  $Mod = 00$  смещение может отсутствовать или быть тридцатидвухбитным в зависимости от значения других полей.

Байт  $SIB$  следует за байтом  $Mod R/M$ , если поле  $R/M$  принимает специальное значение 100. Первые два его бита — масштаб — целое число  $Scale \in [0, 3]$ , определяющее коэффициент, на который умножается индексный регистр (соответственно 1, 2, 4 или 8), далее следуют трёхбитные поля  $Index$  и  $Base$ , хранящие номера регистров. Значение поля  $Index = 100$  является специальным, в этом случае считается  $Index = 0$  (то есть получается базовая адресация без индекса). Таким образом, индексный регистр в (3.1) не может иметь код 100 ни при каких значениях прочих полей.

Если значение поля  $R/M$  отлично от 100, байт  $SIB$  отсутствует.

При  $Mod = 01$  или  $Mod = 10$  любое значение  $R/M \neq 100$  трактуется как номер базового регистра  $Base$  в базовой адресации со смещением, то есть адрес операнда в памяти равен  $Base + Displacement$ , где  $Base$  не может иметь код 100. При  $R/M = 100$  используется базово-индексная адресация со смещением  $Base + 2^{Scale} \cdot Index + Displacement$ , где  $(Scale, Index, Base) \in SIB$ . Используя специальное значение  $Index = 100$ , можно получить базовую адресацию со смещением; в этом случае базовый регистр  $Base \in SIB$  может быть любым, включая  $Base = 100$ .

Режим  $Mod = 00$  — самый запутанный. По умолчанию используется косвенная адресация без смещения (поле  $Displacement$  отсутствует). При этом значение базового регистра (безразлично, записанного в поле  $R/M$  или в поле  $SIB$ )  $Base = 101$  является специальным и соответствует отсутствию в (3.1) базового регистра ( $Base = 0$ ) и одновременно — присутствию в команде тридцатидвухбитного смещения. Таким образом, базовый регистр в адресации без смещения не может иметь код 101 (но обратиться к подобному адресу можно — например, используя  $Mod = 01$ , где 101 является допустимым номером базового регистра, и указав восьмибитное смещение, равное нулю).

В шестидесятичетырехбитном режиме случай  $R/M = Base = 101$  при  $Mod = 00$  трактуется как особый случай и соответствует адресации относительно адреса следующей команды (специального регистра *rip* — указателя команд), то есть адрес операнда в памяти равен  $rip + Displacement$  [3]. В тридцатидвухбитном режиме аналога этой адресации нет.

В шестидесятичетырехбитном режиме разрядность номеров регистров может быть увеличена с помощью префиксов расширения регистра. Разрядность поля *Displacement* в шестидесятичетырехбитном режиме не повышается до 64 бит.

Таким образом, всего у команды x86 может быть не более трёх явных операндов, среди которых не более одного непосредственного значения, не более одного значения в памяти и не более двух регистров, причём два операнда-регистра исключают операнд в памяти.

Адрес операнда в памяти складывается из четырёх компонент по базово-индексной схеме со смещением, причём базовым может быть любой регистр общего назначения (в шестидесятичетырехбитном режиме — ещё и указатель команд, но только в базовой адресации без индекса), индексным не может быть регистр с кодом 100.

#### 3.6.4. Адресация в тридцатидвухбитном режиме

В тридцатидвухбитном режиме номера операндов-регистров задаются только трёхбитовыми полями байтов *Mod R/M* и *SIB*. Каждому трёхбитному номеру соответствует определённый регистр общего назначения с учётом типа и разрядности команды (таблица 3.4).

В зависимости от разрядности команды и от того, входит ли команда в основной набор инструкций или в какое-либо из расширений, один и тот же номер адресует разные регистры. Из таблицы 3.4 видно, что в тридцатидвухбитном режиме младшие байты регистров *sp*, *bp*, *si*, *di* не могут иметь имён, потому что соответствующие номера уже заняты *ah*—*dh*.



## Номера (коды) регистров

Таблица 3.4

Регистр					Код
32 бита	16 бит	8 бит	Команда MMX	Команда XMM	
<i>eax</i>	<i>ax</i>	<i>al</i>	<i>mm0</i>	<i>xmm0</i>	000
<i>ecx</i>	<i>cx</i>	<i>cl</i>	<i>mm1</i>	<i>xmm1</i>	001
<i>edx</i>	<i>dx</i>	<i>dl</i>	<i>mm2</i>	<i>xmm2</i>	010
<i>ebx</i>	<i>bx</i>	<i>bl</i>	<i>mm3</i>	<i>xmm3</i>	011
<i>esp</i>	<i>sp</i>	<i>ah</i>	<i>mm4</i>	<i>xmm4</i>	100
<i>ebp</i>	<i>bp</i>	<i>ch</i>	<i>mm5</i>	<i>xmm5</i>	101
<i>esi</i>	<i>si</i>	<i>dh</i>	<i>mm6</i>	<i>xmm6</i>	110
<i>edi</i>	<i>di</i>	<i>bh</i>	<i>mm7</i>	<i>xmm7</i>	111

В качестве компонент адреса операнда можно без ограничений использовать регистры *A, B, C, D, si, di, bp* необходимой разрядности. Указатель стека *sp* может быть базовым, что широко используется при адресации элементов стека, но не индексным, так как его код совпадает со специальным значением 100 поля *Index*.

### 3.6.5. Расширение регистров в шестидесятичетырехбитном режиме

В шестидесятичетырехбитном режиме между историческими префиксами и опкодом может находиться специальный однобайтовый префикс расширения регистров *REX* (рис. 3.13). Его младшие три бита (биты 0-2, обозначаемые в документации как *B, X, R*) используются для увеличения разрядности номеров регистров, следующий (третий) бит *W* определяет, используются ли шестидесятичетырехбитные операнды ( $REX.W = 1$ ) или разрядность операндов не изменяется ( $REX.W = 0$ ). Старшие четыре бита равны 0100 и служат для идентификации. Таким образом, префиксы *REX* — байты в диапазоне 40 — 4F. Соответственно, команды с опкодами 40 — 4F (это однобайтовые формы ко-

манд `inc` и `dec`) в шестидесятичетырехбитном режиме недоступны (при этом доступны двухбайтовые формы `inc` и `dec` с полем *ModR/M*, так что изменения в программы, написанные на ассемблере, вносить не нужно).

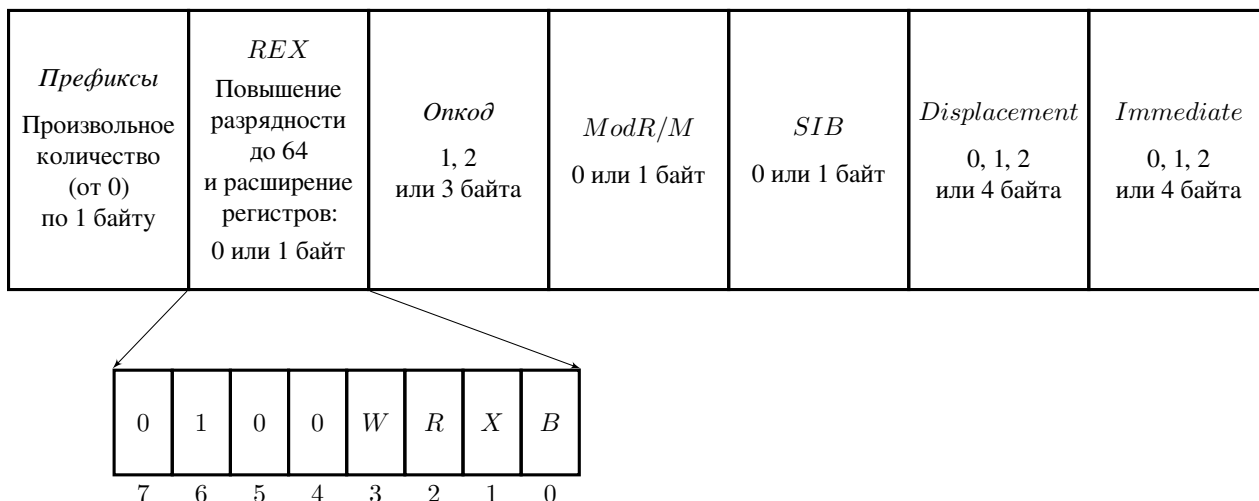


Рис. 3.13. Префикс расширения регистров *REX* в структуре команды x86-64

Бит 2 префикса *REX* (*REX.R*) добавляется к полю *Reg* байта *Mod R/M*, бит 1 (*REX.X*) — к полю *Index* байта *SIB*, младший бит 0 (*REX.B*), в зависимости от используемой адресации — к полю *R/M* байта *Mod R/M*, полю *Base* байта *SIB* или номеру регистра внутри опкода.

Таким образом, количество адресуемых регистров общего назначения возрастает до шестнадцати. Номера регистров *A—di*, указанные в таблице 3.4, соответствуют отсутствию префикса *REX* или нулю в соответствующем бите *REX* (в последнем случае с учётом *REX* получаем четырехбитовые коды 0100—0111). Единица в *REX* в сочетании с трёхбитовым полем в команде даёт номера регистров 1000—1111, то есть *r8—r15* (таблица 3.5).

Изменяя бит 3 префикса *REX* (*REX.W*), можно получить разрядность от 8 до 64 как для *r8—r15*, так и для *A—di*. При использовании восьмибитной команды в сочетании с *REX* с нулевым третьим битом (то есть разрядность не изменяется и операнды занимают 8 бит) коды регистров 0100—0111 описыва-

**Номера (коды) регистров общего назначения при  
использовании *REX***

Таблица 3.5

Регистр				Код	Регистр				Код
64	32	16	8 бит		64	32	16	8 бит	
бита	бита	бит			бита	бита	бит		
<i>rax</i>	<i>eax</i>	<i>ax</i>	<i>al</i>	0000	<i>r8</i>	<i>r8d</i>	<i>r8w</i>	<i>r8b</i>	1000
<i>rcx</i>	<i>ecx</i>	<i>cx</i>	<i>cl</i>	0001	<i>r9</i>	<i>r9d</i>	<i>r9w</i>	<i>r9b</i>	1001
<i>rdx</i>	<i>edx</i>	<i>dx</i>	<i>dl</i>	0010	<i>r10</i>	<i>r10d</i>	<i>r10w</i>	<i>r10b</i>	1010
<i>rbx</i>	<i>ebx</i>	<i>bx</i>	<i>bl</i>	0011	<i>r11</i>	<i>r11d</i>	<i>r11w</i>	<i>r11b</i>	1011
<i>rsp</i>	<i>esp</i>	<i>sp</i>	<i>spl</i>	0100	<i>r12</i>	<i>r12d</i>	<i>r12w</i>	<i>r12b</i>	1100
<i>rbp</i>	<i>ebp</i>	<i>bp</i>	<i>bpl</i>	0101	<i>r13</i>	<i>r13d</i>	<i>r13w</i>	<i>r13b</i>	1101
<i>rsi</i>	<i>esi</i>	<i>si</i>	<i>sil</i>	0110	<i>r14</i>	<i>r14d</i>	<i>r14w</i>	<i>r14b</i>	1110
<i>rdi</i>	<i>edi</i>	<i>di</i>	<i>dil</i>	0111	<i>r15</i>	<i>r15d</i>	<i>r15w</i>	<i>r15b</i>	1111

ют младшие байты регистров *sp–di* — в документации и коде программы они обозначаются *spl–dil*. Таким образом, в шестидесятичетырехбитном режиме доступны как *ah–dh*, так и *spl–dil*. При этом в одной и той же команде нельзя адресовать, например, *ah* и *sil*, так как для первого требуется обязательное отсутствие у команды префикса *REX*, а для второго — наличие *REX* с нулевым третьим битом.

*REX* — не единственный префикс расширения регистров. В частности, команды расширения AVX (YMM) Intel используют двух- или трёхбайтовый префикс *VEX*, одной из функций которого также является расширение номеров регистров *ymm*; расширение AVX512 (ZMM) — префикс *EVEX*. В расширении SSE5 (XOP) AMD также используется похожий, но при этом несовместимый с *VEX* префикс [4].

### 3.6.6. Операнды и адресация в шестидесятичетырехбитном режиме

Шестидесятичетырехбитный режим процессоров семейства x86 является расширением тридцатидвухбитного и отличается от него менее существенно, чем тридцатидвухбитный защищённый режим от шестнадцатибитного реального. В частности, структура команды аналогична описанной ранее. Режимы адресации также в основном сохраняются. Основным новшеством шестидесятичетырехбитной адресации является возможность адресовать данные относительно указателя команд.

Количество доступных регистров общего назначения в шестидесятичетырехбитном режиме увеличено за счёт возможности использования префикса *REX*, хранящего дополнительный бит номера регистра. При этом при формировании адреса операнда в памяти вначале определяются его компоненты, для чего трёхбитовые поля байтов *Mod R/M* и *SIB* проверяются на соответствие специальным значениям, а затем уже рассматривается префикс *REX* для определения номеров используемых регистров. Таким образом, в шестидесятичетырехбитном режиме индексным не может быть ни один регистр, младшие три бита номера которого равны специальному значению 100. Это указатель стека *sp* с номером 0100 и *r12*, обозначаемый номером 1100.

#### Разрядность операндов в регистрах и памяти

Для большинства команд в шестидесятичетырехбитном режиме размер операнда по умолчанию — 32 бита. Разрядность операндов может быть понижена до 16 бит соответствующим префиксом или повышена до 64 префиксом *REX* с единичным третьим битом *REX.W* [1, 17].

В шестидесятичетырехбитном режиме есть две группы команд, которые по умолчанию (без указания префикса *REX*) используют шестидесятичетырехбитные операнды: это команды, работающие с указателями (условные и безусловные переходы) и команды, неявно использующие указатель стека (по умолчанию это *rsp*). Команды вызова и возврата из функций относятся к обеим группам, так

что они по умолчанию используют шестидесятичетырёхбитные адреса и шестидесятичетырёхбитный указатель стека *rsp*. Разрядность операндов этих команд может быть понижена с 64 бит до 16 указанием соответствующего префикса, но способа понизить её до 32 бит не существует [3].

### Разрядность непосредственных операндов и смещения

Размер полей *Immediate* (кроме команды пересылки непосредственного значения в регистр) и *Displacement* в шестидесятичетырёхбитном режиме не повышается до 64 бит даже при использовании префикса *REX*.

Таким образом, непосредственные операнды занимают 1, 2 или 4 байта. Если непосредственный операнд (обычно тридцатидвухбитный) присутствует в шестидесятичетырёхбитной команде, он расширяется до 64 бит во время исполнения команды.

Единственная команда, включающая непосредственный операнд размером 8 байт — команда пересылки непосредственного операнда в регистр в сочетании с единичным *REX.W* (что повышает разрядность регистра-приёмника до 64). В документации Intel эта форма команды обозначается мнемоникой *mov*, как и другие варианты пересылки значения. Синтаксис AT&T выделяет форму с шестидесятичетырёхбитным непосредственным операндом как мнемонику *movabs*.

#### 3.6.7. Адресация относительно указателя команды

В тридцатидвухбитном режиме, как указано в таблице 3.3, существует два способа прямой адресации (указания на фиксированный адрес *Displacement* в памяти):

- без использования байта *SIB*, при  $R/M = 101$ ;
- с использованием *SIB* со специальными значениями как поля базы, так и поля индекса, то есть при  $R/M = 100$ ,  $Base = 101$ ,  $Index = 100$ .

Оба они доступны при  $Mod = 00$ , так что адрес *Displacement* тридцатидвухразряден.

В шестидесятичетырехбитном режиме к фиксированному адресу можно обратиться, только указав специальные значения байта *SIB*, а при  $R/M = 101$  используется адресация относительно счётчика команд *ip* (в шестидесятичетырехбитном режиме — *rip*) [3]:

$$\text{Адрес} = \begin{cases} ip + Displacement, & R/M = 101 \\ Displacement, & R/M = 100, Base = 101, Index = 100 \end{cases} \quad (3.2)$$

Таким образом, хотя шестидесятичетырехбитный указатель команд *rip* по-прежнему не является регистром общего назначения и не имеет собственного номера, он может быть базовым при  $Mod = 00$  без байта *SIB*.

Адресация относительно *ip* используется для решения двух задач:

- получение кода библиотек, работоспособность которого не зависит от адреса библиотеки в памяти (в тридцатидвухбитном режиме эта задача решалась либо вручную, либо корректировкой фиксированных адресов при загрузке [50]);
- обращение к переменным по произвольному шестидесятичетырехразрядному адресу (поле смещения в шестидесятичетырехбитном режиме осталось тридцатидвухбитным [61], так что прямая адресация позволяет обратиться только к адресам в пределах младших четырех гигабайт).

## Контрольные вопросы

1. Какие вы знаете режимы работы процессора?
2. Какие вы знаете сегменты памяти?
3. Чем различается размещение в памяти локальных, глобальных и статических переменных?
4. Какие вы знаете регистры общего назначения x86?
5. Какие регистры используются в FPU для хранения вещественных данных?

6. Какие вы знаете флаги?

7. Какие методы адресации вы знаете?

## Глава 4

### Связь уровней абстракции

Ток, соединивший числовой и словесный смысл пословиц, смысл, падающий с уровня слова на уровень числового мира, даёт ей игру и делал её забавной в своё время.

*В. Хлебников. Одиночество*

Современная вычислительная система включает шесть уровней абстракции, но только нижние четыре из них интерпретируются вычислительной машиной или операционной системой. Для трансляции программы с языка высокого уровня, в частности, C++, или даже с языка ассемблера в исполняемый файл, пригодный для запуска операционной системой, необходимы специализированные программные средства.

Более того, подобные средства должны обеспечивать не только прямую вертикальную связь соседних уровней, то есть компиляцию с языка на язык, но и «диагональные» связи — сборку исполняемого файла из множества фрагментов (модулей, функций), возможно, написанных на различных языках.

Универсальный способ объединения функций, написанных на различных языках — использование промежуточного (так называемого объектного) представления с последующей компоновкой модулей, содержащих эти функции, в единое целое. Так, в частности, из программы, написанной на языке Паскаль, можно вызвать функцию, описанную на C++, если они будут следовать одному соглашению о вызове.

Для соединения языка высокого уровня и языка ассемблера также используется механизм ассемблерных вставок в код. С помощью специальной конструкции ЯВУ можно вставить в код несколько команд ассемблера, не используя вызова и возврата из функции.



## 4.1. Компиляция

Являясь уже продуктом, сам первоначальный сырой материал должен, однако, пройти ещё целый ряд различных процессов, в которых он в постоянно изменяющемся виде каждый раз снова функционирует как сырой материал вплоть до последнего процесса труда, из которого он выходит уже как готовое жизненное средство или готовое средство труда.

*К. Маркс. Капитал*

В простейшем случае вертикальная связь уровней абстракции осуществляется через компиляцию — перевод программы с языка более высокого уровня на язык следующего уровня абстракции.

### 4.1.1. Инструменты разработки

В настоящее время разработка программного обеспечения невозможна без вспомогательных программ — инструментов разработчика. Для перевода программы с языка высокого уровня, в частности, C++, или даже с языка ассемблера, в машинный код, необходима программа-компилятор, для получения исполняемого файла — компоновщик и т. д.

#### Компиляторы C++

Для платформы x86 разработано множество компиляторов C++. Большинство из них входит в состав той или иной коллекции. В подобные коллекции входят компиляторы с языков C и C++, часто ассемблер, а также компилятор с Фортрана и других ЯВУ. Некоторые коллекции включают, кроме компиляторов, и другие средства разработки, иногда даже специализированные отладчик и IDE.

Перечислим наиболее известные коллекции.

- **GNU Compiler Collection (GCC)** реализована более чем для 45 платформ, поддерживает большинство ОС (порт под Microsoft Windows исторически носит название **MinGW**), семь языков (в том числе C и C++, со строгим соблюдением стандартов), и распространяется под лицензией GNU GPL 3+, позволяющей как образовательную, так и коммерческую разработку (в том числе разработку приложений с закрытым исходным кодом).
- **TenDRA/Ten15** — первоначально британская оборонная разработка, сейчас поддерживает архитектуры x86, x86-64, IA-64 (Itanium), DEC Alpha; POSIX-совместимые ОС и распространяется под лицензией BSD, позволяющей любые виды разработки.
- **Portable C Compiler (PCC)** — ранний компилятор C, какое-то время поддерживавшийся в OpenBSD, в настоящее время x86, x86-64, Unix-подобные ОС, в том числе GNU/Linux, Microsoft Windows; лицензия BSD.
- **Intel C++ compiler** — x86, x86-64, IA-64, GNU/Linux, MacOS X, Microsoft Windows, коммерческая собственническая лицензия.
- **Oracle Solaris Studio** — x86, x86-64, SPARC; Solaris, OpenSolaris, GNU/Linux; собственническая лицензия. В настоящее время распространяется бесплатно.
- **Open Watcom** — DOS, OS/2 и Microsoft Windows; лицензия Sybase Open Watcom Public License version 1.0, неполная поддержка стандарта.
- **Microsoft Visual Studio** — x86, x86-64, IA-64 и .NET, только Microsoft Windows, коммерческая собственническая лицензия, грубые нарушения стандарта.

Коллекция GCC портирована на наибольшее количество платформ. В её состав входят, в частности, **gcc** — компилятор C и **g++** — компилятор C++, а также ассемблер **gas**.

## Ассемблеры x86

Если синтаксис языка высокого уровня описан соответствующим стандартом, то синтаксис ассемблера не стандартизирован. Хотя набор мнемонических обозначения ассемблера определяется набором команд процессора, их символьное представление может существенно различаться. Кроме того, практически у каждого ассемблера уникальный набор директив и других синтаксических элементов, которые не транслируются непосредственно в машинные команды, но необходимы для корректной сборки программы.

Таким образом, фактически каждый ассемблер, предназначенный для архитектуры x86, обладает уникальным синтаксисом, несовместимым с остальными. При этом их можно разбить на две большие группы: синтаксис AT&T, традиционно используемый в Unix, но реализованный и для других операционных систем, и множество диалектов синтаксиса Intel.

- **GNU Assembler (GAS)** из коллекции GCC (лицензия GNU GPL 3+) используется на одном из этапов компиляции, поэтому реализован для всех поддерживаемых платформ. GAS использует для всех процессоров единый синтаксис (так называемый синтаксис AT&T).

Другие трансляторы поддерживают только архитектуру x86 и её шестидесятичетырёхбитный вариант x86-64.

- **Flat Assembler (FASM)** реализован для Unix-подобных операционных систем (GNU/Linux, OpenBSD и др.), FreeDOS и Microsoft Windows, распространяется по лицензии BSD. Несколько операционных систем написаны полностью на FASM — MenuetOS и KolibriOS.
- **NASM/Yasm** также реализован для Unix-подобных систем, FreeDOS и Microsoft Windows под лицензией BSD.
- **Turbo Assembler (TASM)/Lazy Assembler** — поддерживает FreeDOS и Microsoft Windows. Оригинальный Turbo Assembler имел два режима — режим совместимости с MASM и более удобный режим Ideal. Lazy

Assembler использует синтаксис режима *Ideal* и при этом поддерживает современные наборы команд.

- **MASM** от Microsoft также поддерживает только FreeDOS и Microsoft Windows.

Они используют различные варианты (диалекты) синтаксиса Intel, предложенного разработчиком x86 и не используемого на неинтеловских процессорах.

Хотя все эти диалекты обычно объединяются термином «синтаксис Intel», они несовместимы между собой как из-за различных директив, так и из-за неоднозначности трактовки операндов. В частности, команда пересылки (её мнемоническое обозначение `mov` происходит от *move* — перемещать) `mov ecx, dword ptr [0xCCCCCCCC]` записана в соответствии с синтаксисом Intel (очевидный приёмник — регистр общего назначения *ecx* — указан первым, размер операнда-константы задан с помощью конструкции `dword ptr`, то есть равен 32 битам), но по-разному трактуется различными ассемблерами.

Так, MASM преобразует её в код `B9 CC CC CC CC`, который запишет в регистр *ecx* значение указателя, то есть константу `0xCCCCCCCC` (что в синтаксисе Intel также может быть записано как `mov ecx, 0xCCCCCCCC`). Для синтаксиса AT&T это действие записывается как `movl $0xCCCCCCCC, %ecx`, где суффикс `l` после имени команды означает разрядность операндов 32 бита (*long*), префикс `$` — непосредственное значение, а префикс `%` — имя регистра.

Другой популярный ассемблер с синтаксисом Intel, NASM, преобразует `mov ecx, dword ptr [0xCCCCCCCC]` в код `8B 0D CC CC CC CC`, который записывает в *ecx* значение из памяти по заданному указателю (для синтаксиса AT&T это соответствует команде `movl 0xCCCCCCCC, %ecx` — отсутствие префикса перед константой означает разыменованное её как указателя).

GAS также поддерживает синтаксис Intel для архитектуры x86, но в данном пособии будет рассматриваться синтаксис AT&T как более наглядный и универсальный.

## Интегрированные среды разработки

Интегрированные среды разработки (integrated development environment, IDE) включают редактор кода и множество инструментов, облегчающих разработку, в том числе интерфейс для запуска компилятора.

Набор файлов исходного кода, настроек и сценариев сборки объединяется в **проект**. Формат проекта различается для разных IDE. Один и тот же проект может быть собран с различными комплектами настроек. Такой комплект может называться целью или конфигурацией. Для каждого проекта IDE по умолчанию создаёт как минимум две цели сборки: отладочную (Debug) и лишённую отладочной информации (Release). Интерактивная отладка на уровне инструкций языка высокого уровня (в частности, точка останова на конкретном операторе ЯВУ) возможна только для отладочной сборки.

Большинство интегрированных сред может работать с различными компиляторами и отладчиками. Если на компьютере разработчика установлено несколько коллекций компиляторов, в настройках потребуется выбрать необходимый вариант. Многие среды могут поставляться как отдельно, так и комплекте с какой-либо коллекцией компиляторов и отладчиком. Некоторые среды, в частности, IDE Visual Studio от Microsoft, поставляющаяся только с соответствующей коллекцией компиляторов, ограничены в функциональности и полностью поддерживают только одну коллекцию (хотя в последних версиях появилась частичная поддержка GCC). Соответственно, IDE Visual Studio непригодна для разработки на ассемблере с синтаксисом AT&T. Наиболее распространена IDE Qt Creator. Также для разработки с помощью GCC можно использовать IDE Code::Blocks или TheIDE.

## Дизассемблер

Так как языка ассемблера близок к машинному коду, возможно преобразовать машинные команды в инструкции ассемблера. Такое преобразование не всегда однозначно. Соответствующие инструменты — дизассемблеры — широко

применяются в Microsoft Windows для исследования программного обеспечения с закрытым исходным кодом.

В GNU/Linux дизассемблер используется в основном как компонент интерактивного отладчика, позволяющий работать с программой, собранной без отладочной информации.

#### 4.1.2. Этапы компиляции

Компиляция программы на языке C++ в исполняемый файл включает четыре этапа (рис. 4.1): препроцессинг, собственно компиляция (с C++ в ассемблерный код), ассемблирование и компоновка [55].

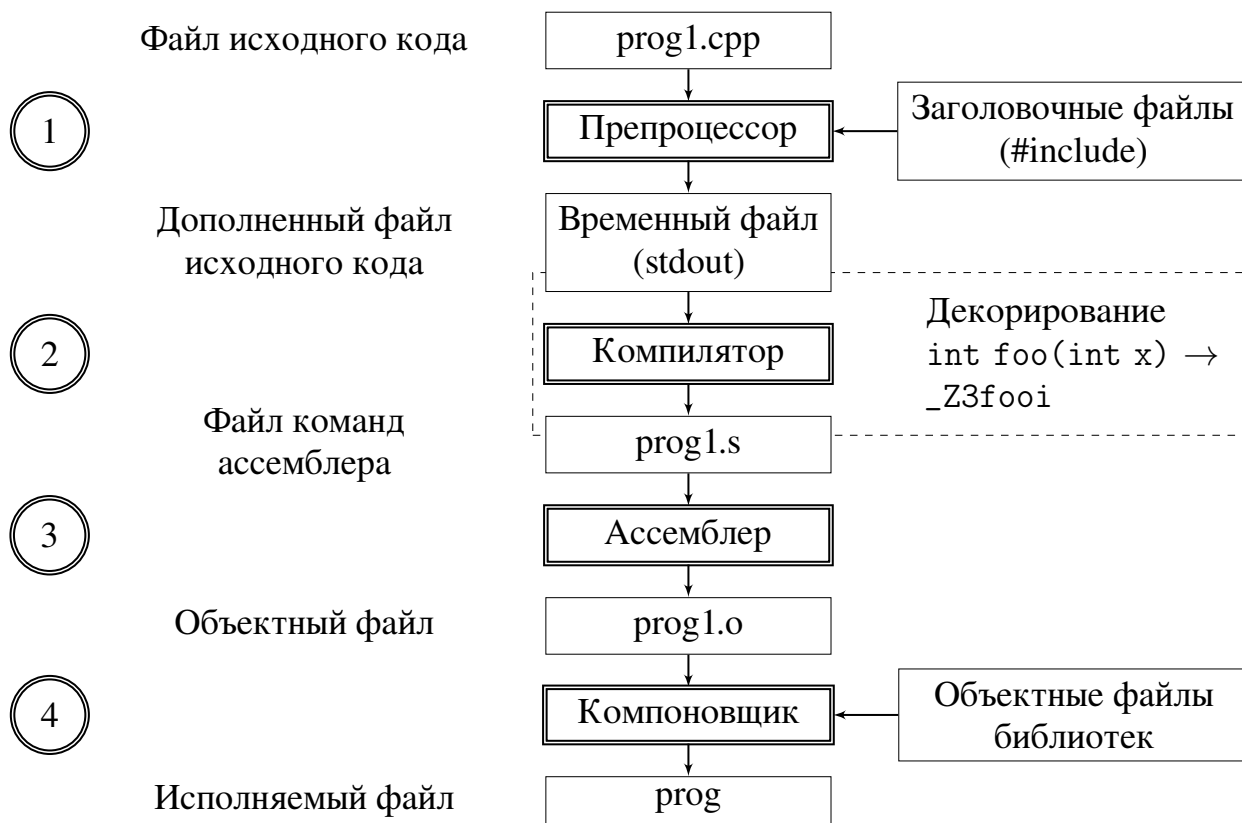


Рис. 4.1. Этапы компиляции программы на C++

Выполнить их все по порядку и получить исполняемый файл, соответствующий программе `main.cpp`, можно командой:

```
1 $ g++ main.cpp
```

Здесь и далее, знак доллара (\$) обозначает стандартное приглашение к вводу команды в консоли (приглашение, оканчивающееся знаком \$, традиционно используется в командных интерпретаторах Unix-подобных систем). Писать знак доллара не требуется. Регистр важен.

Задавая дополнительные ключи, можно остановить компиляцию после какого-либо этапа. Рассмотрим их подробнее.

1. Препроцессор копирует содержимое включённых директивой `#include` заголовочных файлов в исходный код модуля, раскрывает макросы и, в том числе, выполняет текстовые замены «констант», определённых с помощью директивы `#define`, на их значения, а также обрабатывает директивы условной компиляции, выбрасывая из кода те или иные фрагменты.

При использовании компилятора из коллекции GCC увидеть результат препроцессинга можно, воспользовавшись опцией `-E`. Результат будет выведен в стандартный поток вывода (в представленном примере перенаправлен в файл `main.E`).

```
1 $ g++ -E main.cpp > main.E
```

2. Код, обработанный препроцессором, транслируется компилятором в ассемблерный код для соответствующей платформы.

Для остановки компиляции после этого этапа для компилятора `g++` следует воспользоваться ключом `-S`:

```
1 $ g++ -S main.cpp
```

На этапе компиляции выполняется **декорирование** имён функций; таким образом, если остановить ключом `-S` сборку после этого этапа, то в полученном ассемблерном файле имена будут изменены компилятором. В декорированное имя C++-функции включается информация о всех её параметрах (явных и неявных). Имена C-функций изменяются более предсказуемо, так как для них не поддерживается перегрузка (в большинстве случаев имена не изменяются; на

некоторых платформах к ним может добавиться фиксированный префикс или суффикс).

Конкретный алгоритм декорирования зависит от компилятора, платформы и указанного соглашения о вызове. В статье Агнера Фога «Calling conventions for different C++ compilers and operating systems» [9] приведено, в числе прочего, описание алгоритмов декорирования наиболее популярных компиляторов. В частности, имена C-функций при компиляции gcc на платформах GNU/Linux и BSD не изменяются вообще. На 32-разрядной платформе Microsoft Windows имена C-функций при компиляции приобретают дополнительное ведущее подчёркивание (то есть `printf` преобразуется в `_printf`). **Подробнее искажение имён рассматривается в разделе 6.2.7.**

3. Ассемблерный код, созданный компилятором, транслируется в объектный код для соответствующей платформы. Останов компиляции файла после этапа ассемблирования обеспечивается ключом `-c`:

```
1 $ g++ -c main.cpp
```

При ассемблировании имена функций сохраняются.

Некоторые компиляторы скрывают этап ассемблирования, преобразуя исходный код на языке высокого уровня, обработанный препроцессором, напрямую в объектный файл. Компиляторы коллекции GCC можно остановить после любого этапа.

4. Объектные файлы, созданные ассемблером, объединяются компоновщиком (линкером, редактором связей) в исполняемый файл.

На данном этапе компоновщик ищет реализации для всех внешних (`extern`) функций по именам. Соответственно, имена, которые в разных модулях носит одна и та же функция, на этапе компоновки должны совпадать.

Интегрированные среды разработки (IDE) выполняют все этапы автоматически.



### 4.1.3. Расширения файлов исходного кода

Чтобы начать сборку с определённого этапа, достаточно задать для файла расширение, соответствующее этому этапу. Вообще, компилятор — одна из немногих программ, которые учитывают расширение файла при его обработке [58].

Расширение `.s` соответствует ассемблерному файлу. Таким образом, команды `$ g++ main.s` и `$ gcc main.s` эквивалентны и выполняют ассемблирование и компоновку ассемблерного файла `main.s`, минуя этапы препроцессинга и компиляции с ЯВУ.

Расширение `.o` соответствует объектному файлу, и команды `$ g++ main.o` и `$ gcc main.o` выполняют только компоновку файла `main.o`.

В случае, когда необходимо выполнить препроцессинг, ассемблирование и компоновку, выбросив только этап компиляции с ЯВУ (именно такая последовательность оптимальна при сборке модулей, вручную написанных на ассемблере), используется расширение `.S`.

```
1 $ g++ main.S
```

Большинство современных файловых систем чувствительны к регистру имён, а современные **операционные системы**, такие как GNU/Linux и BSD, различают регистр при обработке, так что имена `main.s` и `main.S` будут различаться.

Операционная система Microsoft Windows не различает регистра имён файлов (хотя наиболее часто используемая ею файловая система NTFS теоретически чувствительна к регистру), так что для файлов, требующих препроцессинга, используется расширение `.sx`. Компиляторы GCC трактуют расширение `.sx` аналогично `.S`.

Многие интегрированные среды разработки «не знают» расширений `.s` и `.sx`, так что их необходимо не только вручную добавить в проект, но и указать, что они должны компилироваться и компоноваться как в режиме отладочной сборки, так и в оптимизированном.

#### 4.1.4. Изменение имени выходного файла

Исполняемый файл, полученный после компиляции и ассемблирования GCC, независимо от количества и имён файлов с исходным кодом по умолчанию будет называться `a.out` (assembler output).

Это имя можно изменить, используя ключ `-o`, после которого указывается желаемое имя выходного файла:

```
1 $ g++ main.cpp -o prog
```

Таким образом исполняемый файл, полученный после сборки `main.cpp` (отсутствие ключей остановки сборки соответствует выполнению всех этапов), получит имя `prog`.

Исполняемый файл в большинстве операционных систем не имеет расширения и отличается от неисполняемого правами доступа. **При желании, используя ключ `-o`, можно задать для результирующего исполняемого файла любое имя с любым расширением.**

#### 4.1.5. Компиляция проекта, состоящего из нескольких модулей

При компиляции проекта, включающего несколько модулей, шаги пре-процессинга, компиляции и ассемблирования повторяются для каждого файла исходного кода. Затем из полученных объектных файлов компоновщик собирает единый исполняемый файл (рис. 4.2).

Это можно выполнить одним запуском компилятора из коллекции GCC, **указав в командной строке имена всех файлов исходного кода:**

```
1 g++ -o prog prog1.cpp prog3.cpp
```

Если попытаться собрать каждый из модулей отдельно, мы получим ошибки компоновки (так как во всех модулях, кроме главного, отсутствует точка входа — функция `main`, а в главном — нет функций, описанных в остальных).

Можно остановить сборку после этапа компиляции модулей:

```
1 g++ -c prog1.cpp prog3.cpp
2 g++ -c prog3.cpp
```

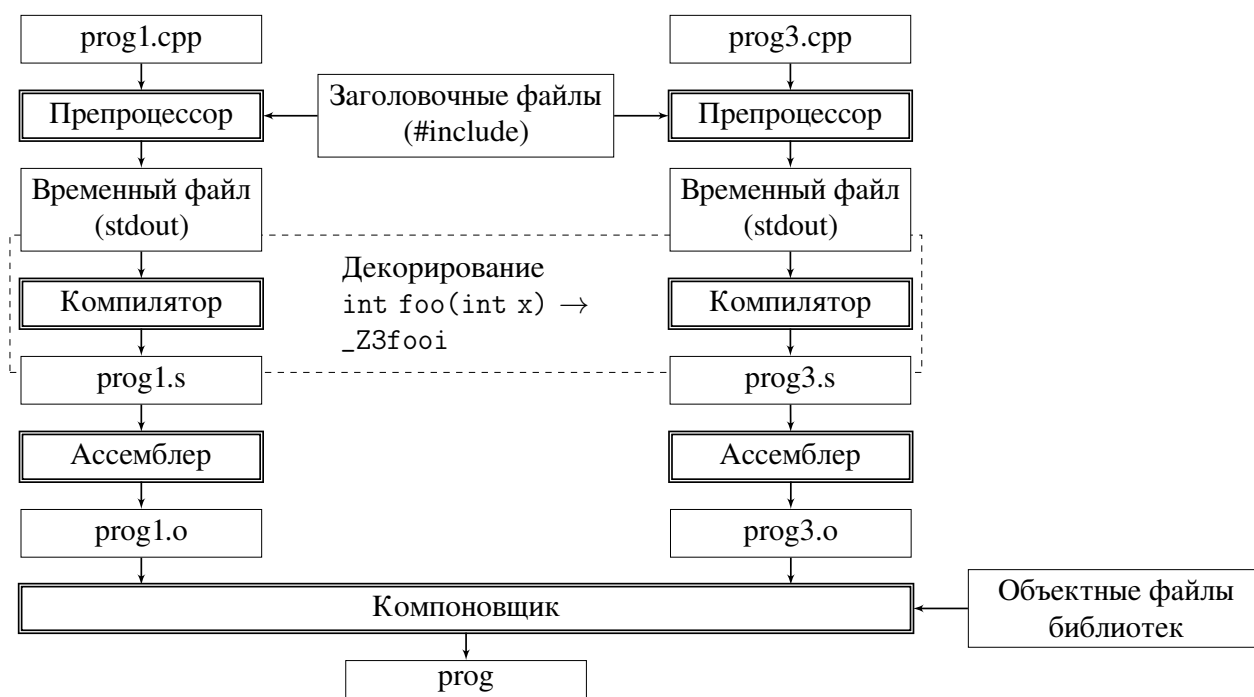


Рис. 4.2. Совместная компиляция нескольких модулей

Затем из полученных объектных файлов одним запуском компоновщика можно получить исполняемый файл:

```
1 g++ -o prog prog1.o prog3.o
```

Ручная сборка небольших проектов обычно выполняется одной командой, в интегрированных средах разработки этапы, как правило, разделяются.

#### 4.1.6. Импорт и экспорт функций

Наиболее универсальным способом использования в одной программе несколько языков программирования является статическая совместная компоновка модулей, написанных на разных языках.

При статической компоновке каждый модуль необходимо скомпилировать из соответствующего языка и ассемблировать, остановив сборку на стадии объектных файлов (напомним, что в GCC для этого используется ключ `-c`). Полученные объектные файлы собираются компоновщиком в единый исполняемый файл (рис. 4.3).

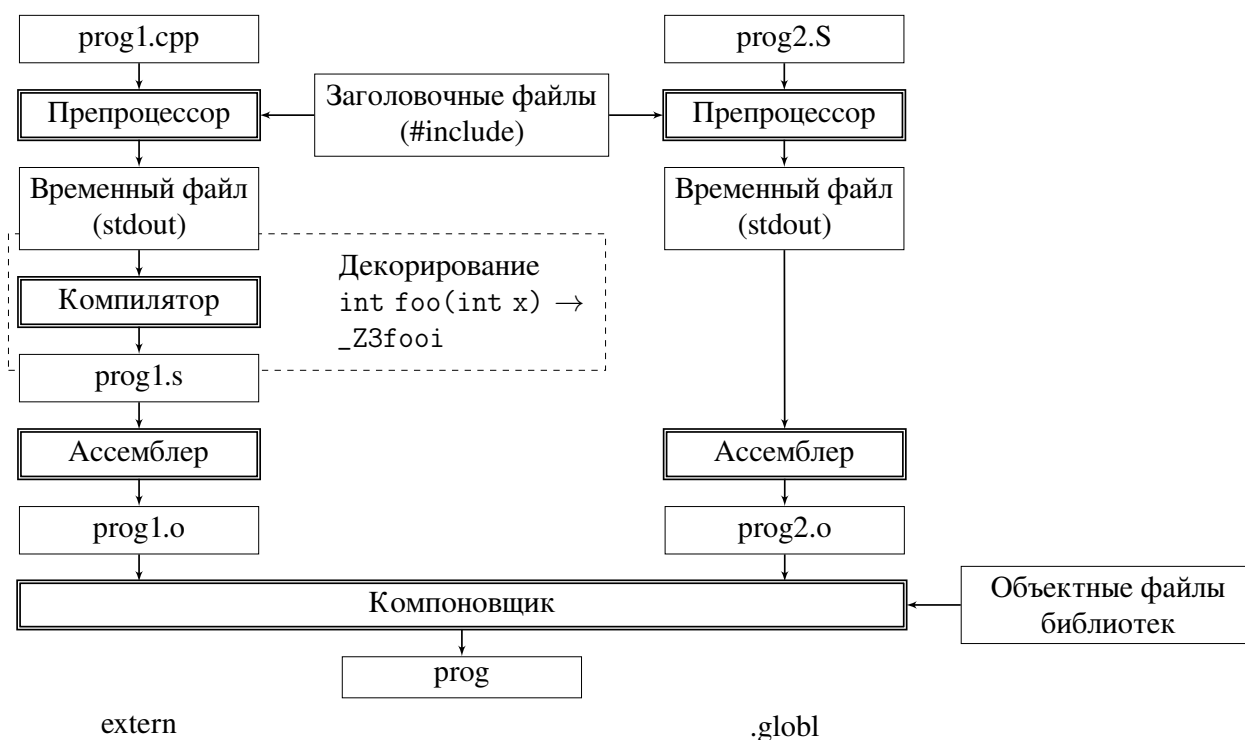


Рис. 4.3. Совместная компиляция модулей на разных языках

Сочетание модулей на языке высокого уровня и на ассемблере используется так часто, что компиляторы коллекции GCC корректно собирают подобный набор модулей одной командой сборки многомодульного проекта, как показано на рис. 4.3:

```
1 g++ prog1.cpp prog2.S
```

Останов сборки и компоновка отдельной командой в этом случае не нужны.

Интегрированные **среды** разработки, поддерживающие компиляторы коллекции GCC, также автоматически выполняют сборку проекта на языке высокого уровня, содержащего ассемблерные модули.

## 4.2. Препроцессор

Чрез горы, степь, моря, леса,  
Вседневно ты по свету скачешь,  
Волшебною ширинкой машешь  
И производишь чудеса.  
*Г. Р. Державин. На счастье*

Первая стадия компиляции программы на C/C++ — обработка исходного кода препроцессором. Препроцессор «не понимает» языков C, C++ или ассемблера и обрабатывает собственный язык — директивы препроцессора. Директива начинается с символа # и заканчивается переводом строки, в частности, #define, #undef, #include, #if, #ifdef, #ifndef, #else, #elif, #endif.

После завершения препроцессинга в тексте программы не остаётся ни директив препроцессора, ни имён определяемых ими макросов.

В настоящее время в программировании на языке C++ препроцессор используется в основном для двух задач — условной компиляции и копирования файла, содержащего заголовки функций, классов и т. п. в несколько файлов исходного кода. Также препроцессор поддерживает **макросы**, которые могут применяться как для обеспечения корректного решения описанных выше задач, так и самостоятельно.

Макросы представляют собой текстовую подстановку, выполняющуюся на этапе препроцессинга, и никак не связаны с особенностями используемого языка — C/C++ или ассемблера. Для того, чтобы отличать их от конструкций языка, принято давать макросам имена, состоящие только из заглавных букв. Их возможности не ограничиваются использованием в условной компиляции.

Тем не менее использование макросов небезопасно и не должно (за исключением условной компиляции) применяться при программировании на высоком уровне.

### 4.2.1. Включение файла

Включение файлов с заголовками выполняется директивой `#include`: запись `#include имя_файла` целиком копирует указанный файл на место, где была эта директива. Имя включаемого файла может содержать путь к нему и заключается в угловые скобки или кавычки. Если имя файла заключено в угловые скобки (`#include <iostream>`), файл должен располагаться в одной из папок со стандартными заголовочными файлами, если имя файла в кавычках (`#include "myheader.h"`) — он должен находиться в папке проекта.

В имени включаемого файла не должно быть комментариев (сочетание `/*` трактуется как маска имени файла, в частности, `"dir/*"` — все файлы в папке *dir*). Зато имя может включать макросы, что позволяет реализовать различные наборы включаемых файлов для различных версий или платформ.

Заголовочные файлы могут включаться как в файлы, содержащие определения функций, так и в другие заголовочные файлы. В первом случае возможна ситуация, когда в файл с определениями в итоге включается несколько копий одного и того же заголовочного файла. Для предотвращения многократного включения внутри заголовочного файла необходимо применять директивы условной компиляции, как показано в листинге 4.1.

#### Листинг 4.1. Защита от повторного включения

```
1 #ifndef THIS_UNIT_ALREADY_INCLUDED
2 #define THIS_UNIT_ALREADY_INCLUDED
3 ... // весь текст заголовочного файла
4 #endif
```

В **актуальный** стандарт C++ планировалось включить поддержку модулей, аналогичных модулям языка Паскаль. Это позволило бы отказаться как от использования директивы `#include`, так и от ручной защиты от многократного включения, но в окончательную редакцию C++17 модули не вошли.

### 4.2.2. Условная компиляция

Условная компиляция обеспечивается директивами `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, обеспечивающими удаление части текста до начала этапа компиляции.

Директивы семейства `#if*` и `#elif` используют условие, которое не должно зависеть от кода C/C++ или ассемблера. Для этого используются **макросы препроцессора**, определяемые директивой `#define`.

Условие директивы `#if` может включать целочисленные литералы, арифметические операторы, макросы и специальный оператор препроцессора `defined`. Истинным считается ненулевое значение условия. Унарный оператор `defined(NAME)` может использоваться только в условиях препроцессора. С его помощью можно узнать, определён ли макрос с именем `NAME`. Для часто употребляемой конструкции `#if defined(NAME)` существует синоним `#ifdef NAME`, для `#if !defined(NAME)` — синоним `#ifndef NAME`.

В простейшем случае текст, расположенный от директивы `#if*` до `#endif` остаётся в коде после препроцессинга в том случае, когда условие истинно и исключается, если условие ложно. Подобная конструкция, в частности, используется для описания отладочных фрагментов которые не должны войти в окончательную сборку программы (листинг 4.2), а также в заголовочных файлах для защиты от повторного включения в один и тот же файл (листинг 4.1).

#### Листинг 4.2. Отладочный фрагмент

```
1 #ifdef DEBUG
2 ...
3 #endif
```

Если между директивой семейства `#if*` и `#endif` находится директива `#else` (листинг 4.3), то в том случае, когда условие истинно, остаётся фрагмент между `#if*` и `#else`, а фрагмент от `#else` до `#endif` удаляется, в случае, если условие ложно — наоборот.

**Листинг 4.3.** Выбор одного из фрагментов

```
1 #ifdef FLAG
2 ...
3 #else
4 ...
5 #endif
```

Используя директиву `#elif`, можно организовать выбор из нескольких фрагментов. В окончательный вариант текста, который будет компилироваться, войдёт только один из фрагментов, расположенных между директивами `#if*` и соответствующей `#endif` и разделённых директивами `#elif` и `#else`.

**4.2.3. Макросы**

Макрос определяется с помощью директивы `#define`. В простейшем случае за директивой следует имя определяемого макроса, а за ним до конца строки — текст, на который производится замена (значение или определение макроса). Имя макроса отделяется от директивы `#define` и значения пробельными символами.

**Листинг 4.4.** Определение макроса без параметров

```
1 #define THE_NUMBER 13
```

После определения макроса `THE_NUMBER` согласно листингу 4.4, приведённого в файле с исходным кодом (непосредственно в его тексте или в тексте включённого заголовочного файла) строка `THE_NUMBER` будет до конца файла с исходным кодом (либо до удаления определения с помощью директивы `#undef`) заменяться на строку `13`. Вхождения строки `THE_NUMBER`, находящиеся до определения этого макроса, останутся без изменений.

При использовании макроса без параметров в тексте его имя заменяется на значение без каких-либо изменений, то есть `int i = THE_NUMBER+1` будет заменяться на `int i = 13+1`. Имя макроса заменяется только в том случае, когда оно является целым словом (то есть отделено от других строк пробельными



ми символами или знаками препинания), в частности, строка `THE_NUMBER_2` не будет заменена на `13_2`.

При этом заменяется любое вхождение макроса как целого слова, то есть, в частности, описание функции `int f(int THE_NUMBER)` будет заменено на `int f(int 13)`, что вызовет ошибку компиляции. В менее благоприятном случае сообщения об ошибке может и не быть. В частности, определение `#define true 0` не приведёт к сбою, но работа программы будет некорректной. Для предупреждения подобных ситуаций необходимо отделять имена макросов препроцессора от имён, используемых в программе. Обычно имена макросов записываются заглавными буквами и не должны быть слишком короткими (в частности, имя `N` с большей вероятностью будет использовано в программе, чем `THE_NUMBER`, поэтому определение `#define N 13` не очень удачно).

Если после имени макроса в строке определения нет ничего, кроме, может быть, пробельных символов, такой макрос имеет пустое значение.

#### Листинг 4.5. Определение макроса с пустым значением

```
1 #define FLAG
```

Макросы с пустым значением обычно используются только в директивах условной компиляции `#ifdef` или `#ifndef`, которые проверяют не значение, а факт наличия макроса. Если такой макрос встречается в тексте, он заменяется на пустую строку.

Удалить определение макроса можно с помощью директивы `#undef`.

#### Листинг 4.6. Удаление макроса

```
1 #undef THE_NUMBER
```

В тексте, лежащем после директивы, приведённой в листинге 4.6, строка `THE_NUMBER` останется без изменения.

## Параметры макросов

С помощью директивы `#define` можно также определить **макросы с параметрами** — лексемы, которые принимают параметры подобно функциям, но фактически являющиеся **текстовой заменой** (более гибким аналогом меню Replace текстового редактора) и раскрываются не во время выполнения (как функции) и не во время компиляции (как шаблоны C++), а до анализа и компиляции программы, никак не сообразуясь с типами переменных, текстом программы и так далее.

При описании макроса после директивы `#define` указывается имя макроса, за которым в скобках (без пробелов) следуют имена параметров, отделённые запятыми и определение макроса, отделённое пробелом.

### Листинг 4.7. Определение макроса с двумя параметрами

```
1 #define MAX(num1, num2) ((num1) > (num2) ? (num1) : (num2))
```

При использовании макроса в тексте после его имени также ставятся круглые скобки, где перечисляются фактические значения параметров макроса.

```
1 int j = MAX(9, i);
```

При подстановке макрос заменяется своим значением, причём на место имён параметров вписывается соответствующий текст.

Параметры макроса при подстановке никак не проверяются. Если в результате такой текстовой подстановки возникнет ошибка, это выяснится только на этапе компиляции, причём не всегда сообщение об ошибке будет вменяемым.

В листинге 4.7 параметры макроса в его определении берутся в скобки, чтобы избежать неприятных ситуаций в том случае, если параметрами будут не имена переменных и литералы, а строки, представляющие собой более сложные выражения C++. Если макрос представляет собой вычисление выражения, то и его тоже лучше взять в скобки, что также сделано в листинге 4.7.

Например, определим макрос `SQUARE(x)` для вычисления квадрата параметра `x`. Макрос, определённый как в листинге 4.8, вычисляет неверное значение в случае, если его параметр является выражением.

**Листинг 4.8.** Неудачное определение макроса с параметрами

```
1 #define SQUARE(x) x*x
```

В частности, текст `i = SQUARE(2+2)` раскроется в `i = 2+2*2+2`, что даст `i`, равное  $2 + 4 + 2 = 8$ , а не  $(2 + 2)^2 = 16$ . Определение из листинга 4.9 заменит `SQUARE(2+2)` на `((2+2)*(2+2))` так что значение квадрата параметра в данном случае будет рассчитано корректно.

**Листинг 4.9.** Более корректное определение макроса с параметрами

```
1 #define SQUARE(x) ((x)*(x))
```

Значение квадрата выражения, модифицирующего свои переменные, будет всё равно вычисляться некорректно. В частности, `SQUARE(i++)` будет раскрыто как `((i++)*(i++))`, так что переменная `i` будет увеличена два раза. Единственный выход — не использовать такие выражения как параметры подобных макросов.

**Объёмные макросы**

Определение макроса должно занимать одну строку. Если строка-подстановка не помещается в строку файла, то в качестве знака переноса строки используется обратная косая черта.

**Листинг 4.10.** Определение объёмного макроса

```
1 #define DISPLAY_ARRAY(arr, size) {\n2     int i;\n3     for (i = 0; i < size; i++) {\n4         printf("%d\\", arr[i]);\n5     }\n6     printf("\\n");\n7 }
```

«Склейка» нескольких строк в одну с помощью косой черты допустима не только в макросах, но и в любом месте программы, но в большинстве прочих случаев не имеет смысла, так как перевод строки в C/C++ является корректным пробельным символом.

### Закавычивание строк

Параметры макроса можно взять в кавычки, используя оператор #.

#### Листинг 4.11. Макрос, заключающий аргумент в кавычки

```
1 #define QUOTES(x) #x
```

Тогда следующий код

```
1 cout << QUOTES(1+2) << "□" << QUOTES(x) << "□" <<  
   QUOTES(мяу) << "□" << QUOTES("мяу") << endl;
```

выведет

```
1 1+2 x мяу "мяу"
```

Чаще всего этот оператор используется в отладочной печати, так как позволяет вывести на стандартный вывод или сохранить в файл имя переменной.

### Конкатенация строк

Параметры можно «склеивать» друг с другом и с произвольными строками с помощью оператора ##. Пример такого макроса приведён в листинге 4.12.

#### Листинг 4.12. Макрос, объявляющий две переменные

```
1 #define DEF_X_Y(typename) typename x##typename = 1,  
   y##typename = 0; \  
2   cout << typeid(x##typename).name() << "□" <<  
   x##typename << endl;
```

Получив имя типа как параметр (например, int), макрос DEF\_X\_Y формирует текст, который объявляет и инициализирует две переменные x<имя типа> и y<имя типа>, а также выводит на стандартный вывод характеристику типа и значение переменной x<имя типа>. Следующий код не вызовет ошибок компиляции.

```
1    DEF_X_Y(int)
2    yint = xint+1;
```

Конкатенация строк может быть использована для изменения имён функций или переменных «на лету» в зависимости от версии программы или используемой платформы, что иногда необходимо.

### 4.3. Ассемблерные вставки в код C++

Наша тяжёлая задача — быть стрелочниками на путях  
встречи Прошлого и Будущего.

*В. Хлебников. Воззвание Председателей  
Земного Шара*

Для вставки одной или нескольких инструкций ассемблера в код на C++ используется ключевое слово `asm` [7, 10, 25].

Стандарт C++ описывает использование `asm` следующим образом:

```
1 asm ( string-literal ) ;
```

конкретный вид `string-literal` при этом не регламентируется. Обычно такие вставки используются для передачи кода непосредственно ассемблеру.

Содержимое таких вставок зависит как от архитектуры целевого аппаратного обеспечения (набор команд, регистров и т. д.), так и от компилятора (мнемоники команд, порядок операндов, синтаксис и т. д.).

Используемая операционная система определяет только обращения *непосредственно к интерфейсам операционной системы* (системные вызовы Linux и BSD, функции API Windows т. п.). Вычисления, а в тридцатидвухбитных системах также и обращение к стандартной библиотеке C (**libc**) или к кроссплатформенным библиотекам типа Qt, описываются одинаково под любой операционной системой (при этом имена функций могут отличаться, что подробнее описано в разделе 6.2.7).

Таким образом, механизм ассемблерных вставок в код на языке C++ позволяет получить переносимую между различными операционными системами программу, включающую фрагменты на ассемблере.

#### 4.3.1. Синтаксис ассемблерных вставок в GCC

Ассемблерная вставка в программу, собираемую компилятором GCC, может быть описана с использованием двух ключевых слов: `asm` либо `__asm__` [62, 66]. Эти формы равнозначны и используют одинаковый синтаксис. Форма `__asm__` применяется, когда стандартное ключевое слово `asm` конфликтует с чем-либо в программе.

##### Базовая форма

Базовая форма ассемблерной вставки выглядит следующим образом.

##### Листинг 4.13. Базовая форма вставки

```
1 asm [volatile] (  
2     "команды_и_директивы_ассемблера"  
3     "как_последовательная_текстовая_строка"  
4     );
```

Ключевое слово `volatile` для базовой формы не оказывает никакого эффекта, так как такая вставка не оптимизируется компилятором никогда.

Например:

```
1 asm("movl_%%ecx, %%eax"); // EAX = ECX
```

или

```
1 __asm__("movb_%%bh, %%eax"); // *EAX = BH (байт)
```

Если код во вставке содержит более одной строки, то каждая строка помещается в кавычки. Тем не менее, этого недостаточно для разделения инструкций.

##### Разделение инструкций

При компиляции соседние строки ассемблерной вставки склеиваются, как склеиваются части любой строковой константы — точно так же, как в объявлении вида

```
1     char *s = "abcd"
```

```
2      "ABCD" ;
```

которое задаёт строку "abcdABCD", не разделённую посередине никаким символом. То есть вставка

```
1 asm ("movl_%eax,%ebx"
2      "movl_$56,%esi");
```

На самом деле выглядит как

```
1 asm ("movl_%eax,%ebxmovl_$56,%esi");
```

и вызывает логичное сообщение о некорректном имени регистра «ebxmovl».

Таким образом, если во вставке необходимо использовать более одной инструкции, то в конце каждой строки необходимо поместить суффикс `\n` для разделения инструкций (если желательно иметь красивый выходной файл, можно использовать суффикс `\n\t`, если форматирование выходного файла безразлично — можно использовать вместо `\n` разделитель `;`).

```
1 asm ("movl_%eax,%ebx\n"
2      "movl_$56,%esi\n"
3      "movb_%ah,_(%ebx)");
```

### Доступные в базовой форме вставки имена

Внутри ассемблерной вставки можно обращаться по именам к регистрам, а также к глобальным переменным программы (листинг 4.14).

#### Листинг 4.14. Увеличение глобальной переменной *n*

```
1 int n = 12;
2
3 int main(){
4     asm ("incl_n");
5     cout << n << endl;
6     return 0;
7 }
```

Локальные переменные функций размещаются компилятором в стеке (причём не всегда в порядке объявления), поэтому к ним необходимо обращаться, используя *расширенный ассемблер GCC*, а именно **параметры вставок**.

Необходимо также учитывать, что в регистрах, используемых во вставке, ранее компилятором могла быть размещена какая-либо регистровая переменная. В этом случае компилятор не определяет автоматически, что регистр изменён, что может привести к краху программы.

На практике описанную выше базовую форму ассемблерных вставок (без параметров) не стоит использовать *никогда*.

Только расширенная форма даёт возможность корректного взаимодействия с программой на ЯВУ.

#### 4.3.2. Расширенная форма

Синтаксис вставки (расширенный) в GCC может иметь две формы.

Первая выглядит следующим образом (листинг 4.15).

##### Листинг 4.15. Расширенная форма с выходными параметрами

```
1 asm [volatile] (  
2     "команды_и_директивы_ассемблера"  
3     "как_последовательная_текстовая_строка"  
4     : [<выходные параметры>] : [<входные параметры>] :  
      [<перезаписываемые регистры>]  
5     );
```

Ключевое слово `volatile` используется для того, чтобы указать компилятору, что вставляемый ассемблерный код может обладать побочными эффектами, поэтому попытки оптимизации могут привести к логическим ошибкам.

Вторая форма расширенной ассемблерной вставки имеет вид, приведённый в листинге 4.16.

##### Листинг 4.16. Расширенная форма с метками выхода

```
1 asm [volatile] goto (  
2     "команды_и_директивы_ассемблера"  
3     "как_последовательная_текстовая_строка"  
4     :: <входные параметры> : <перезаписываемые регистры> : <метки>  
5     );
```

Ключевое слово `goto` указывает, что ассемблерный код может делать переходы на метки, перечисленные в соответствующем разделе.



Обращение к параметру-метке предваряется префиксом %1 (от *label*), за которым идёт порядковый номер метки в списке всех параметров; псевдоним для метки указать нельзя. В частности, данный код переходит к метке `carry`: (и функция возвращает 1), если в числе  $p1$  взведён бит с номером  $p2$  (младший бит имеет нулевой номер).

#### Листинг 4.17. Переход к метке `carry` по значению бита

```

1 asm goto (
2     "bt1_ %[BitIdx] ,_ %[Val] \n\t"
3     "jc_ %12"
4     :: [Val]"r" (p1), [BitIdx]"r" (p2)
5     : "cc"
6     : carry
7 );
8
9 return 0;
10
11 carry:
12 return 1;
```

Команда `bt` копирует бит, номер которого передается вторым параметром, в флаг *CF*. Затем происходит условный переход на метку `label`: в том случае, если флаг *CF* взведён.

Параметры указываются как %0, %1 и т. д. или как %[Value], где Value — определённый в разделе параметров псевдоним (если параметр соответствует какой-либо переменной ЯВУ, то псевдоним может как совпадать, так и не совпадать с именем этой переменной). Чтобы избежать конфликтов с именами регистров, регистры во вставке с расширенным синтаксисом указываются с префиксом %, например, %eax.

Общее количество параметров ограничено:  $input + output + goto \leq 30$ .

### Выходные параметры

Раздел выходных параметров состоит из описаний отдельных параметров (операндов), разделённых запятыми.

Описание выходного параметра в общем случае имеет вид [24]:

```
1 [ [asmSymbolicName] ] constraint (cvariablename)
```

где

**asmSymbolicName** определяет псевдоним для операнда. Область определения такого псевдонима — вся ассемблерная вставка. Псевдоним может быть любым допустимым идентификатором C++. Два разных операнда не могут использовать один псевдоним.

Псевдоним выходного параметра может быть указан как ограничение расположения входного. В этом случае параметр с данным псевдонимом является входным и выходным одновременно (как правило, это изменяемая во вставке переменная).

Если псевдоним не определён, используются номер операнда, начиная с нуля: %0 для первого, %1 для второго и т. д.

**constraint** строковая константа, описывающая ограничения на расположение операнда.

Для выходного параметра начинается с символа = (если значение параметра перезаписывается) или + (если значение параметра используется как для чтения, так и для записи).

После префикса указывается одно или более ограничение. Чаще всего используются ограничения r (выходной параметр размещается в регистре) и m (выходной параметр размещается в памяти). Если указано несколько вариантов размещения, компилятор выбирает наиболее эффективный.

При выборе регистра для размещения выходного параметра компилятор не рассматривает регистры из списка перезаписываемых.

**cvariablename** — выражение C++ (lvalue), куда будет записано значение выходного параметра (обычно имя переменной).

Если для выходного параметра указано размещение в регистре и имя соответствующей переменной, это не значит, что переменная обязательно будет размещена в регистре. Возможно, в регистре будет сформировано вы-

ходное значение, которое затем будет скопировано в локальную стековую переменную.

Например, данный фрагмент записывает константу 12 в переменную *x*.

#### Листинг 4.18. Перезапись выходного параметра

```
1 int x;  
2 asm  
3 (  
4     "movl_$12,%[x]\n"  
5     : [x] "=rm"(x)  
6 );
```

начальное значение параметра `[x]` не определено.

Следующий фрагмент кода удваивает *x*:

#### Листинг 4.19. Модификация выходного параметра

```
1 asm  
2 (  
3     "addl_%[x],_%[x]\n"  
4     : [x] "+r"(x)  
5 );
```

Символ `+` в строке расположения указывает на то, что данный параметр является не чисто выходным, а модифицируемым. Начальное значение параметра `[x]` берётся из выражения-приёмника, в данном случае — переменной *x*.

### Входные параметры

Раздел входных параметров состоит из описаний отдельных параметров (операндов), разделённых запятыми.

Описание входного параметра в общем случае:

```
1 [ [asmSymbolicName] ] constraint (cexpression)
```

где

**asmSymbolicName** определяет псевдоним для операнда, аналогично псевдонимам выходных параметров.

Если псевдоним не определён, используются номер операнда, начиная со следующего за последним выходным параметром. Если есть два выходных параметра, то первый входной будет обозначен %2 и т. д.

**constraint** строковая константа, описывающая ограничения на расположение операнда.

Для входного параметра не может начинаться ни с =, ни с +.

После префикса указывается одно или более ограничение. Чаще всего используются ограничения r (входной параметр размещается в регистре) и m (входной параметр размещается в памяти). Если указано несколько вариантов размещения, компилятор выбирает наиболее эффективный.

Если входной параметр должен располагаться там же, где один из выходных, расположение также может быть указано цифрой или псевдонимом данного выходного параметра.

При выборе регистра для размещения входного параметра компилятор не рассматривает регистры из списка перезаписываемых.

**cvariablename** — выражение C++, откуда берётся значение входного параметра (также может быть именем переменной).

Например, конструкция:

```
1 asm ("movl %0, %%eax" : "i"(1) : "%eax");
```

превратится после компиляции в:

```
1 movl $1, %eax
```

Данный фрагмент эквивалентен `foo += bar`:

```
1 int foo = 6, bar = 12;
2 asm ("addl %[bar], %[foo]"
3      : [foo]"r"(foo)
4      : [bar]"g"(bar)
5      : "cc"
6 ); // foo += bar
```

причём, если переменная *foo* не располагается в регистре изначально, её значение будет скопировано в регистр, а после добавления *bar* результат будет

скопирован обратно в *foo*. Для параметров в памяти это в общем случае не так. Если исходное значение находится не в памяти, при компиляции возможно сообщение о некорректном расположении.

### Перезаписываемые элементы

Код в ассемблерной вставке может прямо или косвенно изменять значения не только выходных параметров, но и прочих регистров. Эти регистры перечисляются в списке перезаписываемых (clobber) регистров [24].

В частности, команда умножения в своей классической форме (*eax = eax · sm*) записывает старшую часть произведения в *edx*.

#### Листинг 4.20. Неявно перезаписываемый регистр

```
1 asm("imull□[M]\n"
2     : [X]"=a"(foo)
3     : "[X]"(foo), [M]"r"(3)
4     : "cc", "%edx"
5 );      // foo *= 3
```

Если во вставке явно модифицируется какой-либо регистр (что не рекомендуется; в случае, если нужен регистр для хранения временного значения, лучше объявить фиктивный входной параметр, как описано в следующем подразделе), его также необходимо указать в списке перезаписываемых элементов.

#### Листинг 4.21. Явно перезаписываемый регистр

```
1 asm ("movl□[One],□%%eax"
2     : : [One]"i"(1)
3     : "%eax");    // movl $1, %eax
```

Неуказание таких регистров в списке перезаписываемых, скорее всего, приведёт к тому, что компилятор поместит один из параметров вставки или переменных ЯВУ в этот же регистр и, соответственно, к непредсказуемому значению в нём.

Специальные аргументы для списка перезаписываемых регистров:

"cc" показывает, что код изменяет регистр флагов.

Флаги изменяют все арифметические инструкции.

#### Листинг 4.22. Перезапись флагов

```
1 asm("addl_□$12,□[X]\n"  
2     : [X] "=rm" (foo)  
3     : "[X]" (foo)  
4     : "cc"  
5 );      // foo += 12
```

"memory" показывает, что код меняет значение в памяти (не считая явно указанных выходных параметров) — например, интерпретирует какой-либо из параметров как указатель.

#### Листинг 4.23. Перезапись памяти по указателю

```
1 asm("movl_□$12,□(%[p])\n"  
2     :: [p] "r" (&foo)  
3     : "memory"  
4 );      // foo = 12
```

При указании "memory" в списке перезаписываемых элементов все операции работы с памятью, которые были в программе до ассемблерной вставки, выполнятся до неё, а те, что стоят в программе после — будут после. В противном случае компилятор может поменять местами как операторы C++, так и ассемблерные вставки.

### 4.3.3. Практическое использование параметров

#### Временные регистры

Использовать под временные данные фиксированные регистры и указывать их в списке перезаписываемых не вполне корректно [52].

Более правильным будет ввести фиктивный выходной параметр, размещаемый в регистре (=r или =&r, либо более конкретное ограничение — подробнее указано в разделе 4.3.4), так как это даст компилятору больше свободы при оптимизации.

В тексте вставки можно использовать данный параметр для хранения произвольных временных данных.

Пересылка через временный регистр:

#### Листинг 4.24. Фиктивный выходной параметр как временный регистр

```

1 int src = 1, dst, tmp;
2 asm
3 (
4     "movl_%[SRC],_%[TMP]\n"
5     "movl_%[TMP],_%[DST]\n"
6     : [DST]"=g"(dst), [TMP]"=r"(tmp)
7     : [SRC]"g"(src)
8 );

```

#### Модификация входных параметров

Модифицировать входные параметры, не являющиеся одновременно и выходными, **нельзя**.

Если возникает необходимость изменить значение, переданное как входной параметр, оно должно быть описано ещё и как выходной.

Рассмотрим код, который увеличивает переменную *foo* на значение переменной *bar*. В данном случае *bar* может быть только входным параметром, а *foo* — модифицируемым, то есть одновременно и входным, и выходным.

В следующем фрагменте неименованный параметр %0 является входным и выходным одновременно (точнее, входной параметр %1 располагается там же, где и выходной %0). Это определяется ограничением размещения входного параметра %1 — там указан номер параметра %0.

```

1 asm ("addl_%2,_%0"
2     : "=r" (foo)
3     : "0" (foo), "g" (bar)
4     : "cc"
5 ); // foo += bar

```

Аналогичные вычисления с использованием именованных параметров ([S] здесь — выходной параметр в регистре, и в том же регистре расположен неименованный входной параметр) выглядят следующим образом.

```

1 asm ("addl_%[A],_%[S]"
2     : [S]"=r" (foo)
3     : "[S]"(foo), [A]"g" (bar)
4     : "cc"

```

```
5 ); // foo += bar
```

Мы используем неименованный выходной параметр %1, чтобы задать начальное значение входному [S]. В данном случае (входной параметр инициализируется исходным значением *foo* — приёмника совмещённого с ним выходного параметра [S]) можно было и не вводить дополнительный входной параметр, а заменить описание [S] на [S] "+r" (*foo*).

Описание параметра как выходного и входного одновременно (точнее, однорасположенной пары параметров «выходной-входной»), в частности, делает параметр сохраняющим предсказуемое значение на всём протяжении вставки. Если такой параметр размещается в регистре, никакой другой параметр данной вставки не будет помещён в этот регистр.

Отличие такого описания параметра от выходного с начальным значением (расположение которого начинается с +) — то, что источник начального значения может отличаться от исходного значения назначения. Так, следующий фрагмент кода устанавливает начальное значение параметра-регистра [S] нулём, а значение этого регистра после завершения вставки записывается в переменную *foo*:

```
1 asm ("addl_□[A],_□[S] "
2      : [S] "=r" (foo)
3      : "[S]"(0), [A] "g"(bar)
4      : "cc"
5 ); // foo = 0 + bar
```

Также для входного параметра, расположенного на месте выходного, можно задать отдельный псевдоним.

```
1 asm ("addl_□[A],_□[S] "
2      : [S] "=r" (foo)
3      : [S_initial_value] "[S]"(0), [A] "g"(bar)
4      : "cc"
5 ); // foo = 0 + bar
```

## Конфликт выходных и входных долгоживущих параметров

При размещении параметров в регистрах компилятор считает, что входные параметры используются в начале вставки, а выходные — в конце, *после* ис-



пользования всех входных. Соответственно, «только входные» параметры часто располагаются в тех же регистрах, что и «только выходные», чтобы освободить максимальное количество регистров для других переменных.

Это может привести к конфликту, если какой-либо входной параметр используется в конце вставки, после присваивания хотя бы одного из выходных параметров. При размещении этих двух параметров в одном регистре присваивание уничтожит оригинальное значение входного параметра.

В частности, данный фрагмент (вычисляющий  $y = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$ ) в случае размещения входного параметра [Zero] и выходного [y] в одном регистре работает некорректно ( $y$  всегда равен 1, так как инициализация выходного параметра [y] единицей выполняется до чтения входного параметра [Zero], и в момент условного присваивания `cmovll %[Zero], %[y]` регистр, где расположены оба этих параметра, равен 1).

#### Листинг 4.25. Параметр $y$ конфликтует с *Zero*

```

1 int x = 10, y;
2 asm
3 (
4   "cml_0,_%[x]\n"
5   "movl_$1,_%[y]\n"
6   "cmovll_%[Zero],_%[y]\n"
7   : [y]"=r"(y)
8   : [x]"rm" (x), [Zero]"rm"(0)
9   : "cc"
10 ); // y = 1 независимо от x

```

Это можно исправить, объявив [y] выходным параметром с начальным значением (+ вместо = в расположении), хотя нам и не требуется исходное значение переменной  $y$ . Такое ограничение не позволит компилятору совместить какой-либо входной параметр с [y].

```

1 asm
2 (
3   "cml_0,_%[x]\n"
4   "movl_$1,_%[y]\n"
5   "cmovll_%[Zero],_%[y]\n"

```

```

6   : [y] "+r" (y)
7   : [x] "rm" (x), [Zero] "rm" (0)
8   : "cc"
9 );

```

Другой способ — явно указать, какой входной параметр следует расположить на месте выходного [y]. Здесь мы можем совместить с [y] входной [x], так как последнее чтение [x] (сравнение [x] с нулём) выполняется до инициализации [y]:

```

1  asm
2  (
3    "cmpl_□$0,□□[x]\n"
4    "movl_□$1,□□[y]\n"
5    "cmovll_□□[Zero],□□[y]\n"
6    : [y] "=r" (y)
7    : [x] "[y]" (x), [Zero] "rm" (0)
8    : "cc"
9 );

```

Оба этих способа позволяют разрешить конфликт и получить корректное значение *y*.

#### 4.3.4. Ограничения на расположение операнда

Некоторые общие ограничения (типы размещения параметров) [24]:

**r** — регистр;

**m** — память;

**i** — непосредственное значение (константа), известное на этапе компиляции или компоновки (без ограничений по значению — в диапазоне 0..0xffffffff);

**g** — память, регистр общего назначения или непосредственное значение (обычно этот тип эквивалентен *rim*);

**n** — непосредственное значение (константа), известное на этапе компиляции.

Некоторые ограничения (типы), специфичные для i386 (определены в файле i386.h GCC)

**q** — регистр, который можно адресовать по байтам младшего слова (*A, B, C, D*);

**A** — пара регистров *D : A*;

**a, b, c, d, S, D** — регистры *A, B, C, D, si, di* соответственно;

**I..P** — машинно-зависимые ограничения для констант:

**I** — непосредственное значение в диапазоне 0..31 (32-битный сдвиг);

**J** — непосредственное значение в диапазоне 0..63 (64-битный сдвиг);

**M** — непосредственное значение в диапазоне 0..3 (сдвиги, которые может выполнить `lea`);

**N** — непосредственное значение в диапазоне 0..255 (беззнаковое однобайтовое).

#### 4.3.5. Модификаторы параметров

Иногда в коде программы требуется подставить не значение параметра в неизменённом виде, а какую-либо его характеристику. В этом случае необходимо использовать так называемые модификаторы параметров (таблица 4.1).

Модификатор указывается между префиксом % и именем параметра.

В частности, при инициализации параметра `[dmem]`, находящегося в памяти, необходимо указать размер приёмника. В GCC это делается при помощи суффикса размера (раздел 5.1.5). Явное указание суффикса сделает программу неустойчивой к изменению типа переменной, передающейся как `[dmem]`. Если суффикс указан с помощью модификатора `z`:

```
"mov%z [dmem] , $0 , % [dmem] \n"
```

данная команда будет корректно инициализировать переменную как типа *short*, так и *int* или *long long*.

Печать константы без префикса \$ необходима, если эта константа используется не как непосредственный операнд команды, а как-то иначе. В частности, такая константа может быть частью адреса.

```
"movl $13 , %c [FieldDisp] (% [Struct]) \n"
```

## Модификаторы параметров ассемблерных вставок GCC

Таблица 4.1

Модификатор	Действие
<b>z</b>	Печать суффикса команды для размера операнда
<b>c</b>	Печать константы без префикса \$
<b>b</b>	Печать имени младшего байта регистра (%a1 для регистра <i>A</i> )
<b>h</b>	Печать имени старшего байта младшего слова регистра (%ah для регистра <i>A</i> )
<b>w</b>	Печать имени младшего слова регистра (%ax для регистра <i>A</i> )
<b>k</b>	Печать имени младшего двойного слова регистра (%eax для регистра <i>A</i> )
<b>q</b>	Печать 64-битного варианта имени регистра (%rax для регистра <i>A</i> )

Приведённый фрагмент ассемблерной вставки инициализирует поле структуры, расположенной по адресу [Struct]. Смещение поля задано параметром [FieldDisp].

Модификаторы печати имени части регистра доступны только для параметров в регистрах, причём печать младшего байта и старшего байта младшего слова — только в тех, где эти байты можно адресовать. В тридцатидвухбитном режиме как младший байт, так и следующий за ним можно адресовать только для *A–D*. В шестидесятичетырехбитном младший байт адресуется для всех шестнадцати регистров общего назначения с помощью префикса *REX*. Второй байт доступен только для *A–D*, причём не в любой ситуации (использование префикса *REX* запрещает доступ к *ah–dh*).

## Контрольные вопросы

1. Какие стадии включает компиляция программы с помощью GCC?
2. Какое расширение имеет файл с исходным кодом на языке ассемблера?
3. Какое расширение имеет файл с исходным кодом на языке C++?
4. Как изменить имя выходного файла при сборке?
5. Как собрать программу, состоящую из нескольких модулей?
6. Каким ключевым словом открывается ассемблерная вставка?
7. Как из ассемблерной вставки обратиться к локальным переменным?
8. Какие вы знаете ограничения на размещение параметров ассемблерных вставок?

## Глава 5

# Синтаксис и команды GNU Assembler x86

Истинно, и при тяготе бывает льгота.

*Коран. 94.6*

Язык ассемблера — простейший символический язык программирования, каждая команда которого транслируется в одну команду машинного языка. Также существуют операторы ассемблера, не соответствующие машинным командам — директивы, комментарии, пустые операторы.

Набор команд ассемблера определяется как архитектурой используемого компьютера, так и собственно ассемблером — транслятором с символического языка в объектный код. Команды имеют текстовые мнемонические обозначения (мнемоники). При этом одной и той же мнемонике может соответствовать несколько опкодов, выполняющих схожие действия над операндами, расположенными в разных местах или разного размера. Часто существует один опкод для восьмибитных операндов и другой — для операндов, имеющих размер 32, 64 или 16 бит (для него разрядность операндов определяется текущим режимом и префиксами). Таким образом, иногда команды обрабатывают восьмибитные операнды немного иначе, чем любые другие. Кроме того, одному опкоду может соответствовать несколько разных мнемонических обозначений.

В данной главе описывается часть набора команд GNU Assembler (GAS) для архитектуры x86 с использованием традиционного синтаксиса AT&T, а также особенности этого синтаксиса.

## 5.1. Особенности GNU Assembler

Поскоблите язык — и вы увидите пространство  
и его шкуру.

*В. Хлебников. Зангези*

GAS, как и его предок, ассемблер Unix `as`, использует так называемый синтаксис AT&T System V/386, часто называемый просто синтаксисом AT&T или синтаксисом GAS [8, 43].

Также для процессоров семейства Intel x86 часто используется синтаксис, предложенный фирмой Intel. Основными отличиями синтаксиса Intel от AT&T считаются обратный порядок операндов, другие обозначения адресации и невозможность явного указания разрядности операции. Менее известно различие в мнемонических обозначениях команд. Современные версии ассемблера GAS поддерживают оба варианта синтаксиса.

Основными недостатками синтаксиса Intel является неоднозначность и трудность чтения инструкций. Кроме того, синтаксис Intel используется только для процессоров Intel или совместимых с ними.

Синтаксис AT&T иногда называется кроссплатформенным, так как GCC и, соответственно, GAS реализован для множества различных архитектур. Полной кроссплатформенности при использовании языка ассемблера достичь невозможно, так как у каждой платформы свой набор команд, регистров и методов адресации, но использование схожего синтаксиса облегчает переход между ними.

Также инструкции, записанные в соответствии с синтаксисом AT&T, легче читаются. Даже обозначение адреса в памяти, которое вначале кажется контринтуитивным, распознаётся однозначно. Косвенный адрес в памяти, записанный по правилам синтаксиса AT&T, невозможно перепутать с непосредственным значением, что происходит в диалектах Intel.

### 5.1.1. Общие правила

Так как GAS в основном используется на одном из этапов компиляции программы на C/C++, многие синтаксические конструкции GAS и C/C++ совпадают.

В программе могут использоваться латинские буквы, цифры, а также нижнее подчёркивание и точка. Допустимые пробельные символы — пробел и табуляция; они могут сочетаться в любом порядке. Перевод строки является разделителем операторов.

Допускаются многострочные комментарии `/*` в стиле C `*/` и однострочные `//` в стиле C++. Также для различных платформ поддерживаются платформо-специфичные виды однострочных комментариев. В частности, для x86, кроме однострочного комментария в стиле C++, поддерживается символ комментария `#`, но он считается устаревшим.

Оператор ассемблера целиком размещается на одной строке. В начале строки может быть одна или несколько меток, заканчивающихся двоеточием. Если первый символ оператора — точка, то это — директива ассемблера (первая строка листинга 5.1). Набор основных директив совпадает для всех архитектур, но для многих платформ есть и специфичные (в частности, директивы определения данных).

#### Листинг 5.1. Директива и команда

```
1 the_label:      .directive      ...
2 another_label:           // Пустой оператор
3           instruction  operand_1, operand_2, ...
```

Пустой оператор может состоять только из пробельных символов или быть пустой строкой (вторая строка). Оператор, начинающийся с буквы, представляет собой мнемоническое обозначение машинной команды, за которым при необходимости следуют операнды, разделённые запятыми (третья строка листинга 5.1).



Строковые литералы ограничиваются двойными кавычками, экранирующим символом является обратный слеш «\», спецсимволы кодируются аналогично C/C++. Числовые литералы также описываются аналогично C/C++.

### 5.1.2. Основные директивы

Директива ассемблера не соответствует никакой машинной команде. Рассмотрим несколько наиболее употребительных директив; их можно разбить на несколько классов.

#### Директивы определения секций

Как было описано в разделе 3.2, код программы и различные виды данных должны располагаться в различных диапазонах адресного пространства. По историческим причинам эти диапазоны называются сегментами (соответствующие фрагменты исходного кода и исполняемого файла — секциями). Начало содержимого того или иного сегмента в исходном ассемблерном коде отмечается специальными директивами.

С начала файла до первой директивы располагается сегмент кода. Указать продолжение сегмента кода можно директивой **.text**.

Сегмент данных открывается директивой **.data**. В принципе, описание статических данных в сегменте кода не вызовет ошибки, но такие данные будет невозможно модифицировать, так как сегмент кода защищён от изменений.

Для сегментов стека и кучи нет соответствующих секций, они заполняются динамически в процессе выполнения программы.

#### Директивы определения данных

В сегменте **.data** статические данные описываются также с помощью директив.

После директивы определения данных указывается литерал подходящего типа или несколько литералов, перечисленных через запятую. В памяти соответственно резервируется одна или несколько ячеек соответствующего размера, которые инициализируются указанными значениями.

Для того, чтобы дать адресу ячейки имя, перед соответствующим определением необходимо поставить метку (листинг 5.2).

### Листинг 5.2. Определение статических данных

```
1 foo: .long 0, 1, 2
2 bar: .double -8.7
```

Важно помнить, что ассемблер, в отличие от языков высокого уровня, не является типизированным. Таким образом, если, в частности, по адресу *bar* расположено восьмибайтовое значение двойной точности  $-8,7$ , а программист обратится к нему как к числу одинарной точности (четыре байта), это не вызовет сообщения об ошибке, но прочитанное значение будет другим.

Для инициализации памяти целыми значениями различного размера используются следующие директивы: **.byte** — однобайтовое (восьмибитное) целое, **.short** — шестнадцатибитное, **.long** — тридцатидвухбитное, **.quad** — шестидесятичетырехбитное. Размер и порядок байтов определяются платформой; приведены размеры для x86 (они же наиболее распространённые). Также существуют директивы **.word** и **.int**, для x86 определяющие шестнадцати- и тридцатидвухбитные целые соответственно.

Значения с плавающей запятой одинарной (32 бита) и двойной (64 бита) точности описываются директивами **.float (.single)** и **.double**.

Для инициализации памяти строковыми константами различного вида используются директивы **.string (.asciz)** и **.ascii**.

Функции стандартной библиотеки C используют строки, завершающиеся нулём; их можно описать директивой **.string (.asciz)** (листинг 5.3).

### Листинг 5.3. Определение строки, завершающейся нулём

```
1 msg: .string "Hello, \world!\n"
```

Если после директивы указывается несколько строковых литералов через запятую, завершающий ноль добавляется после каждого.

Строка без завершающего нуля описывается директивой **.ascii** (листинг 5.4).

#### Листинг 5.4. Определение строки без завершающего нуля

```
1 msg:
2 .ascii "Hello,␣world!\n"
3 len = . - msg // символу len присваивается длина строки
```

Для обработки подобных строк нужно знать их длину (её нельзя определить, анализируя содержимое памяти). Для этого используется специальный символ «.» — адрес текущего оператора (в том числе ячейки с данными).

Директивы определения данных точно так же сработают и будучи размещёнными в сегменте кода (`.text` или неименованная секция в начале программы), но такое размещение будет ошибкой. Если данные попадут во фрагмент кода, который выполняется, они будут интерпретированы как команды, что, скорее всего, приведёт к сбою при декодировании. Даже если данные находятся в той части кода, которая не получает управления, их будет невозможно модифицировать.

#### Прочие директивы

Парные директивы **.rept** ... **.endr** соответствуют повторению фрагмента между ними, в частности

#### Листинг 5.5. Директива `.rept`

```
1 .rept 4
2 .long 0
3 .endr
```

интерпретируется как четырёхкратное повторение оператора `.long 0`:

#### Листинг 5.6. Эквивалентное описание данных без использования `.rept`

```
1 .long 0
2 .long 0
3 .long 0
4 .long 0
```

Директивы `.rept ... .endr` могут применяться и к командам. При этом происходит именно многократное дублирование фрагмента кода в исполняемом файле, а не циклическое повторение одной и той же его копии.

Директива **`.globl (.global)`** делает символ доступным компоновщику (видимым из других модулей).

**Листинг 5.7.** Перевод символа *main* в глобальную область видимости

```
1 .globl main
```

Переменные и функции, имена которых сделаны общедоступными при помощи директивы `.globl` или `.global` (возможны оба написания), могут быть импортированы, в частности, в модуль на C++ с помощью ключевого слова `extern`.

Парные директивы **`.func ... .endfunc`** включают в исполняемый файл отладочную информацию о функции (только при сборке в отладочном режиме).

Таким образом, если в файле несколько функций, можно указать для отладчика их границы.

**Листинг 5.8.** Границы функции *sqr()*

```
1 .globl sqr
2 .func sqr
3     sqr:
4         movl 4(%esp), %eax
5         imull %eax, %eax
6         ret
7 .endfunc
```

Директивы `.func ... .endfunc` не влияют ни на что, кроме отладочной информации. В режиме Release игнорируются.

### 5.1.3. Порядок операндов

В GAS принят порядок записи операндов слева направо, следуя европейскому направлению письма. Соответственно, инструкция GAS обычно имеет вид

```
1 mnemonic source, destination
```

то есть вначале указывается источник, затем приёмник.

Для команд с тремя операндами (один из них в x86 — всегда непосредственное значение) вначале записывается непосредственное значение, затем источник, затем приёмник.

```
mnemonic immediate, source, destination
```

Если среди операндов нет приёмника (в частности, команды с двумя непосредственными операндами, такие как `enter`), порядок в AT&T совпадает с порядком, указанным в документации Intel [8].

При этом, если команда принимает операнды в регистрах или памяти, но не модифицирует их (в частности, команды сравнения `cmp` или выделения бита `btl`), в большинстве случаев один из операндов всё равно считается приёмником. Каждая из приведённых команд входит в семейство, большая часть команд которого модифицирует этот операнд.

В частности, рассмотрим одну из наиболее употребительных мнемоник ассемблера — команду пересылки (копирования) `mov`. Она соответствует оператору присваивания языков высокого уровня. Её операнды — источник (обозначим его *src*) и приёмник (*dest*). В синтаксисе Intel пересылка  $dest = src$  имела бы вид `mov dest, src`; в используемом в данном пособии синтаксисе AT&T она имеет вид `mov src, dest`. Например, команда `movb $0x05, %al` помещает значение 5 в регистр *al*.

#### 5.1.4. Адресация операндов

Параметры команд ассемблера, в отличие от операндов ЯВУ, не могут быть произвольными выражениями. В разделе 3.6.1 были описаны различные виды адресации. Рассмотрим обозначения, принятые в GAS для методов адресации явно передаваемых параметров команд x86.

1. **Непосредственная** — константа, значение которой при компиляции непосредственно включается в код команды (адрес глобальной переменной или

выражение, вычисляемое на этапе компиляции, также является непосредственным значением).

Непосредственные операнды отмечаются префиксом \$. Например, \$0, \$13, \$0xFFFFFFFF, \$(0 + 1 + 2\*2 - 7/4) (значение, равное 4), \$n (адрес глобальной переменной *n*).

2. **Прямая** (абсолютная) — переменная в памяти по фиксированному адресу (статическая или глобальная), адрес при компиляции также включается в код команды.

Операнды, описываемые статическим адресом в памяти, не имеют префикса. Например, 0 (вызовет ошибку чтения по нулевому адресу), *n* (значение глобальной переменной *n*).

3. **Прямая относительная**, также *rip*-relative или PC-relative, от program counter — переменная или функция располагается в памяти по фиксированному адресу, но в команду включается не сам адрес, а его смещение относительно указателя команд *ip*.

Для адресов в коде (функций, меток для перехода) неявно применяется по умолчанию как в тридцатидвухбитном, так и в шестидесятичетырехбитном режиме. Операнд не имеет префикса и в ассемблером коде выглядит так же, как и прямой абсолютный адрес: `call func, jmp label`.

В шестидесятичетырехбитном режиме появилась возможность использовать прямую относительную адресацию и для данных. Операнд выглядит как базовый адрес со смещением, где базой явно задан указатель команд: `lea msg(%rip), %rsi` (здесь `msg` — метка в секции данных). Относительная адресация используется только для базового регистра *rip*. Аналогичная запись с другой базой соответствует косвенной адресации и использованию *абсолютного*, а не относительного адреса `msg`.

4. **Регистровая** — переменная в регистре, в команду при компиляции включается имя (номер) регистра;

Операнды в регистрах отмечаются префиксом %. Например, %eax, %dh, %bp, %rsi, %r13.

5. **Косвенно-регистровая (косвенная)** — переменная в памяти, указатель на неё в регистре (или наборе регистров).

Например, (%ebp) — значение в памяти по адресу *ebp*. Можно указать смещение относительно адреса, хранимого в регистре: 4(%esp) — значение в памяти по адресу *esp* + 4.

В x86-совместимых процессорах для косвенной адресации можно использовать до двух регистров и, кроме того, до двух констант. Таким образом, для вычисления адреса используется до четырёх параметров: 4(%ebp, %edi, 8) — значение в памяти по адресу  $ebp + 8edi + 4$ . Часть параметров может отсутствовать: (%edx, %esi, 8) — значение по адресу  $edx + 8esi$ , (%edx, %esi) — по адресу  $edx + esi$ , (,%esi, 8) — по адресу  $8esi$ .

### Компоненты адреса

В языке ассемблера x86 конструкция косвенной адресации включает в себя вычисление адреса и его разыменование (частичным аналогом может быть оператор обращения к элементу массива на ЯВУ — [], а для сокращённых форм — разыменование указателя — \*, но при этом косвенная адресация — более сложный и гибкий механизм).

Как было сказано в разделе 3.6.3, команда x86 может содержать до четырёх полей, задающих адрес — номер базового регистра *Base*, номер индексного регистра *Index*, показатель масштаба индексного регистра *Scale* и смещение *Displacement*.

Используя для обозначения разыменования \*, как в C++, результат вычисления адреса с разыменованием в полной форме (с четырьмя параметрами) можно записать как

$$* (Base + \sigma \cdot Index + Displacement) \quad (5.1)$$

где *Base* и *Index* — значения соответствующих регистров (32-разрядные для соответствующей платформы), *Displacement* — целое знаковое число (смещение),  $\sigma$  — натуральное число (масштабный коэффициент — степень двойки  $2^{\text{Scale}}$ , причём только 1, 2, 4 или 8 из-за размера соответствующего поля в теле команды). Одно или оба числовых значения, а также любой из регистров могут быть опущены (если не указан масштаб, используется  $\sigma = 1$ , вместо остальных пропущенных параметров используется 0).

### Полная форма косвенной адресации

Полная форма косвенной адресации (вычисления адреса с разыменованием) в GAS, соответствующая (5.1), имеет вид

```
1 displacement(%base, %index,  $\sigma$ )
```

Любой из компонентов может отсутствовать, в этом случае опускается и соответствующий разделитель (только в одном случае — когда отсутствует база, но есть индекс — разделяющая их запятая сохраняется, чтобы отличать эту ситуацию от базы без индекса).

Таким образом, прямая адресация в принципе может рассматриваться как частный случай косвенной, когда оба регистра и масштабный коэффициент опущены вместе со скобками, и адрес равен смещению *Displacement*.

Например, следующая команда использует все четыре параметра и загружает в *A* значение  $*(bp + D \cdot 4 - 4)$  (команда `mov src, dest` загружает в приёмник *dest* значение источника *src*, *l* — суффикс размера).

```
1 movl    -4(%ebp, %edx, 4), %eax    // eax = *(ebp + 4edx - 4)
```

Чаще используются сокращённые варианты адресации, когда указывается только часть параметров [6, 8].



## Базовая косвенная адресация со смещением

Если используется только параметр *Base*, получим эквивалент разыменования указателя в C++. В частности, следующая команда записывает четырёхбайтовое значение по адресу *C* в регистр *D*.

```
1 movl (%ecx), %edx    // edx = *ecx
```

С параметрами *Base* и *Displacement* получим  $*(Base + Displacement)$ , что соответствует обращению к полю структуры (*Base* — адрес структуры, константа *Displacement* — относительное смещение нужного поля), к параметру функции или к локальной переменной. Следующая команда загружает значение из адреса *bp* − 4) в регистр *A*.

```
1 movl -4(%ebp), %eax   // eax = *(ebp - 4)
```

При передаче параметров функции через стек обратиться к ним внутри функции можно только используя адресацию относительно указателя стека *sp*. На вершине стека, то есть по адресу  $*sp = (\%esp)$ , находится адрес возврата. Под ним (но по большему адресу, так как стек растёт вниз) помещаются параметры.

## Базово-индексная косвенная адресация

При использовании всех параметров, кроме *Displacement*, получим  $*(Base + \sigma \cdot Index)$ , что соответствует обращению к элементу массива. Действительно, адрес элемента одномерного массива складывается из адреса начала массива, индекса элемента и размера элемента, то есть запись  $M[i]$  эквивалентна  $*(M + i \cdot sizeof(M[0]))$ . Если размер элемента равен 1, 2, 4 или 8, он может быть масштабным коэффициентом ( $\sigma$ ) и к элементу можно обратиться, используя три из четырёх параметров адреса:  $*(Base + Index \cdot \sigma)$ .

В частности, адрес *i*-го элемента массива *M* из чисел типа `int` равен  $M + i \cdot 4$ . Если адрес начала массива *M* находится в регистре *C*, а индекс — в *si*, то элемент  $M[i]$ , или  $*(M + i \cdot 4)$ , будет записан как  $(\%ecx, \%esi, 4)$ . Соответственно, запись  $M[i]$  в регистр *A* будет выглядеть следующим образом:

```
1 movl (%ecx, %esi, 4), %eax // eax = *(ecx + 4esi) = C[si]
```

Если размер элемента равен одному байту, адрес можно записать ещё компактнее:

```
movl (%ecx, %esi), %eax // eax = *(ecx + esi)
```

На платформах, отличных от x86 и x86-64, могут использоваться другие методы адресации и, соответственно, немного другие обозначения для них.

### 5.1.5. Размер операндов команды

Большая часть мнемоник соответствует не одной команде уровня архитектуры команд, а целому семейству однотипных команд, которые выполняют одни и те же действия над операндами **различного расположения или** различных размеров и, соответственно, имеют разные коды.

Размер операндов маркируется суффиксом, добавляемым к базовой форме мнемоники; например, копирование (базовая форма команды — `mov`) из `edx` в `eax` (размер операндов *long*) записывается как `movl %edx, %eax`. Суффиксы перечислены в таблице 5.1.

#### Суффиксы размера операндов

Таблица 5.1

Суффикс	Целый операнд (бит)	Вещественный операнд (бит)
b	byte (8)	
s	short (16)	single (32)
w	word (16)	
l	long (32)	double (64)
q	quad (64)	
t		ten bytes (80)

Необходимо отметить, что для целочисленных команд и команд обработки вещественных чисел одни и те же суффиксы означают различную разрядность.

Некоторые суффиксы допустимы только для одного семейства команд. Для целочисленных команд суффиксы *s* и *w* обозначают 16-битное целое и в целом равноправны (для команды `mov` используется только *w*, так как существует другая команда с базовой формой `movs`).

У команд с операндами разных размеров указывается два суффикса. Порядок суффиксов, как и порядок операндов — слева направо (от источника к приёмнику). Так, копирование целого числа со знаковым расширением (базовая форма команды — `movs`) из *al* в *edx* выглядит как `movsbl %al, %edx`. Возможны двойные суффиксы *bl* (от *byte* к *long*), *bw* (от *byte* к *word*) и *wl* (от *word* к *long*) и так далее. Суффикс *s* не используется как компонент составного суффикса (таблица 5.2).

### Двойные суффиксы размера для копирования целых чисел с расширением

Таблица 5.2

Суффикс	Источник (бит)		Приёмник (бит)
<i>bw</i>	byte (8)	→	word/short (16)
<i>bl</i>	byte (8)	→	long (32)
<i>bq</i>	byte (8)	→	quad (64)
<i>wl</i>	word/short (16)	→	long (32)
<i>wq</i>	word/short (16)	→	quad (64)
<i>lq</i>	long (32)	→	quad (64)

Если суффикс не указан, GAS определяет размер по регистровому операнду. Такое поведение несовместимо с оригинальным ассемблером AT&T Unix, который предполагает, что отсутствие суффикса означает размер операнда *long*. Эта несовместимость не влияет на компиляцию с ЯВУ, так как компиляторы всегда выставляют суффиксы размера [8].

Если размер не удалось определить по операндам (то есть используются либо непосредственные операнды, либо расположенные в памяти), по умолчанию для основного набора команд принимается размер *long* (32 бита). Такая ситуация не всегда вызывает ошибку компиляции, но в некоторых случаях приводит к странному результату.

При отсутствии суффикса размера у команды FPU для целых операндов, находящихся в памяти, используется размер *short* (16 бит), а для вещественных — одинарная точность (*float*, 32 бита), так что при работе с FPU надо быть особенно внимательным к суффиксам.

### 5.1.6. Мнемоники

Первоначальные мнемонические обозначений команд процессора предлагаются его разработчиками в документации, описывающей набор команд. В дальнейшем ассемблеры в основном используют именно их. Большая часть мнемоник GAS (их базовые формы) также совпадает с документацией Intel.

Тем не менее, часть обозначений различается. В частности, отличаются базовые формы команд копирования со знаковым расширением (*movs* в GAS, *movsx* в Intel) и копирования с беззнаковым расширением (*movz* в GAS, *movzx* в Intel); подробнее они описаны в разделе 5.2.4. Впрочем, для данных команд некоторые шестидесятичетырехбитные версии GAS используют обозначения Intel; для этих версий мнемоники *movs* и *movz* некорректны.

Синтаксис AT&T предлагает для команд знакового расширения (их неявным аргументом всегда является регистр *A*; подробнее в разделе 5.2.4) обозначения, построенные по схеме *cStD* (convert *S* to *D*), где *S* — суффикс размера источника, *D* — суффикс размера или обозначение расположения (в случае расширения в пару регистров) приёмника. В документации Intel приведены другие обозначения: они построены по схеме *cSD* или *cSDe* и используют другие обозначения размера. При программировании в GAS с использованием синтаксиса

AT&T можно без ограничений использовать для команд расширения регистра *A* как вариант AT&T, так и вариант Intel.

Опкоды дальнего перехода (с указанием сегмента и абсолютного адреса) в AT&T соответствуют мнемоникам `lcall/lret` (long call/long return) и `ljmp` (long jump), в то время как Intel обозначает их как `call far/retf` и `jump far`. Для этих команд GAS также поддерживает оба варианта.

Для несимметричных арифметических команд обработки чисел с плавающей точкой (`fsub/fsubr` и `fdiv/fdivr`) GAS использует те же мнемоники, что и Intel, но при этом реализует качественно иное поведение (раздел 5.3.5).

В шестидесятичетырехбитном режиме команда загрузки шестидесятичетырехбитного непосредственного значения в регистр обозначается `movabs` в синтаксисе AT&T (загрузка абсолютного адреса). Синтаксис Intel не выделяет единственную команду с шестидесятичетырехбитным непосредственным операндом из семейства команд пересылки и обозначает её как `mov`.

### 5.1.7. Префиксы

Как уже было сказано, регистры и непосредственные операнды обозначаются специальными префиксами. Для ассемблера x86 имена регистров начинаются с `%`, а непосредственные значения (константы) отмечаются префиксом `$`.

Дополнительно возможны префиксы `0x` для шестнадцатеричных констант, `0` для восьмеричных и `0b` для двоичных. Десятичные константы записываются без ведущих нулей, шестнадцатеричные и двоичные могут иметь ведущие нули после префикса.

Префикс непосредственного операнда `$` указывается перед префиксом системы счисления (`$0xFF`, `$0577`, `$0b101`).

На платформах, отличных от x86 и x86-64, могут использоваться другие префиксы для указания метода адресации операнда.

## 5.2. Основные команды

ЭТО не работает на восьмиразрядных машинах.

*Программистский фольклор*

Основной набор команд x86 включает команды обработки целых чисел и разнообразные команды управления вычислениями. Полный список команд приведён в первом томе руководства разработчика программного обеспечения для архитектур Intel 64 (так архитектура x86-64 называется в документации Intel) и IA-32 [16], а подробное описание — во втором [17]. Также набор инструкций описан в третьем томе руководства разработчика AMD [3].

Структура команды такова, что если у команды два операнда, они не могут оба находиться в памяти. Таким образом, если указано, что операнды могут быть переменными в памяти или регистрами, то возможны комбинации регистр-память, память-регистр и регистр-регистр.

Если не указано иное, используются следующие обозначения. Операнд, принимающий значение (приёмник) может быть обозначен либо как *dest*, если он может быть регистром или переменной в памяти, либо как *dreg* или *dmem*, если он может быть соответственно только регистром или только в памяти. Неизменяемый операнд (источник) может быть обозначен как *src* (регистр, переменная в памяти или непосредственное значение), *srcm* (регистр или переменная в памяти), *sreg* (регистр), *smem* (переменная в памяти). Непосредственно адресуемая константа обозначается как *imm*.

Для большинства команд источник и приёмник должны быть одного размера. Это может быть байт, два байта и четыре байта (для шестидесятичетырехбитных систем — до восьми байт).

Наиболее часто используемые команды доступны как в тридцатидвухбитном, так и в шестидесятичетырехбитном режимах (*mov*, *lea* и т. д.); некоторые —

только в шестидесятичетырехбитном (в частности, movabs). Некоторые команды тридцатидвухбитного режима недоступны в шестидесятичетырехбитном, но для аналогичных операций введены новые (syscall вместо sysenter).

Есть и такие команды, которые доступны в тридцатидвухбитном режиме, но полностью исключены из шестидесятичетырехбитного. В частности, это команды коррекции двоично-десятичной арифметики, проверка выхода за границы bound, условный вызов прерывания into и другие [70].

В данном пособии описывается только малая часть доступных команд x86. В частности, во избежание путаницы в описании не вошли команды, полностью исключённые из шестидесятичетырехбитного режима.

### 5.2.1. Общие команды

В таблице 5.3 приведены некоторые наиболее употребительные команды x86-совместимых процессоров.

#### Основные общие команды

Таблица 5.3

Команда	Действие
nop nop srm	Ничего не делает ( <i>no operation</i> )
mov src, dest	Присваивание $dest = src$ ( <i>move</i> )
movabs imm64, dreg64	В 64-битном режиме присваивание абсолютного 64-битного адреса $dreg64 = imm64$
lea smem, dreg	Вычисление адреса <i>smem</i> и запись его в <i>dreg</i> $dreg = \&smem$ ( <i>load effective address</i> )
xchg srm, dest	Обмен значений <i>srm</i> и <i>dest</i>
<b>Работа со стеком</b>	
push src	Помещение <i>src</i> в стек (уменьшает указатель стека)
pop dest	Вытаскивание значение из стека в <i>dest</i> (увеличивает указатель стека)

Наверное, самой популярной командой является команда пересылки `mov src, dest` — аналог оператора присваивания  $dest = src$  языков высокого уровня. Рассмотрим некоторые примеры её работы:

```
movl $4, %eax    // eax = 4
movb $42, %al    // al = 42
movl %eax, (%esi) // *esi = eax
movl %eax, 4(%esi) // *(esi+4 байта) = eax
movl $some_var, %eax // eax = &some_var
movl $some_var+4, %eax // eax = &some_var+4 байта
movl some_var, %eax // eax = some_var
movl %eax, foo // foo = eax
```

Аналогом оператора получения адреса (оператор `&` в C++) является команда `lea`. Если `mov smem, dreg` загружает в регистр *dreg* значение по адресу *smem*, то `lea smem, dreg` загружает в *dreg* сам адрес *smem*.

Например, следующая команда загружает в *A* значение  $*(bp + 4D - 4)$ , используя косвенную адресацию — вычисление адреса из четырёх компонент с разыменованием:

```
movl    -4(%ebp, %edx, 4), %eax
```

Команда `lea` загружает в приёмник адрес источника, что компенсирует разыменованное, то есть команда

```
leal    -4(%ebp, %edx, 4), %eax
```

загружает в *A* значение  $bp + 4D - 4$ .

Адрес статической переменной в тридцатидвухбитном режиме (то есть при использовании прямой абсолютной адресации) может быть загружен в регистр двумя способами — копированием адреса как тридцатидвухбитной константы или с помощью вычисления адреса:

```
movl $msg, %esi // si = &msg
leal msg, %esi  // si = &msg
```

Непосредственным операндом первой команды и смещением без базы во второй будет абсолютный тридцатидвухбитный адрес метки `msg`.



В шестидесятичетырёхбитном режиме предпочтительной является прямая относительная (*rip*-relative) адресация. В этом случае загрузка должна осуществляться командой `lea`:

```
lea msg(%rip), %rsi      // si = ip + (&msg - ip) = &msg
```

Смещением в соответствующей команде будет не абсолютный шестидесятичетырёхбитный адрес метки `msg`, а вычисленная ассемблером тридцатидвухбитная знаковая разность адреса `msg` и адреса следующей команды `ip`.

Использование в данной команде вместо *rip* любого другого регистра приведёт к тому, что в поле смещения будет записана не разность шестидесятичетырёхбитного адреса метки `msg` и значения базового регистра, а собственно адрес `msg` [61], причём усечённый до 32 бит:

```
lea msg(%rsp), %rsi      // si = sp + &msg
```

Таким образом, прямая относительная адресация не является частным случаем косвенной, хотя для их описания используется одна синтаксическая конструкция; эти виды адресации необходимо различать.

Если по какой-то причине требуется загрузить именно абсолютный шестидесятичетырёхбитный адрес как константу, это можно сделать командой `movabs`:

```
movabs $msg, %rsi        // si = &msg
```

Команда `mov` в шестидесятичетырёхбитном режиме может содержать не более чем тридцатидвухбитный непосредственный операнд.

Команда вычисления адреса `lea` также часто используются для арифметических вычислений, так как позволяет выполнить умножение и сложение за один шаг, а также выполняется намного быстрее, чем команды умножения `mul/imul`:

```
leal    8(,%eax,4), %eax    // A = A*4 + 8
leal    (%eax,%eax,2), %eax  // A = A*2 + A = A*3
```

В отличие от «настоящих» арифметических команд, `lea` не изменяет флагов.

## Работа со стеком

Для работы со стеком предназначены в основном команды `push` и `pop`. Они работают только с операндами размером 4 или 2 байта, то есть указатель вершины стека всегда выравнен на 2 байта (его начальное значение делается двоично-круглым). В GNU/Linux стек по соглашению выравнен по *long* (на 4 байта).

Команда `push src` помещает источник в стек. При этом указатель стека *sp* уменьшается на размер источника.

Таким образом, если попытаться смоделировать работу команды `push` при помощи команды пересылки, то, в частности `pushl $13` (здесь суффикс *l* = *long* необходим, так как разрядность операнда невозможно определить без явного указания) эквивалентна последовательному уменьшению *sp* и записи значения в память:

```
sub $4, %esp    // esp -= sizeof(long)
movl $13, (%esp) // *esp = 13
```

Комбинация команд изменения *sp* и пересылки неэффективна, так как она и выполняется медленнее, чем `push`, и занимает больше места в памяти. Тем не менее, иногда необходимо зарезервировать в стеке место для локальных переменных, начальное значение которых пока неизвестно. В этом случае можно воспользоваться командой `sub $size, %esp`.

Команда `pop dest` — извлечение значения из стека и помещение его в приёмник *dest* — увеличивает указатель стека *sp* на размер приёмника.

Таким образом, `popl %eax` можно также выполнить с помощью команд

```
1 movl (%esp), %eax // eax = *esp
2 add $4, %esp      // esp += sizeof(int)
```

Комбинация команд изменения *sp* и пересылки здесь так же менее эффективна, чем `pop`. При этом отдельная команда `add $size, %esp` для удаления элемента или набора элементов из стека «в никуда» используется очень часто. Она

быстрее, чем однократный вызов `ror`, так как не обращается к памяти; короче множественного вызова `ror`, а также не требует указания приёмника.

### 5.2.2. Передача управления, вызов и возврат

Команды передачи управления (таблица 5.4) делятся на две основные группы. Некоторые из них просто замещают указатель команд новым адресом, то есть передают управление новому фрагменту кода аналогично оператору `goto` языка C++. Другие перед передачей управления запоминают адрес следующей по счёту команды, так что затем можно вернуться к выполнению последовательности команд. Такие команды соответствуют вызовам разного рода подпрограмм (в том числе функций, прерываний, системных вызовов); каждой из них соответствует своя команда возврата, которая должна находиться в конце соответствующей подпрограммы.

#### Простая передача управления

К командам «безвозвратной» передачи управления относятся команда безусловного перехода `jmp src` и семейство команд условного перехода `jCC src`. Команды условного перехода отличаются только тем, что передача управления осуществляется только при наличии некоторой комбинации флагов *CC* в регистре *flags*.

Операнд команд передачи управления *src* может быть непосредственным значением (обычно меткой *label*), регистром или памятью. В первом случае неявно используется прямая относительная адресация, так что в соответствующий машинный код включается не абсолютный адрес *label*, а смещение относительно текущего указателя команд *label – ip*. Это позволяет получить переносимый код.

Если операнд команды перехода находится в регистре или памяти, то это — абсолютный адрес перехода. Использование такой адресации позволяет выбирать адрес перехода во время выполнения программы.

## Команды передачи управления, вызова и возврата

Таблица 5.4

Команда	Действие
<code>jmp src</code>	Безусловный переход по адресу <i>src</i> ( <i>goto src</i> )
<code>jcc src</code>	Условный переход по адресу <i>src</i> (если верно условие <i>src</i> )
<b>Вызов и возврат из функций</b>	
<code>call src</code>	Вызов подпрограммы — помещает в стек адрес следующей инструкции (адрес возврата) и переходит по адресу <i>src</i>
<code>ret [imm]</code>	Возврат из подпрограммы — снимает со стека адрес возврата и помещает его в указатель команд. Если указан параметр <i>imm</i> , снимает со стека ещё <i>imm</i> байтов.
<b>Вызов и возврат из программного прерывания</b>	
<code>int imm8</code>	Вызов прерывания с номером <i>imm8</i> — помещает в стек флаги <i>flags</i> , затем адрес возврата, после чего переходит к обработчику прерывания <i>imm8</i>
<code>iret</code>	Возврат из прерывания — снимает со стека адрес возврата и флаги, возвращает к прерванной программе
<b>Вызов и возврат из системного вызова (32 бита)</b>	
<code>sysenter</code>	Быстрый системный вызов
<code>sysexit</code>	Возврат из системного вызова
<b>Вызов и возврат из системного вызова (64 бита)</b>	
<code>syscall</code>	Быстрый системный вызов
<code>sysret</code>	Возврат из системного вызова

## Вызов и возврат

Команды вызова используются для передачи управления подпрограмме — последовательности команд, завершающихся командой возврата. Команды вызова и возврата всегда бывают парными.

Любая команда вызова сохраняет в определённом месте адрес той команды, которая следует за ней (адрес возврата), некоторые из них также сохраняют и другие данные (флаги, указатель стека и т. д.). Парная к ней команда возврата восстанавливает сохранённые элементы и помещает в указатель команд адрес

возврата. Таким образом, после выполнения подпрограммы управление перейдёт обратно к вызвавшей её программе и продолжится именно с той команды, которая следует за командой вызова.

Для вызова подпрограммы любого вида (функции или системного вызова) допустимо использовать только ту команду, которая является парной к завершающей эту подпрограмму команде возврата. В противном случае произойдёт крах.

Команды вызова и возврата не осуществляют передачу в подпрограмму параметров и возврат значения. Эти действия выполняются вручную и регламентируются соглашениями о вызовах (подробнее механизм вызова рассматривается в разделе 6.2).

В систему команд x86 входят три пары команд вызова/возврата.

1. Команды `call/ret` предназначены для вызова функций и процедур, описанных в самой программе и прикладных библиотеках.
2. Команды `int/iret` предназначены для программного обращения к прерыванию.

В современной прикладной программе явный вызов программного прерывания обычно используется только для обращения к ядру операционной системы (системного вызова, подробнее в разделе 6.2.8).

3. Команды, предназначенные специально для системных вызовов.

В тридцатидвухбитном режиме это `sysenter/sysexit`, в шестидесятичетырёхбитном — `syscall/sysret`.

## Вызов и возврат из функций

Вызов функции в ассемблере выполняется командой `call`. Эта команда имеет один операнд — адрес подпрограммы в памяти.

Команда `call foo` сохраняет указатель команд в стеке, управление передаётся `foo`. Возврат из подпрограммы выполняется командой `ret`, которая

должна находиться в конце кода подпрограммы. — управление передаётся адресу, снятому со стека.

В листинге 5.9 показана подпрограмма *foo()*, прибавляющая к значению регистра *eax* константу 5, а также её вызов.

### Листинг 5.9. Функция и её вызов

```
1 // Вызывающая программа
2  movl $10, %eax
3  call foo
4  // теперь %eax == 15
5  ...
6 // Функция foo()
7 foo:
8  addl $5, %eax
9  ret
10
```

Так же, как и для команд перехода *jmp/jcc*, если операндом команды *call* является метка *label*, неявно используется прямая относительная адресация (в машинный код включается смещение относительно текущего указателя команд *label — ip*), а если операнд в регистре или памяти — это абсолютный адрес перехода, задаваемый соответственно регистровой или косвенной адресацией.

Вызов подпрограммы по указателю применяется, в частности, для реализации механизма виртуальных функций, адрес которых выбирается из таблицы виртуальных методов на этапе выполнения программы.

### Вызов и возврат из прерывания

Для вызова прерывания необходимо указать его восьмибитный номер *imm8* (то есть номера прерываний могут принимать значения от 0 до 255). Каждому номеру прерывания соответствует специальный системный регистр, содержащий адрес обработчика этого прерывания.

Команда вызова прерывания *int imm8* помещает в стек сначала регистр флагов *flags*, затем адрес возврата. Если обработчик выполняется с привиле-

гиями ядра операционной системы, в специальном регистре сохраняется также указатель стека, так как система использует другой стек. После этого управление передаётся обработчику прерывания *imm8*.

Команда возврата из прерывания *iret* восстанавливает указатель стека (при необходимости), флаги и передаёт управление по адресу возврата (из стека при этом извлекаются оба помещённых туда командой *int* машинных слова).

Необходимо отметить, что в документации AMD команда вызова программного прерывания *int* относится к командам общего назначения, а соответствующая команда возврата *iret* — к системным [3].

Один из номеров прерываний в тридцатидвухбитных операционных системах обычно используется для системных вызовов.

### **Вызов и возврат из системных вызовов**

Начиная с Pentium II, доступна предложенная Intel команда *sysenter*, ускоряющая обращение к ядру. Адрес возврата и другие сохраняемые данные помещаются командой *sysenter* в специальные регистры, что быстрее обращения к памяти. В конце обработчика системного вызова их восстанавливает команда *sysexit*.

В шестидесятичетырехбитном режиме использование команды *sysenter* невозможно.

Для быстрого обращения к функциям ядра в шестидесятичетырехбитном режиме применяется команда *syscall*, предложенная AMD. Она также сохраняет данные для возврата в регистрах. Для возврата из системного вызова в шестидесятичетырехбитном режиме предназначена команда *sysret*.

Отметим, что команда возврата из системного вызова (*sysexit* или *sysret*) используется только в обработчике этого вызова, то есть в коде ядра операционной системы.

### 5.2.3. Обнуление регистра

Исторически для обнуления регистров использовались команды побитового исключающего «или» с одинаковыми операндами `xor %reg, %reg` и вычитания регистра из самого себя `sub %reg, %reg`, так как они выполнялись быстрее команды пересылки `mov $0, %reg`, а также занимали меньше места. Зависимость по данным в ранних моделях процессора не имела значения, так как вычисления не были конвейеризированы.

После введения конвейера традиция обнуления регистров командами `xor` и `sub` сохранилась. Поэтому в современных моделях процессоров команды обнуления регистров (zero idioms) распознаются при декодировании и выполняются как не имеющие зависимостей по данным.

Таким образом, сейчас руководство по оптимизации Intel [15] снова рекомендует использовать для обнуления регистров общего назначения команды:

```
1 xor %reg, %reg
2 sub %reg, %reg
```

Для регистров расширения XMM распознаются следующие команды обнуления:

```
1 xorps/pd %xmmreg, %xmmreg
2 pxor %xmmreg, %xmmreg
3 subps/pd %xmmreg, %xmmreg
4 psubb/w/d/q %xmmreg, %xmmreg
```

Для некоторых архитектур используются и другие команды обнуления регистров расширения X/Y/ZMM [15].

При этом команды `xor` и `sub`, не распознающиеся как zero idioms, выполняются медленнее из-за зависимости по данным (даже если результатом будет ноль).

### 5.2.4. Команды целочисленной арифметики

Из-за ограниченного количества операндов в системе команд x86 практически нет привычных по языкам высокого уровня неразрушающих арифметических операторов. Один из операндов, как правило, используется и как исходное дан-



ное, и как ячейка для записи результата. В частности, аналогом ассемблерной команды сложения `add src, dest` в C++ будет не оператор «плюс», не изменяющий свои операнды ( $dest = src_1 + src_2$ ), а оператор «+=» ( $dest += src$ , то есть  $dest = dest + src$ ).

Некоторые команды, предназначенные для обработки целых чисел, перечислены в таблице 5.5.

### Команды сложения и вычитания

К этой группе, кроме собственно сложения и вычитания, относятся также команды инкремента, декремента и изменения знака.

Как было сказано в разделе 2.5, представление отрицательных чисел выбиралось так, чтобы знаковые числа можно было складывать и вычитать с помощью беззнакового сумматора. Соответственно, операции сложения и вычитания не делятся на знаковые и беззнаковые. Команды из группы сложения и вычитания выставляют значения всех шести флагов состояния соответственно результату, так что программист, понимая, какого рода числа он обрабатывает, может выбрать для анализа нужные флаги.

Числа, разрядность которых превышает разрядность системы, при необходимости можно складывать и вычитать по частям. Для этого вначале младшие части обрабатываются командами `add/sub`, затем к остальным в порядке возрастания адресов — `adc/sbb`, учитывающие перенос из младшей части. Части могут иметь любую разрядность (в частности, шестибайтовые целые можно разбить на две части — четыре и два байта или на шесть однобайтовых), но логичнее использовать четырёхбайтовые части на тридцатидвухбитной системе и восьмибайтовые — на шестидесятичетырёхбитной.

### Вычисление линейной комбинации регистров

Также для арифметических вычислений используется команда `lea`, которая, в соответствии с возможностями косвенной адресации, может рассчитать выражение  $r1 + \sigma \cdot r2 + \delta$  (может быть опущен любой из регистров и любая из

## Команды целочисленной арифметики

Таблица 5.5

Команда	Действие
<b>Сложение и вычитание</b>	
inc dest	Инкремент $++dest$ ( $dest = dest + 1$ , выполняется быстрее add)
dec dest	Декремент $--dest$ ( $dest = dest - 1$ , выполняется быстрее sub)
add src, dest	Сложение $dest += src$ ( $dest = dest + src$ )
adc src, dest	Сложение с переносом из предыдущей части $dest += src + CF$ ( $dest = dest + (src + CF)$ )
sub src, dest	Вычитание $dest -= src$ ( $dest = dest - src$ )
cmp src, dest	Вычитание $dest - src$ без изменения $dest$ (только флаги)
sbb src, dest	Вычитание с переносом из предыдущей части $dest -= src + CF$ ( $dest = dest - (src + CF)$ )
neg dest	Изменение знака $dest = -dest$
<b>Расчёт линейной комбинации</b>	
lea $\delta(r1, r2, \sigma)$ , dreg	Вычисление эффективного адреса часто используется для расчёта линейной комбинации $dreg = r1 + \sigma \cdot r2 + \delta$ $dreg, r1, r2$ — регистры, $\sigma$ — 1, 2, 4 или 8, $\delta$ — константа
<b>Умножение и деление</b>	
mul srm	Беззнаковое умножение $D:A = A \cdot srm$ (таблица 5.6)
imul srm	Знаковое умножение $D:A = A \cdot srm$ (таблица 5.6)
imul srm, dreg	Умножение $dreg *= srm$ ( $dreg = dreg \cdot srm$ )
imul imm, srm, dreg	Знаковое умножение $dreg = imm \cdot srm$
div srm	Беззнаковое деление с остатком $\begin{cases} A = (D:A)/srm \\ D = (D:A)\%srm \end{cases}$ (таблица 5.6)
idiv srm	Знаковое деление с остатком $\begin{cases} A = (D:A)/srm \\ D = (D:A)\%srm \end{cases}$ (таблица 5.6)
<b>Масштабирование (битовый сдвиг)</b>	
shr times, dest	Беззнаковое деление $dest / = 2^{times}$ (остаток не вычисляется)
sar times, dest	Знаковое деление $dest / = 2^{times}$ (остаток не вычисляется и предполагается неотрицательным, см. раздел 2.7.3)
shl times, dest sal times, dest	Умножение $dest *= 2^{times}$

констант). Команда `lea` предназначена для манипуляций с беззнаковыми данными (указателями), но смещение  $\delta$  интерпретируется как знаковое. Так как разрядность  $r1$  и  $r2$  совпадает с разрядностью  $dreg$ , результат совпадает со знаковым.

В отличие от «настоящих» арифметических команд, `lea` не изменяет флагов.

### Команды умножения и деления

Самые старые команды умножения рассчитывают произведение заданного множителя  $srm$  на неявный операнд — регистр  $A$  той же разрядности, что и  $srm$ . Разрядность произведения при этом вдвое больше разрядности множителей, так что младшая половина  $A \cdot srm$  помещается в регистр  $A$  на место неявного множителя, а старшая — в регистр  $D$  той же разрядности. Исключением является случай с восьмибитными множителями — так как на момент появления команд умножения уже существовали шестнадцатибитные регистры, результат  $al \cdot srm$  размещается в  $ax$  (таблица 5.6). Старшая половина результата отличается для знаковой и беззнаковой интерпретации множителей, так что существуют две команды описанного действия — `mul` для беззнакового умножения и `imul` для знакового.

### Команды умножения и деления неявного аргумента $A$

Таблица 5.6

Размер $srm$	Действие <code>[i]mul srm</code>	Действие <code>[i]div srm</code>
4 байта	$edx : eax = eax \cdot srm$	$\begin{cases} eax = (edx : eax) / srm \\ edx = (edx : eax) \% srm \end{cases}$
2 байта	$dx : ax = ax \cdot srm$	$\begin{cases} ax = (dx : ax) / srm \\ dx = (dx : ax) \% srm \end{cases}$
1 байт	$ax = al \cdot srm$	$\begin{cases} al = ax / srm \\ ah = ax \% srm \end{cases}$

Введённая позже команда `imul srm, dreg` рассчитывает только младшую половину произведения, а она совпадает для знаковых и беззнаковых чисел. Таким образом, двухоперандную форму команды `imul` можно использовать и для знакового, и для беззнакового умножения.

Трёхоперандная форма `imul imm, srm, dreg` также рассчитывает только младшую половину произведения, но перед этим константа `imm` при необходимости расширяется. Данная форма соответствует двум опкодам — с константой `imm`, разрядность которой соответствует разрядности источника и приёмника (в этом случае расширение не требуется) и с восьмибитной константой `imm8` [17]. Во втором случае `imm8` расширяется как знаковое, поэтому трёхоперандную форму `imul` следует считать командой знакового умножения.

Если произведение помещается в младшей половине произведения, все формы команд `mul/imul` сбрасывают оба флага *CF* и *OF*. Если в старшей половине есть значащие биты (для двух- и трёхоперандной форм `imul` это значит, что результат некорректен), оба этих флага взводятся [53]. Значения флагов нуля и знака не определены [71].

Для деления существуют только однооперандная форма. Делимое (неявный операнд) всегда вдвое больше делителя (явного операнда *srm*) и располагается в паре регистров  $D : A$  (старшая половина — в *D*, младшая — в *A*), кроме случая восьмибитного делителя (таблица 5.6). Необходимо помнить об этом и корректно инициализировать регистр *D* перед делением.

Таким образом, команды деления обратны однооперандной форме умножения. Соответственно, деление также будет беззнаковым (`div`) и знаковым (`idiv`).

После деления  $D : A$  на *srm* частное помещается на место младшей половины делимого (в *A*), остаток — на место старшей половины (в *D*). Если старшая половина делимого содержит значащие биты, возможна ситуация, когда част-

ное не помещается в отведённом для него регистре. Соответственно, результат деления будет некорректным.

Умножение и деление на  $2^{times}$  (*times* трактуется как беззнаковое число) может также быть выполнено с помощью битовых сдвигов (раздел 5.2.5).

## Расширение целых чисел

Также к командам целочисленной арифметики можно отнести команды расширения (таблица 5.7).

### Команды расширения (увеличения разрядности)

Таблица 5.7

Команда	Действие
<code>movz srm, dreg</code>	$dreg = srm$ с беззнаковым расширением (размер <i>srm</i> меньше <i>dreg</i> )
<code>movs srm, dreg</code>	$dreg = srm$ со знаковым расширением (размер <i>srm</i> меньше <i>dreg</i> )
<code>cStD</code>	Знаковое расширение регистра <i>A</i> (таблица 5.8)

Современная система команд x86 включает два вида команд, которые используются для расширения — пересылка из источника малой разрядности в приёмник большей и удвоение разрядности неявного операнда в регистре *A*.

Существует две команды пересылки с расширением — `movz` для беззнакового расширения (дополнения нулями) и `movs` для знакового (дополнения знаковым битом).

При пересылке возможно увеличение разрядности более чем в два раза, поэтому, если источник находится в памяти, для команды обязательно нужно указывать два суффикса (раздел 5.1.5).

Также существует набор команд для удвоения разрядности неявного операнда в регистре *A* (таблица 5.8).

Практически для всех случаев есть две команды — расширение *A* до *A* вдвое большей разрядности и расширение *A* до пары *D* : *A*. Последний вариант

Мнемоники команд знакового расширения  $A$ 

Таблица 5.8

Размер	Расширение в регистр $A$			Расширение в пару $D : A$		
	Действие	Intel	AT&T	Действие	Intel	AT&T
$8 \rightarrow 16$	$al \rightarrow ax$	cbw	cbtw	—		
$16 \rightarrow 32$	$ax \rightarrow eax$	cwde	cwtl	$ax \rightarrow dx : ax$	cwd	cwtd
$32 \rightarrow 64$	$eax \rightarrow rax$	cdqe	cltq	$eax \rightarrow edx : eax$	cdq	cltd
$64 \rightarrow 128$	—			$rax \rightarrow rdx : rax$	cqo	cqto

необходимо использовать перед командами, использующими пару регистров  $D : A$  как источник, в частности, командами деления  $(D : A)/srn$ .

Для команд удвоения разрядности регистра  $A$  есть два набора мнемоник — исторически используемые в ассемблере Unix (AT&T) и предложенные Intel.

Мнемоники AT&T для команд удвоения разрядности регистра  $A$  строятся по схеме  $cStD$  (convert  $S$  to  $D$ ), где  $S$  — размер источника,  $D$  — размер приёмника или символ  $d$  для пары регистров (кроме `cqto`).

Мнемоники Intel построены по одной из двух схем —  $cSD$  или  $cSDe$ . Исторически на шестнадцатибитных машинах первыми доступными вариантами удвоения были  $al \rightarrow ax$  и  $ax \rightarrow dx : ax$ , так что они получили имена без суффикса  $e$ . В дальнейшем суффикс  $e$  использовался для расширения  $A \rightarrow A$ , а расширение  $A \rightarrow D : A$  выполняется командой без суффикса.

Для команд удвоения разрядности  $A$  GAS поддерживает оба набора мнемоник — AT&T и Intel — перечисленные в таблице 5.8.

Увеличить разрядность неявного операнда  $A$  более чем в два раза с помощью команд таблицы 5.8 можно только последовательным применением нескольких команд удвоения.

Все команды удвоения разрядности  $A$  выполняют знаковое расширение. Беззнаковое расширение  $A \rightarrow D : A$  может быть выполнено явным обнулением регистра  $D$ .

### 5.2.5. Битовые операции

Некоторые команды, предназначенные для обработки битовых строк, перечислены в таблице 5.9.

Система команд x86 включает поразрядные логические операции, все описанные в разделе 2.7 битовые сдвиги, а также команды выделения бита по номеру.

Поразрядные логические операции «и», «или», «не» и «исключающее или» изменяют в соответствии с полученным результатом три флага состояния — флаги нуля, знака и чётности.

Команды битового сдвига принимают два операнда: сдвигаемое значение  $dest$  и беззнаковое количество сдвигов  $times$ . Количество  $times$  может быть непосредственным значением или регистром  $cl$ , причём даже в шестидесятичетырёхбитных системах используются только младшие шесть его бит (в тридцатидвухбитных — пять).

При  $times = 1$  происходит сдвиг в указанную сторону на один бит. Значение бита, вышедшего за разрядную сетку, после выполнения команды заносится в  $CF$ . Освободившаяся с другого конца  $dest$  ячейка инициализируется в соответствии с видом сдвига.

При  $times > 1$  однобитовый сдвиг повторяется  $times$  раз.

Знаковый и беззнаковый сдвиги используются для быстрого умножения и деления на степени двойки. Сдвиги вправо эквивалентны соответственно знаковому (в этом случае остаток предполагается неотрицательным как для положительных, так и для отрицательных делимых) или беззнаковому делению на  $2^{times}$ . В случае однобитового сдвига (деления на два) остаток попадает в  $CF$ . Для больших значений сдвига остаток не вычисляется.

## Основные битовые операции

Таблица 5.9

Команда	Действие
<b>Поразрядные операции</b>	
not dest	Побитовая инверсия $dest = \sim dest$
and src, dest	Побитовое «и» $dest \&= src$ ( $dest = dest \& src$ )
test src, dest	Побитовое «и» $dest \& src$ без изменения $dest$ (только флаги)
or src, dest	Побитовое «или» $dest  = src$ ( $dest = dest   src$ )
xor src, dest	Побитовое «исключающее или» $dest \hat{=} src$ ( $dest = dest \hat{src}$ )
<b>Битовые сдвиги</b>	
shr times, dest	Беззнаковый (логический) сдвиг вправо $dest = (unsigned)dest >> times$ Освободившиеся старшие разряды заполняются нулями, младшие теряются, кроме последнего, который попадает в $CF$
sar times, dest	Знаковый (арифметический) сдвиг вправо $dest = (signed)dest >> times$ Освободившиеся старшие разряды заполняются знаковым битом, младшие теряются, кроме последнего, который попадает в $CF$
shl times, dest sal times, dest	Сдвиг влево (shl и sal — синонимы) $dest = dest << times$ Освободившиеся младшие разряды заполняются нулями, старшие теряются, кроме последнего, который попадает в $CF$
ror times, dest	Циклический сдвиг $dest$ вправо
rol times, dest	Циклический сдвиг $dest$ влево
rcr times, dest	Циклический сдвиг через флаг переноса $dest \cup CF$ вправо
rcl times, dest	Циклический сдвиг через флаг переноса $dest \cup CF$ влево
<b>Загрузка <math>idx</math>-го бита числа во флаг <math>CF</math></b>	
bt idx, dest	$CF = dest[idx]$
btc idx, dest	$CF = dest[idx]$ с последующей инверсией бита $dest[idx] = \neg dest[idx]$
btr idx, dest	$CF = dest[idx]$ с последующим сбросом бита $dest[idx] = 0$
bts idx, dest	$CF = dest[idx]$ с последующей установкой бита $dest[idx] = 1$



Сдвиг влево эквивалентен умножению на  $2^{times}$ , если результат умножения помещается в *dest*. Если не помещается, старшая часть произведения теряется.

Кроме того, с помощью команд семейства *btX* можно выделить отдельный бит числа. Эти команды принимают два операнда — число-приёмник, один бит которого будет скопирован в флаг *CF* и затем изменён, и номер бита *idx* — непосредственное значение или регистр. Для младшего бита  $idx = 0$ . Одна из команд семейства, *bt*, не изменяет значение битов в числе, но для единообразия её операнд, из которого выделяется бит, также считается приёмником.

### 5.2.6. Флаги

Все арифметические команды устанавливают по результатам вычислений флаги состояния.

Команды группы сложения/вычитания (*add/sub*, *inc/dec* и т. д.) выставляют все шесть флагов состояния *CF*, *PF*, *AF*, *ZF*, *SF*, *OF* в соответствии с результатом:

- флаг нуля *ZF*, если результат равен нулю;
- флаг переноса (беззнакового переполнения) *CF* в случае переноса/заёма за пределы разрядной сетки (беззнакового переполнения);
- флаг знака *SF*, если старший (знаковый) бит результата равен 1;
- флаг знакового переполнения *OF*, если произошло знаковое переполнение (перенос/заём из знакового бита, но не за пределы разрядной сетки, или наоборот);
- флаг чётности *PF*, если количество единиц в младшем байте результата чётно;
- флаг вспомогательного переноса *AF*, если в младшем байте был перенос между тетрадами.

При этом *add \$-1, dest* и *sub \$1, dest* устанавливают флаги по-разному, в частности, при сложении числа  $-1$  (что на 32-разрядной платформе равно

0xFFFFFFFF) с нулём не происходит переноса в старший бит ( $OF = 0$ ); при вычитании единицы из нуля возникает заём из старшего бита ( $OF = 1$ ).

Побитовые команды (`and`, `or`, `xor`) выставляют флаги  $SF$ ,  $ZF$  и  $PF$  в соответствии с результатом аналогично группе сложения/вычитания, флаги переноса и знакового переполнения сбрасываются:  $CF = OF = 0$ . Значение флага  $AF$  не определено.

Команды умножения выставляют флаги  $CF = OF$  в зависимости от того, выходит ли результат за разрядность множителей. Значение остальных флагов не определено. После команд деления все шесть флагов имеют неопределённое значение.

Существуют команды, которые только выставляют флаги и не меняют значения своих операндов. Они предназначены для сравнения чисел. Это:

- `cmp` — то же самое, что и `sub` (группа сложения/вычитания), но операнд-приёмник не изменяется (используется для сравнения целых чисел);
- `test` — то же самое, что и `and` (группа побитовых операций), но операнд-приёмник не изменяется (используется для сравнения битовых строк).

Основной набор инструкций x86 не содержит команд для обработки и, в частности, сравнения вещественных чисел. Предназначенные для этого инструкции сравнения относятся к набору команд FPU (раздел 5.3.7), но могут взаимодействовать с регистром *flags*. Вещественные числа можно сравнить командой `fcom` и подобными ей. После сравнения флаги состояния сопроцессора копируются в *flags* (вручную или автоматически — в зависимости от используемой команды сравнения) таким образом, что результат сравнения можно анализировать так же, как для целых беззнаковых чисел:  $ZF$  указывает на равенство,  $CF$  — на  $dest < src$ ; кроме того, в  $PF$  копируется флаг несравнимости операндов.

Кроме того, флаги можно установить или сбросить вручную с помощью специальных команд или загрузив изменённый регистр *flags* (таблица 5.10).

## Команды обработки флагов

Таблица 5.10

Команда	Действие
<b>Установка отдельных флагов</b>	
stc/clc/cmc	Установка (set, $CF = 1$ )/сброс (clear, $CF = 0$ )/инверсия ( $CF = ! CF$ ) флага переноса $CF$
std/cld	Установка/сброс флага направления $DF$
sti/cli	Установка/сброс флага разрешения прерываний $IF$
<b>Обработка <math>flags</math> в целом или фрагментов</b>	
lahf/sahf	Сохранение младшего байта $flags$ в $ah$ /загрузка $ah$ в $flags$
pushf/pushfd/pushfq	Загрузить младшие 16/32/64 бита $flags$ в стек
popf/popfd/popfq	Выгрузить 16/32/64 бита из стека в $flags$ (в младшую часть)

Младший байт регистра флагов, содержащий большую часть флагов состояния, можно загрузить для анализа в регистр  $ah$  командой `lahf` (*Load Flags into AH Register*). Обратная операция выполняется командой `sahf` (*Store AH into Flags*). Регистр  $flags/eflags$  можно полностью поместить в стек командами `pushf/pushfd`, загрузить из стека — командами `popf/popfd` соответственно. При загрузке флагов из  $ah$  или стека зарезервированные биты не загружаются в  $flags$ .

### 5.2.7. Условные команды

Существует несколько семейств команд, действие которых определяется значением флагов состояния в регистре  $flags$ . Это команды условного перехода `jcc`, условной установки байта `setcc` и условной пересылки `movcc` и `f movcc`.

Команда `f movcc` относится к набору команд FPU и описана также в разделе 5.3.4 (таблица 5.15), но условие  $CC$  определяется значением регистра  $flags$ , а не собственного регистра состояния FPU.

Мнемоники таких команд состоят из двух частей — общего для всех команд семейства обозначения действия (j от *jump*, *set* и т. п.) и обозначения условия *CC*. Условие не может быть произвольным. Существует определённый набор обозначений *CC*, каждому из которых соответствует некоторое состояние флагов в регистре *flags* — условие.

### Условия

Рассмотрим доступные варианты условий, их обозначения и связь с арифметическими операциями (таблица 5.11).

Каждое из условий имеет некоторое буквенное обозначение *CC*, приведённое в первом столбце. Одно и то же условие может обозначаться по-разному, в частности, «меньше или равно» — *le* и «не больше» — *ng*, но машинный код в таких случаях одинаков. Различные обозначения одного условия помещены в одну ячейку таблицы и разделяются косой чертой.

Мнемоники одного семейства с разными условиями, в частности, *e* и *l*, ассемблируются в разные машинные коды.

Условие складывается из некоторой комбинации флагов состояния в регистре *flags*, указанной во втором столбце. Эти флаги определяются результатом последней команды. Различные виды арифметических команд выставляют их в соответствии с полученным результатом. В частности, команды группы сложения/вычитания (включая часто используемые для реализации цикла команды *inc/dec*) изменяют все шесть флагов состояния; побитовые команды — флаги *SF*, *ZF* и *PF*. В третьем столбце таблицы указаны свойства результата, приводящие к подобному сочетанию флагов.

Часто перед условной командой вызывается команда сравнения *cmp src, dest*, выставляющая флаги аналогично команде *sub src, dest* (то есть её результатом будет *dest - src*), но не изменяющая *dest*. В четвёртом столбце указаны соотношения между *dest* и *src* для каждого условия. Большинство обозначений условий образовано именно от них.

Условия и их связь с флагами состояния *flags*

Таблица 5.11

CC	Условие (флаги)	Условие (арифметика)	Условие (sub src, dest или cmp src, dest)
e/z	$ZF = 1$	Результат равен нулю <i>if zero</i>	$dest = src$ $src = dest$ <i>if equal</i>
ne/nz	$ZF = 0$	Результат не равен нулю <i>if not zero</i>	$dest \neq src$ $src \neq dest$ <i>if not equal</i>
c/ b/nae	$CF = 1$	Есть беззнаковое переполнение <i>if carry</i>	$dest < src$ как беззнаковое $src > dest$ <i>if below / if not above or equal</i>
be/na	$\begin{cases} CF = 1 \\ ZF = 1 \end{cases}$	Есть беззнаковое переполнение или результат равен нулю	$dest \leq src$ как беззнаковое $src \geq dest$ <i>if below or equal / if not above</i>
nc/ nb/ae	$CF = 0$	Нет беззнакового переполнения <i>if not carry</i>	$dest \geq src$ как беззнаковое $src \leq dest$ <i>if not below / if above or equal</i>
nbe/a	$\begin{cases} CF = 0 \\ ZF = 0 \end{cases}$	Нет беззнакового переполнения и результат не равен нулю	$dest > src$ как беззнаковое $src < dest$ <i>if not below or equal / if above</i>
s	$SF = 1$	Старший (знаковый) бит результата равен 1	<i>if sign</i>
ns	$SF = 0$	Старший (знаковый) бит результата равен 0	<i>if not sign</i>
o	$OF = 1$	Есть знаковое переполнение	<i>if overflow</i>
no	$OF = 0$	Нет знакового переполнения	<i>if not overflow</i>
l/nge	$SF \neq OF$	Результат отрицателен (знак равен 1 и корректен или равен 0, но некорректен)	$dest < src$ как знаковое $src > dest$ <i>if less / if not greater or equal</i>
le/ng	$\begin{cases} SF \neq OF \\ ZF = 1 \end{cases}$	Результат отрицателен или равен нулю	$dest \leq src$ как знаковое $src \geq dest$ <i>if less or equal / if not greater</i>
nl/ge	$SF = OF$	Результат неотрицателен (знак равен 0 и корректен или равен 1, но некорректен)	$dest \geq src$ как знаковое $src \leq dest$ <i>if not less / if greater or equal</i>
nle/g	$\begin{cases} SF = OF \\ ZF = 0 \end{cases}$	Результат неотрицателен и не равен нулю	$dest > src$ как знаковое $src < dest$ <i>if not less or equal / if greater</i>
p/pe	$PF = 1$	Число единичных бит младшего байта результата чётно <i>if parity / if parity even</i>	
np/po	$PF = 0$	Число единичных бит младшего байта результата нечётно <i>if not parity / if parity odd</i>	
u	$PF = 1$	$src$ и $dest$ несравнимы (только для fcmovCC)	<i>if unordered</i>
nu	$PF = 0$	$src$ и $dest$ сравнимы (только для fcmovCC)	<i>if not unordered</i>

Для беззнаковых операндов  $dest$  и  $src$  признаком отрицательности результата  $dest - src$  (то есть соотношения  $dest < src$ ) будет заём в старший бит при вычитании, то есть беззнаковое переполнение  $CF = 1$ . Равенство операндов достигается при  $dest - src = 0$ , что отмечается флагом нуля  $ZF = 1$ . Соответственно,  $dest > src$  может быть, если  $dest \not< src$  и при этом  $dest \neq src$ , то есть  $CF = ZF = 0$ . Таким образом, любое соотношение между беззнаковыми операндами можно выразить через флаги  $CF$  и  $ZF$ .

Чтобы отличить условия знакового и беззнакового сравнения, для беззнакового вместо термина «меньше» часто используется термин «ниже» (*below*), вместо «больше» — «выше» (*above*).

Для знаковых арифметических операций знак результата невозможно определить только по знаковому биту (флагу  $SF$ ), так как этот бит может быть искажён знаковым переполнением. То есть если знак равен единице ( $SF = 1$ ), но в процессе вычислений произошло знаковое переполнение ( $OF = 1$ ), то знаковый бит неверен и результат на самом деле положителен. Таким образом, результат будет отрицательным в двух случаях:  $\begin{cases} SF = 1 \\ OF = 0 \end{cases}$  (знаковый бит — единица и переполнения не было) и  $\begin{cases} SF = 0 \\ OF = 1 \end{cases}$  (было переполнение и знаковый бит — ноль). Обычно это условие записывается в виде  $SF \neq OF$ . Соответственно, при  $SF = OF$  результат неотрицателен.

После сравнения вещественных чисел флаг вещественной несравнимости выгружается в бит  $PF$  регистра *flags*, поэтому для команд условной пересылки вещественных чисел (и только для них) условие  $PF = 1$  записывается как *u*.

### Условные и безусловные переходы

В системе команд x86, а соответственно, и в языке ассемблера, нет операторов, аналогичных операторам C++ *if*, *while*, *for* и т. п. Ветвления и циклы реализуются при помощи команд условного и безусловного перехода [53].

В таблице 5.12 приведены эти команды.

## Команды передачи управления

Таблица 5.12

Команда	Действие
<code>jmp label</code>	Безусловный переход ( <i>goto</i> )
<code>jcc label</code>	Переход по адресу <i>label</i> , если верно условие <i>CC</i> (кроме <i>u</i> и <i>nu</i> )

Безусловный переход `jmp` является аналогом оператора `goto` языка C++ — передаёт управление команде по адресу *label*.

Команды условного перехода `jcc` передают адресу *label* при выполнении какого-либо условия (чаще всего при определённой комбинации флагов в регистре *flags*). Если условие не выполнено, `jcc` не делает ничего, и выполняется команда, следующая за `jcc` по тексту программы.

Условие *CC* может быть любым из перечисленных в таблице 5.11, кроме *u* и *nu* (но могут использоваться *p/pe* и *np/ro*). В частности, команда `jnae label` (*jump if not above or equal*) передаст управление на метку *label* в случае  $CF = 1$ . Если перед командой условного перехода выполнялась команда `cmp src, dest`, управление будет передано на метку *label* в случае, если  $dest \not\geq src$  как беззнаковые числа.

Это условие эквивалентно  $dest < src$ . Действительно, команды `jnae` и `jb` (*jump if below*) имеют один и тот же опкод. Также этот опкод соответствует мнемонике `jc` (*jump if carry*).

Кроме того, в набор инструкций современных процессоров входят унаследованные от Intel 8086 команды псевдоцикла `loop`, псевдоцикла с анализом флага нуля `loope/loopz` и `loopne/loopnz`, а также такие команды условного перехода, как `jsxz` и `jesxz` (переход, если регистр *cx/ecx* равен нулю). По своему действию команда `loop label` эквивалентна командам `dec %ecx; jz label`, при этом `loop` не меняет флаги *flags*. В случае команд `loope/loopz`

и `loopne/loopnz` анализируется не только  $cx/ecx$ , но и флаг нуля  $ZF$  (управление на метку передаётся, если  $\begin{cases} cx \neq 0 \\ ZF = 1 \end{cases}$  и  $\begin{cases} cx \neq 0 \\ ZF = 0 \end{cases}$  соответственно).

Эти команды были введены в набор 8086 для получения более компактного кода (однобайтовая инструкция `loop` заменяет связку двух однобайтовых `dec+jz`) и экономии дорогой памяти. При этом пара инструкций `dec+jz` выполняется быстрее `loop` [10, 28], легче читается и позволяет организовывать вложенные циклы. В настоящее время оптимизация направлена на ускорение, соответственно, использования команд псевдоцикла `loop`, `loopne/loopnz`, `loopne/loopnz` (а также команд `jsxz` и `jesxz`) следует избегать.

### Условная пересылка

Для каждого условия  $CC$ , кроме команды условного перехода `jCC`, существует команда условной пересылки `cmovCC src, dest`, выполняющая присваивание  $dest = src$ , если соответствующее условие верно.

В таблице 5.13 показаны различные команды безусловной и условной пересылки.

### Команды пересылки данных

Таблица 5.13

Команда	Действие
<code>mov src, dest</code>	Присваивание $dest = src$
<code>movs srm, dreg</code> <code>movz srm, dreg</code>	Присваивание $dreg = srm$ с расширением
<code>cmovCC srm, dreg</code>	Присваивание $dreg = srm$ , если верно $CC$ (кроме <code>u</code> и <code>nu</code> )
<code>fcmovCC %st(i), %st(0)</code>	Присваивание регистров FPU $st(0) = st(i)$ , если верно $CC$ ( <code>e</code> , <code>ne</code> , <code>b/nae</code> , <code>be/na</code> , <code>ae/nb</code> , <code>a/nbe</code> , <code>u</code> и <code>nu</code> )
<code>setCC dest8</code>	Присваивание $dest8 = \begin{cases} 1, & \text{если верно } CC \text{ (кроме } u \text{ и } nu) \\ 0, & \text{иначе} \end{cases}$



Команды условной пересылки не полностью аналогичны `mov`: источник может быть только регистром или в памяти, приёмник — только регистром. Пересылаемое значение не может иметь размер 8 бит.

Для флагов, которые могут быть установлены командами сравнения FPU (*ZF*, *CF*, *PF*) существует также команда условной пересылки в стеке FPU из *st(i)* в *st(0)* `fcmovCC %st(i), %st(0)` (раздел 5.3.4, таблица 5.16).

### Установка байта по условию

Для каждого условия *CC* существует команда установки байта по условию `setCC dest8`, выполняющая присваивание  $dest8 = 1$ , если соответствующее условие верно, и  $dest8 = 0$  иначе.

Приёмник *dest8* может быть как регистром, так и переменной в памяти, но только однобайтовыми.

## 5.3. Команды FPU

Напишу в чернилах: верь!  
Близок день, что всех возвысил!  
И грядёт бесшумно зверь  
С парой белых нежных чисел!  
*В. Хлебников. Зверь+число*

Команды расширения FPU, или математического сопроцессора, предназначены для обработки числовых данных в формате с плавающей точкой.

FPU выполняет все вычисления в 80-битном расширенном формате. Для обмена данными с памятью используются также вещественные числа одинарной (32 бита) и двойной (64 бита) точности, соответствующие стандарту IEEE 754-2008.

Регистры **данных** FPU образуют стек с плавающей вершиной. Соответственно, система команд FPU идеологически отличается от основной системы команд. **Один из операндов вычислений — всегда вершина стека FPU.**

Мнемоническое обозначение команд сопроцессора характеризует особенности их работы. Все мнемонические обозначения начинаются с символа *f* (FPU). Вторая буква мнемонического обозначения (если она не является частью имени действия, как в *finit*) определяет тип операнда в памяти, с которым работает команда:

- *i* — целое двоичное число со знаком;
- *b* — целое двоично-десятичное (BCD) число;
- отсутствие буквы для арифметических команд обозначает вещественное число.

Последняя буква *r* в мнемоническом обозначении команды означает, что последним действием команды обязательно является извлечение операнда из стека (удвоенная *rr* — из стека извлекаются оба операнда). Размер операнда в памяти, если он используется, задаётся суффиксом команды в соответствии с правилами синтаксиса AT&T.

Команды FPU не могут иметь непосредственных операндов и, за исключением команды выгрузки слова состояния, не могут работать с регистрами основного процессора. Если не указано иное, используются следующие обозначения. Операнд-приёмник может быть обозначен либо как *dest*, если он может быть регистром сопроцессора или переменной в памяти либо как *dmem*, если он может быть только в памяти. Операнд-источник может быть обозначен как *src* (регистр сопроцессора или переменная в памяти) или *smem* (переменная в памяти).

### 5.3.1. Внутреннее представление чисел

Значения в сопроцессоре представлены в нестандартном 80-битном формате с плавающей запятой, называемом форматом с двойной расширенной (или просто расширенной) точностью, описанном в разделе 2.8.2.

Нормализованное двоичное представление вещественного числа имеет вид [51, 73]:

$$(-1)^s \cdot 2^p \cdot \mu, \quad 0,1_2 \leq \mu < 1 \quad (5.2)$$

где  $p$  — порядок числа,  $\mu$  — мантисса,  $s$  определяет знак. Таким образом, все значащие разряды мантиссы находятся в дробной части. Старший из них (следующий сразу после запятой) для нормализованного числа всегда равен единице.

Старший бит 80-битного формата — знак  $s$ , порядок занимает следующие 15 бит и представляется кодом с избытком  $2^{14} - 2$  (так называемый смещённый порядок). В оставшиеся 64 бита записывается дробная часть мантиссы, включая ведущую единицу.

В частности, единица в нормализованном представлении имеет вид  $(-1)^0 \cdot 2^1 \cdot 0,1_2$ . Тогда значение смещённого порядка (после добавления избытка) будет равно  $2^{14} - 1$ :

$$1 = (-1)^0 \cdot 2^1 \cdot 0,1_2 \rightarrow \begin{array}{|c|c|c|} \hline \text{Знак} & \text{Порядок} & \text{Мантисса} \\ \hline 0 & 2^{14} - 1 & 100... \\ \hline \end{array} =$$

$$= \begin{array}{|c|c|c|} \hline \text{Знак} & \text{Порядок} & \text{Мантисса} \\ \hline 0 & 011...11 & 100... \\ \hline \end{array}$$

Таким образом, вещественное число расширенной точности, равное единице, имеет вид 3FFF 8000 0000 0000 0000 0000 0000 0000. Это подтверждает исследование с помощью отладчика.

Значение порядка, состоящее из пятнадцати нулей, зарезервировано под специальные значения, таким образом, минимально возможное значение порядка корректного вещественного числа имеет вид 00...001 и равно  $p_{min} = 1 + (-2^{14} + 2) = -2^{14} + 3$ . Соответственно, минимальное положительное число,

представимое в нормализованном виде в формате расширенной точности, равно  $X_{min} = 2^{p_{min}} \cdot 0,1_2 = 2^{-2^{14}+2} = 0001\ 8000 \dots 00$ . Числа в диапазоне  $(0, X_{min})$  представляются в виде  $2^{p_{min}} \cdot \mu$ ,  $0 < \mu < 0,1_2$  и называются **денормализованными**. В поле смещённого порядка таких чисел при этом записываются нули. В частности,  $\frac{X_{min}}{2} = 2^{p_{min}} \cdot 0,01_2 = 2^{p_{min}-1} \cdot 0,1_2$ , но представляется это число как  $0000\ 4000 \dots 00$ , в чём можно убедиться при помощи отладчика. Если попытаться прочесть такую запись как корректное число, то получим нулевой знаковый бит, нулевой смещённый порядок, что соответствует порядку  $p_{min} - 1$ , и мантиссу  $0,0100\dots_2$ , то есть  $(-1)^0 \cdot 2^{p_{min}-1} \cdot 0,01_2 = \frac{X_{min}}{4}$ , что неверно. Денормализованные числа — один из видов **специальных значений**, которые нельзя раскодировать по общему правилу.

Представление отрицательных вещественных чисел, в том числе из диапазона  $(-X_{min}, 0)$ , отличается от представления их модулей только знаковым битом.

Как было сказано в разделе 2.8.2, в некоторых источниках нормализованной формой мантиссы считается число, включающее один разряд целой части и 63 бита дробной [16] или целое беззнаковое 64-битное число с единицей в старшем разряде [80]. Обе этих трактовки приводят к тому же самому двоичному представлению, что и описанная выше.

## Виды значений

Регистры сопроцессора могут содержать следующие значения:

- вещественные числа — порядок не равен 0 и не состоит из всех единиц, старший бит мантиссы равен 1;
- денормализованные вещественные числа — порядок и старший бит мантиссы равны 0, но мантисса не равна нулю;
- нули ( $+0,0$  и  $-0,0$ , в соответствии со знаковым битом) — порядок и мантисса равны нулю;

- бесконечности ( $+\infty$  и  $-\infty$ , в соответствии со знаковым битом, обозначаются как `+inf` и `-inf`) — порядок состоит из всех единиц, старший бит мантиссы — единица, остальные равны нулю;
- нечисла двух типов:
  - сигнальные нечисла (при появлении такого значения в стеке генерируется исключение недействительной операции);
  - тихие нечисла (не генерируют исключения, но результат вычислений с операндом-нечислом — тоже нечисло):
    - вещественная неопределённость *nan* (знаковый бит не имеет значения) — порядок состоит из всех единиц, два старших бита мантиссы — единицы, остальные нули;
    - другие тихие нечисла — порядок состоит из всех единиц, два старших бита мантиссы — единицы, остальные — не все нули;
- недопустимые значения.

Начиная с 80387 некоторые ранее недопустимые значения стали нечислами различного типа, и наоборот — многие недопустимые для современных сопроцессоров значения были корректными нечислами в ранних дискретных моделях.

Если один из операндов равен произвольному тихому нечислу, он интерпретируется как вещественная неопределённость. Если вещественная неопределённость является результатом операции, она может быть равной только описанному значению *nan*.

### 5.3.2. Возможные форматы экспорта-импорта

Регистры сопроцессора могут содержать только вещественные числа расширенной точности или специальные значения формата расширенной точности. Тем не менее, при выгрузке значений из стека возможно преобразовать их в различные форматы трёх основных видов — с плавающей запятой, целые двоичные и целые двоично-десятичные.

Соответственно, при явной загрузке значений из памяти в стек FPU или выполнении вычислений с операндом в памяти возможен экспорт значений из этих форматов.

### Форматы с плавающей запятой

FPU поддерживает импорт и экспорт в стандартные форматы с плавающей запятой одинарной и двойной точности, соответствующие стандарту IEEE 754-2008. Также возможен импорт-экспорт в нестандартный 80-битный формат двойной расширенной точности, совпадающий с внутренним представлением чисел FPU.

Конкретный выбор формата определяется суффиксом команды (раздел 5.1.5). Суффикс *s* соответствует одинарной точности (32 бита, *float*), *l* — двойной (64 бита, *double*), *t* — нестандартному формату расширенной точности (80 бит, для GCC этот формат соответствует типу *long double*).

Если суффикс **размера** не указан, используется одинарная точность (*float*).

### Целые форматы

Поддерживается импорт и экспорт в двоичные знаковые целые форматы от двух до восьми байт. **При экспорте значение округляется в соответствии с текущими настройками FPU.** Отрицательные числа представлены в дополнительном коде.

Выбор формата определяется суффиксом команды (раздел 5.1.5). Суффикс *s* соответствует короткому целому (16 бит, *short*), *l* — длинному (32 бита, *long* и чаще всего *int*), *q* — четверному (64 бита, *long long*).

Если суффикс **размера** не указан, **число импортируется или экспортируется как короткое (16-битное) знаковое целое, в большинстве реализаций языка C++ соответствующее типу *short*.**

## Двоично-десятичный формат

FPU поддерживает экспорт и импорт только в один вид двоично-десятичных чисел — это 80-битный упакованный целый BCD-формат в виде значения со знаком.

Всего такое число занимает десять байт. Старший из них — знаковый. Его старший бит хранит знак числа — ноль соответствует положительному числу, единица — отрицательному. Младшие семь бит знакового байта не имеют значения. Остальные девять байтов содержат модуль числа в виде **восемнадцати** упакованных десятичных цифр.

Таким образом, BCD-формат FPU, как и форматы с плавающей запятой, включает два нуля:  $+0$  и  $-0$ .

### 5.3.3. Общие команды

Исторически набор команд FPU включает команды для начальной настройки сопроцессора, а также для синхронизации с центральным процессором. В настоящее время синхронизация не требуется.

#### Сброс FPU

Так как ранние модели сопроцессора **FPU** были отдельными устройствами, перед началом работы было необходимо определить, есть ли сопроцессор в системе, и, в случае его наличия, инициализировать сопроцессор. Для инициализации предназначена команда `finit` — сброс сопроцессора.

Команда `finit` восстанавливает значения по умолчанию в регистрах *sw*, *tw*, а начиная с 80387 — *fip* и *fdp*. Управляющее слово инициализируется значением `0x037F` (округление к ближайшему, 64-битная мантисса, все исключения замаскированы — то есть можно спокойно делить на 0, брать корень из отрицательных чисел и т. п., но результат будет не числом). Слово состояния обнуляется (*top* = 0, никакие флаги исключений не установлены). Регистры данных никак не изменяются, но все они помечаются пустыми в слове тегов *tw*. Регистры *fip* и *fdp* обнуляются.

Современные операционные системы сбрасывают и настраивают сопроцессор во время загрузки. Выполнять сброс вручную не стоит, так как это может повлиять на выполнение дальнейших расчётов на ЯВУ.

### Ожидание синхронизации

Оригинальный арифметический сопроцессор, выполненный в виде отдельной микросхемы, мог работать параллельно с центральным процессором. Для их синхронизации использовалась команда `wait/fwait`. Этим мнемоникам соответствует один и тот же машинный код. Эта команда приостанавливает работу либо FPU, либо центрального процессора — в зависимости от того, какой из них «вырвался вперёд» — и ждёт отстающего. Кроме того, многие команды управления сопроцессором реализованы в двух вариантах — с ожиданием и без. Мнемоника команды без ожидания отличается префиксом `n` после префикса FPU `f`, например, `fnstsw` и `fstsw`. При этом, согласно документации Intel, машинный код команды без префикса `n` состоит из кода команды `wait/fwait` и кода команды с префиксом `n`. В частности, команда `fstsw` полностью эквивалентна последовательности `fwait + fnstsw`.

В современных процессорах параллельная работа команд FPU и основного набора невозможна, так что команда `wait/fwait` эквивалентна `nop`. Соответственно, из двух команд — с префиксом `n` и без — в настоящее время необходимо выбирать вариант с префиксом.

#### 5.3.4. Загрузка, выгрузка и пересылка данных

Невозможно напрямую загрузить в стек сопроцессора значение регистра основного процессора или, наоборот, выгрузить значение из стека FPU в регистр CPU. Также невозможно загрузить в стек произвольную константу.

Это необходимо учитывать при написании ассемблерных вставок — все входные и выходные параметры, использующиеся как аргументы инструкций сопроцессора, должны располагаться в памяти. В частности, листинг 5.10 показывает вычисление значения выражения  $x + a$  и запись результата в  $y$ . Параметры



(как входные  $[X]$  и  $[A]$ , так и выходной  $[Y]$ ) расположены в памяти. К константе  $a$  (значению входного параметра  $[A]$ ) применён модификатор `volatile`, чтобы компилятор не оптимизировал её и разместил в памяти, как и необходимо.

**Листинг 5.10.** Вычисление  $y = x + a$  вставкой в код C++

```

1 const volatile double a = 12;
2 double x = 1, y;
3 asm(
4     "fldl_%[X]\n" // st(0) = %[X]
5     "faddl_%[A]\n" // st(0) = %[X] + %[A]
6     "fstpl_%[Y]\n" // %[Y] = %[X] + %[A], стек пуст
7     : [Y] "=m"(y)
8     : [X] "m"(x), [A] "m"(a)
9     : "cc"
10 );

```

В списке перезаписываемых регистров GCC не позволяет описывать элементы стека сопроцессора. Это не приводит к ошибкам, так как временные переменные не помещаются в стек сопроцессора.

Если вычисления должны быть не вставкой в код C++, а частью программы на ассемблере, все числа, включая константы (кроме, может быть, тех, для загрузки которых есть специальные команды), необходимо разместить в памяти.

**Листинг 5.11.** Вычисление  $y = x + a$

```

1 .data
2 a: .double 12
3 x: .double 1
4 y: .double
5 .text
6 fldl x
7 faddl a
8 fstpl y

```

При выходе из вставки или функции стек сопроцессора должен быть таким же, как на входе — обычно пустым, если только через него не возвращается значение (тогда в стеке не должно быть ничего, кроме возвращаемого значения).

## Загрузка данных в стек FPU

Для загрузки данных в стек сопроцессора предназначен набор инструкций `f*ld` (таблица 5.14). После загрузки значение преобразуется в число с двойной расширенной точностью (80 бит). Ячейка, куда было помещено значение, получает обозначение  $st(0)$ .

### Команды загрузки данных в стек FPU

Таблица 5.14

Команда	Действие
<code>fld src</code>	Помещает вещественное число <i>src</i> на вершину стека сопроцессора Если источник <i>src</i> в памяти, его размер определяется суффиксом команды (по умолчанию — 32 бита, <i>float</i> )
<b>Загрузка целых чисел</b>	
<code>fild smem</code>	Помещает целое знаковое число <i>smem</i> на вершину стека сопроцессора Размер источника определяется суффиксом команды (по умолчанию — 16 бит, <i>short</i> )
<code>fbld smem</code>	Помещает целое двоично-десятичное число <i>smem</i> на вершину стека сопроцессора Размер источника — 80 бит
<b>Загрузка констант</b>	
<code>fldz</code>	Загрузка 0
<code>fld1</code>	Загрузка 1
<code>fldpi</code>	Загрузка $\pi$
<code>fldl2t</code>	Загрузка $\log_2 10$
<code>fldl2e</code>	Загрузка $\log_2 e$
<code>fldlg2</code>	Загрузка $\log_{10} 2$
<code>fldln2</code>	Загрузка $\ln 2$

В стек можно поместить значение одного из элементов стека сопроцессора, значение из памяти или одну из predetermined набора констант. Невоз-

можно напрямую загрузить в стек сопроцессора значение регистра основного процессора.

### Выгрузка данных из стека FPU

Для выгрузки данных из стека сопроцессора предназначен набор инструкций `f*st [p]` (таблица 5.15).

#### Команды выгрузки данных из стека FPU

Таблица 5.15

Команда	Действие
<code>fst dest</code>	Копирует $st(0)$ в $dest$
<code>fstp dest</code>	Выталкивает $st(0)$ в $dest$ Приёмник $dest$ может быть переменной в памяти или пустым регистром сопроцессора Если $dest$ в памяти, его размер определяется суффиксом команды (по умолчанию — 32 бита, <i>float</i> )
<b>Выгрузка с округлением</b>	
<code>fist dmem</code>	Копирует $st(0)$ в $dmem$ как целое
<code>fistp dmem</code>	Выталкивает $st(0)$ в $dmem$ как целое Размер приёмника определяется суффиксом команды (по умолчанию — 16 бит, <i>short</i> )
<code>fbst dmem</code>	Копирует $st(0)$ в $dmem$ как двоично-десятичное целое
<code>fbstp dmem</code>	Выталкивает $st(0)$ в $dmem$ как двоично-десятичное целое Размер приёмника — 80 бит

Ниже показан пример использования команд загрузки и выгрузки (листинг 5.12).

#### Листинг 5.12. Последовательная загрузка и выгрузка данных

```

1 double x = 5.7, y;
2 float f;
3 long double l;
4 int i = 10;
5
6 asm(                                     //       $st(0), st(1), st(2), st(3)$ 
```

```

7      "fldl_ %[x]\n"    // в стеке: [x]
8      "fldz\n"          // в стеке: 0,      [x]
9      "fld1\n"          // в стеке: 1,      0,      [x]
10     "fildl_ %[i]\n"    // в стеке: [i],    1,      0,      [x]
11     "fstps_ %[f]\n"    // в стеке: 1,      0,      [x]      [f] = [i]
12     "fstpt_ %[l]\n"    // в стеке: 0,      [x]      [l] = 1
13     "fstpl_ %[y]\n"    // в стеке: [x]      [y] = 0
14     "fistpl_ %[i]\n"   // стек пуст      [i] = [x]
15
16     : [y] "=m"(y), [i] "+m"(i), [f] "=m"(f), [l] "=m"(l)
17     : [x] "m"(x)
18     : "cc"
19 )
20
21 #define PRINT(val) cout << #val << "_=" << val << "_";
22
23 PRINT(x)
24 PRINT(y)
25 PRINT(i)
26 PRINT(f)
27 PRINT(l)

```

Вначале в стек сопроцессора последовательно загружаются четыре значения: вещественная переменная двойной точности  $x = 5,7$ , константы — ноль и единица, а также целое 32-битное число  $i = 10$ .

После загрузки всех четырёх значений в стеке сопроцессора находятся следующие значения:

$$st(0) = 10 = i$$

$$st(1) = 1$$

$$st(2) = 0$$

$$st(3) = 5,7 = x$$

Все они внутри стека хранятся в 80-битном вещественном формате.

Затем верхнее значение  $st(0)$ , равное последнему загруженному значению  $i = 10$ , выталкивается из стека и записывается по адресу параметра  $[f]$ , преобразованное в вещественное число одинарной точности. Новое значение вершины стека  $st(0)$  после выталкивания — единица. Соответственно, изменятся и обозначения более глубоких элементов стека:  $st(1) = 0$  и  $st(2) = x$ .

Затем новое значение вершины стека, равное единице, выталкивается в параметр  $[l]$  как 80-битное число (суффикс  $t$  — *ten bytes*, что для компиляторов GCC соответствует типу *long double*). Ноль выталкивается из стека и записывается в  $[y]$  как число двойной точности. Последний оператор выталкивает значение  $x$  в параметр  $[i]$  как длинное целое (*int*), после чего стек остаётся пустым. Значение 5,7 округляется в соответствии с текущими настройками округления, в данном случае — к ближайшему, то есть получим  $[i] = 6$ .

Соответственно, результат отладочной печати в конце листинга выглядит следующим образом:

```
x = 5.7  y = 0  i = 6  f = 10  l = 1
```

Используя команды выгрузки или загрузки из памяти, необходимо внимательно следить за суффиксами команд. В вышеописанном примере команда `fstp %[y]` не вызвала бы ошибки ни во время компиляции, ни во время выполнения, но переменная  $y$  получила бы весьма странное значение. По умолчанию (без указания суффикса) `fstp` записывает снятое с вершины стека значение как вещественное число одинарной точности, то есть из 64 бит переменной  $y$  будут перезаписаны только первые 32, причём в формате, не соответствующем типу *double*.

### Пересылка данных внутри стека FPU

Для пересылки данных внутри стека сопроцессора можно использовать команды `fld st(i)` для загрузки копии значения  $st(i)$  в вершину стека и `fst st(i)` для помещения значения  $st(0)$  в ранее пустую ячейку  $st(i)$ . Кроме того, существует две специализированные команды (таблица 5.16).

Команда условного копирования `fscovCC` использует как условие флаги регистра *flags*, а не регистра состояния *sw*. При этом для неё доступны не все условия *CC*, перечисленные в таблице 5.11, и даже не все синонимы доступных условий. Используемые условия перечислены в таблице 5.17.

## Команды пересылки данных FPU

Таблица 5.16

Команда	Действие
<code>fcmovCC %st(i), %st(0)</code>	Присваивание $st(0) = st(i)$ , если верно $CC$ (е, ne, b/nae, be/na, ae/nb, a/nbe, u и nu)
<code>fxch</code>	Меняет местами $st(0)$ и $st(1)$
<code>fxch %st(i)</code>	Меняет местами $st(0)$ и $st(i)$

Условия `fcmovCC` и их связь с флагами состояния *flags*

Таблица 5.17

CC	Условие (флаги)	Условие ( $f*cmp\ src$ )
e	$ZF = 1$	$st(0) = src$ <i>if equal</i>
ne	$ZF = 0$	$st(0) \neq src$ <i>if not equal</i>
b/nae	$CF = 1$	$st(0) < src$ <i>if below / if not above or equal</i>
be/na	$\begin{cases} CF = 1 \\ ZF = 1 \end{cases}$	$st(0) \leq src$ <i>if below or equal / if not above</i>
nb/ae	$CF = 0$	$st(0) \geq src$ <i>if not below / if above or equal</i>
nbe/a	$\begin{cases} CF = 0 \\ ZF = 0 \end{cases}$	$st(0) > src$ <i>if not below or equal / if above</i>
u	$PF = 1$	$src$ и $st(0)$ несравнимы <i>if unordered</i>
nu	$PF = 0$	$src$ и $st(0)$ сравнимы <i>if not unordered</i>

Условия `fcmovCC` включают те биты регистра флагов *flags*, которые могут быть прямо или косвенно (путём сохранения слова состояния *sw* и загрузки его части в *flags*) установлены командами сравнения FPU, то есть флаги  $ZF, CF, PF$ .

Команда обмена регистров `fxch` на самом деле не копирует данные, а переименовывает регистры, так что её выполнение практически не занимает времени.

Так как большинство команд работает с вершиной стека, переименование регистров с помощью `fxch` часто бывает удобным.

### Загрузка и выгрузка управляющих регистров

Содержимое управляющих регистров также может быть сохранено в памяти (таблица 5.18).

#### Команды загрузки и выгрузки управляющих регистров FPU

Таблица 5.18

Команда	Действие
<code>fnstcw dmem16</code>	Выгрузка управляющего слова <i>cw</i> в память
<code>fldcw smem16</code>	Загрузка управляющего слова <i>cw</i> из памяти
<code>fnstsw dest16</code>	Выгрузка (разрушающая) слова состояния <i>sw</i> в память или регистр <i>ax</i>

Управляющее слово можно как выгрузить в память по заданному адресу, так и загрузить из неё.

Слово состояния сопроцессора можно только сохранить в память, а также в регистр *ax* (и только в этот регистр).

После выгрузки слова состояния *sw* командой `fnstsw` теряется значение специальных флагов *C0–C3*.

### 5.3.5. Основные арифметические команды

Основные арифметические команды сопроцессора выполняют базовые бинарные арифметические операции — сложение, вычитание, умножение и деление. Хотя бы один операнд должен быть в вершине стека сопроцессора  $st(0)$ . Результат помещается на место одного из операндов (приёмника) в стек сопроцессора, заменяя старое значение. Приёмник должен быть в стеке сопроцессора, но не обязательно на его вершине.

Каждая из основных арифметических команд может быть записана в нескольких формах. Они различаются положением источника и приёмника, также неко-

торые из форм после вычисления результата выталкивают источник из стека сопроцессора, что обозначается суффиксом *p*.

### Обозначения основных арифметических команд

Четырём арифметическим операциям в FPU соответствует шесть различных операций. При этом каждой из симметричных относительно перестановки операндов арифметических операций — сложению и умножению — соответствует по одной операции FPU: сложение *fadd* и умножение *fmul*. Несимметричным операциям — вычитанию и делению — соответствует по две операции FPU, отличающиеся порядком операндов. Это соответственно прямое вычитание *fsub* и обратное вычитание *fsubr*, а также прямое деление *fdiv* и обратное деление *fdivr* (таблица 5.19).

### Основные арифметические операции FPU

Таблица 5.19

Команда	Действие Intel	Действие GAS*
<i>fadd</i> (сложение)	$dest = dest + src = st(0) + \xi$	
<i>fsub</i> (вычитание)	$dest = dest - src$	$dest = st(0) - \xi$
<i>fsubr</i> (обратное вычитание)	$dest = src - dest$	$dest = \xi - st(0)$
<i>fmul</i> (умножение)	$dest = dest \cdot src = st(0) \cdot \xi$	
<i>fdiv</i> (деление)	$dest = dest / src$	$dest = st(0) / \xi$
<i>fdivr</i> (обратное деление)	$dest = src / dest$	$dest = \xi / st(0)$

\*  $\xi$  — операнд, не лежащий на вершине стека.

Может быть как источником, так и приёмником, в зависимости от используемой формы.

Все формы основных арифметических команд используют два явно или неявно заданных операнда. Один из них всегда в вершине стека  $st(0)$ , другой (обозначим его  $\xi$ ) может быть в памяти или в регистре  $st(i)$ . Кроме того, один из этих операндов является приёмником  $dest$ , второй — источником  $src$ .



**Внимание!**

Ассемблер Unix исторически использовал для основных арифметических команд FPU те же мнемонические обозначения, что и предложенные Intel, но другую семантику операндов.

Таким образом, в GAS поведение мнемоник несимметричных операций (`fsub/fsubr` и `fdiv/fdivr`) качественно иное, чем описанное в документации Intel и учебниках, описывающих синтаксис Intel.

Согласно документации Intel (и в ассемблерах с синтаксисом Intel) прямое вычитание `fsub` вычисляет  $dest - src$ , а обратное `fsubr` —  $src - dest$ , то есть результаты команд `fsub %st(0), %st(i)` и `fsub %st(i), %st(0)` не только записываются в различные регистры, но и отличаются знаком.

В GAS, в соответствии с традиционным поведением Unix-ассемблеров, `fsub` вычисляет  $st(0) - \xi$  даже в том случае, если приёмником является  $\xi$ . В частности, команды `fsub %st(0), %st(i)` и `fsub %st(i), %st(0)` вычисляют одно и то же значение, но помещают его в разные регистры. Обратное вычитание `fsubr` вычисляет  $\xi - st(0)$ .

Таким образом, команде `fsub %st(0), %st(i)` соответствует опкод, который, согласно документации Intel, должен соответствовать команде `fsubr` [59]. Анализ сгенерированного компилятором из коллекции GCC кода это подтверждает. Аналогично ведут себя `fdiv/fdivr`. Такое поведение в случае сочетания синтаксиса AT&T и платформы x86 в некоторых источниках описывается как баг GCC [54], но из соображений совместимости с имеющимся кодом меняться не будет.

Поведение Intel и GAS совпадает в тех случаях, когда приёмником является  $st(0)$ , в том числе в ситуациях, когда источник находится в памяти. Также поведение Intel и GAS полностью совпадает для симметричных операций — сложения и умножения.

## Формы основных арифметических команд

Согласно документации Intel, сопроцессор использует шесть форм [48] основных арифметических команд (таблица 5.20). Строка XXX соответствует выполняемой операции (add, sub, subr, mul, div, divr).

### Шесть форм основных арифметических команд FPU

Таблица 5.20

Команда	Действие
fXXXp	Источник — $st(0)$ , приёмник — $st(1)$ , после выполнения источник $st(0)$ извлекается из стека (то есть результат получает обозначение $st(0)$ ) Данная форма эквивалентна fXXXp %st(0), %st(1) (ассемблируется в тот же опкод)
fXXX smem	Источник — память (вещественное число), приёмник — $st(0)$ , указатель стека не изменяется Размер источника определяется суффиксом команды, может быть одинарной или двойной точности, но не 80-битным (по умолчанию — 32 бита, <i>float</i> )
fiXXX smem	Источник — память (целое знаковое число), приёмник — $st(0)$ , указатель стека не изменяется Размер источника определяется суффиксом команды, может быть 16- или 32-, но не 64-битным (по умолчанию — 16 бит, <i>short</i> )
fXXX %st(i), %st(0)	Источник — $st(i)$ , приёмник — $st(0)$ , указатель стека не изменяется
fXXX %st(0), %st(i)	Источник — $st(0)$ , приёмник — $st(i)$ , указатель стека не изменяется
fXXXp %st(0), %st(i)	Источник — $st(0)$ , приёмник — $st(i)$ , после выполнения источник $st(0)$ извлекается из стека (то есть результат получает обозначение $st(i - 1)$ )

Если посмотреть на опкоды этих форм [17], видно, что опкод формы без параметров `fXXXp` полностью совпадает с опкодом формы с двумя параметрами `fXXXp %st(0), %st(1)`. Напротив, формы `f[i]add smem` для различной разрядности источника *smem* имеют по два различных опкода.

В частности, для операции сложения `fadd` возможны следующие формы:

- а) `faddp` выполняет сложение  $st(1) = st(0) + st(1)$  и выталкивание  $st(0)$  из стека, так что после этой операции результат оказывается в  $st(0)$  (эквивалент `faddp %st(0), %st(1)`);
- б) `fadd smem` —  $st(0) = st(0) + \text{вещественное } smem$ ;
- в) `fiadd smem` —  $st(0) = st(0) + \text{целое } smem$ ;
- г) `fadd %st(i), %st(0)` —  $st(0) = st(0) + st(i)$ ;
- д) `fadd %st(0), %st(i)` —  $st(i) = st(0) + st(i)$ ;
- е) `faddp %st(0), %st(i)` —  $st(i) = st(0) + st(i)$  и выталкивание  $st(0)$  из стека, так что после этой операции результат оказывается в  $st(i - 1)$ .

Большинство ассемблеров, в частности, GAS, поддерживает и некоторые дополнительные формы основных арифметических команд, оба операнда которых находятся в стеке. В частности, для формы без операндов `fXXXp` практически во всех ассемблерах принят синоним `fXXX`. Но, так как мнемоника без суффикса `p` не отражает выполняемое выталкивание  $st(0)$  из стека, её использование не рекомендуется.

Кроме того, по аналогии с `fXXX smem` поддерживается форма `fXXX %st(i)` с приёмником в  $st(0)$ , а также `fXXXp %st(i)` с приёмником изначально в  $st(i)$ , а после выталкивания  $st(0)$  — в  $st(i - 1)$ .

В GAS, кроме всего прочего, доступна «нелегальная» форма записи `fXXXp %st(i), %st(0)`, например, `fsubp %st(i), %st(0)`. Такая запись вызывает при компиляции предупреждение, но не ошибку (хотя по сути является ошибочной, источник  $st(i)$  невозможно вытолкнуть из стека) и преобразуется в `fsubp %st(0), %st(i)`.

Подобные формы лучше не использовать из-за неочевидности расположения операндов. При этом неуказание части операндов в программе не даёт преимущества в исполняемом файле, так как любая форма из перечисленных дополнительных форм арифметических команд будет ассемблироваться в тот же опкод, что и форма с двумя явно указанными операндами.

### Практическое использование основных арифметических команд

Ниже показано использование основных арифметических команд для расчёта значения выражения  $x + \frac{1}{i} + a \cdot \pi$ . Так как используется GAS, команда `fdivr` без операндов рассчитывает  $st(1)/st(0)$ , после чего источник  $st(0)$  выталкивается из стека FPU.

В листинге 5.13 приведена функция `double foo(double x, int i, double a)`, соответствующая соглашению о вызове `cdecl` (см. раздел 6.2.3).

#### Листинг 5.13. Расчёт $y = x + \frac{1}{i} + a \cdot \pi$ как функция

```

1 .globl foo
2 foo:                //          st(0),    st(1),    st(2),    st(3)
3   fldl 4(%esp)       // в стеке: x
4   fldpi              // в стеке: pi,      x
5   fldl              // в стеке: 1,       pi,      x
6   fildl 12(%esp)     // в стеке: i,      1,       pi,      x
7   fdivr              // в стеке: 1/i,    pi,      x
8   fldl 16(%esp)     // в стеке: a,      1/i,    pi,      x
9   fmulp %st(0), %st(2) // в стеке: 1/i,    pi*a,   x
10  faddp              // в стеке: 1/i+pi*a, x
11  faddp              // в стеке: 1/i+pi*a+x (результат)

```

Операнды, в соответствии с тридцатидвухбитным соглашением `cdecl`, передаются в стеке в памяти (первый параметр  $x$  находится по адресу  $sp + 4$  и занимает восемь байт, так что следующий параметр,  $i$ , располагается по адресу  $\&x + 8 = sp + 4 + 8 = sp + 12$  и  $a$  — по адресу  $sp + 16$ ), поэтому их можно загрузить командами `f*ld` в стек регистров FPU. Возвращаемое вещественное значение, согласно тому же соглашению, передаётся в  $st(0)$ , поэтому оно не выталкивается после вычислений.

Далее показан тот же код расчёта значения выражения  $x + \frac{1}{i} + a \cdot \pi$ , оформленный как вставка в программу на C++, где  $x$ ,  $i$  и  $a$  — значения переменных (листинг 5.14).

**Листинг 5.14.** Расчёт  $y = x + \frac{1}{i} + a \cdot \pi$  как вставка

```

1 const volatile double a = 0.01;
2 double x = 5, y;
3 int i = 10;
4 asm(                                     //          st(0),    st(1), st(2), st(3)
5     "fldl_%[x]\n"                       // в стеке: x
6     "fldpi\n"                           // в стеке: pi,      x
7     "fldl\n"                             // в стеке: 1,      pi,      x
8     "fildl_%[i]\n"                      // в стеке: i,      1,      pi,      x
9     "fdivr\n"                           // в стеке: 1/i,    pi,      x
10    "fldl_%[A]\n"                       // в стеке: A,      1/i,    pi,      x
11    "fmulp_%st(0),_%st(2)\n"             // в стеке: 1/i,    pi*A,   x
12    "faddp\n"                           // в стеке: 1/i+pi*A, x
13    "faddp\n"                           // в стеке: 1/i+pi*A+x
14    "fstpl_%[y]\n"                      // y = 1/i + pi*A + x, стек пуст
15
16    : [y] "=m" (y)
17    : [x] "m" (x), [A] "m" (a), [i] "m" (i)
18    : "cc"
19 ); // y = x + 1/i + a*pi

```

Приведённый код — не единственный способ расчёта значения указанного выражения. В зависимости от того, в каком порядке программист будет рассчитывать компоненты выражения, может различаться как порядок команд, так и сами команды (в частности, возможно использование прямого деления `fdiv`, а не обратного `fdivr`).

### 5.3.6. Дополнительные арифметические и трансцендентные команды

Дополнительные арифметические и трансцендентные команды [17, 33] работают с вершиной стека  $st(0)$  и, при необходимости, с  $st(1)$ . Для них не указывают явных операндов. Соответственно, каждая из команд этой группы имеет только одну форму.

Некоторые дополнительные арифметические и трансцендентные команды перечислены в таблице 5.21.

## Дополнительные арифметические и трансцендентные команды FPU

Таблица 5.21

Команда	Действие
fabs	$st(0) =  st(0) $
fsqrt	$st(0) = \sqrt{st(0)}$
fptan	$\begin{cases} st(0) = 1 \\ st(1) = \operatorname{tg}(st(0)) \end{cases} \quad \text{— частичный тангенс } st(0)$
fsincos	$\begin{cases} st(0) = \cos(st(0)) \\ st(1) = \sin(st(0)) \end{cases}$ <p>fsincos выполняется столько же времени, сколько fsin или fcos (и вдвое меньше отдельного расчёта синуса и косинуса)</p>
fsin	$st(0) = \sin(st(0))$
fcos	$st(0) = \cos(st(0))$
fpatan	$\left[ st(1) \rightarrow st(0) \right] = \operatorname{parctg} \left( \frac{st(1)}{st(0)} \right) \quad \text{— частичный арктангенс } \frac{st(1)}{st(0)}.$ <p>Результат в диапазоне <math>[-\pi, \pi]</math>.</p>
fscale	$st(0) = st(0) \cdot 2^{\lfloor st(1) \rfloor}$ , $st(1)$ остаётся в стеке
f2xmi	$st(0) = 2^{st(0)} - 1$ <p>Значение <math>st(0)</math> должно лежать в пределах от <math>-1</math> до <math>+1</math>, иначе результат не определён.</p>
fyl2x	$\left[ st(1) \rightarrow st(0) \right] = st(1) \cdot \log_2(st(0)), \quad st(0) \geq 0$ <p>Если регистр <math>st(0)</math> содержал ноль, результат (если <math>ZM = 1</math>) будет равен бесконечности со знаком, обратным <math>st(1)</math>.</p>
fyl2xp1	$\left[ st(1) \rightarrow st(0) \right] = st(1) \cdot \log_2(st(0) + 1),$ <p><math>st(0)</math> от <math>-(1 - \frac{\sqrt{2}}{2}) \approx -0,3</math> до <math>(1 + \frac{\sqrt{2}}{2}) \approx 1,7</math>, иначе результат не определён.</p> <p>Команда fyl2xp1 даёт большую точность для <math>st(0)</math>, близких к нулю, чем fyl2x для суммы того же <math>st(0)</math> и 1.</p>
fprem1	$\left[ st(1) \rightarrow st(0) \right] = st(0) \bmod st(1) \quad \text{— частичный остаток по IEEE-754}$ <p>Применяется, в частности, для уменьшения аргументов периодических функций: остаток от деления <math>3 \cdot \pi + 1,2</math> на <math>\pi</math> равен 1,2.</p>

Использование этих команд не перезаписывает значения, лежащие в стеке ниже неявных аргументов. Если у команды только один аргумент в  $st(0)$  и один результат, результат записывается в  $st(0)$  на место аргумента. Если у команды один аргумент в  $st(0)$  и два результата (`fptan`, `fsincos` и т. д.), то один из результатов помещается в  $st(0)$ , второй затем помещается в стек сверху (так что первый результат оказывается в  $st(1)$ , второй — в  $st(0)$ ).

В случае команд с двумя аргументами  $st(0)$ ,  $st(1)$  и одним результатом чаще всего результат помещается в  $st(1)$ , затем  $st(0)$  выталкивается из стека, так что после этой операции результат оказывается в  $st(0)$ . Таким образом, результат замещает собой аргументы (в таблице 5.21 такая ситуация соответствует обозначению  $[st(1) \rightarrow st(0)]$  для результата).

Иногда (в частности, `fscale`) команда с двумя аргументами в  $st(0)$  и  $st(1)$  записывает результат в  $st(0)$ , оставляя аргумент в  $st(1)$  в стеке.

Результат трансцендентных и тригонометрических команд (`fsin`, `fcos`, `fsincos`, `fptan`, `fpatan`, `f2xm1`, `fyl2x`, `fyl2xp1`) всегда помечается как неточный (исключительная ситуация `#P`).

Пример использования тригонометрических команд для расчёта значения выражения  $a \cdot \cos(x) + \sin(x)$  показан в листинге 5.15.

### Листинг 5.15. Расчёт $y = a \cdot \cos(x) + \sin(x)$

```

1 const volatile double a = 100;
2 double x = M_PI/6, y;
3 asm(
4     "fldl %[X]\n"    // st(0) = %[X]
5     "fsincos\n"      // st(0) = cos(%[X]),      st(1) = sin(%[X])
6     "fmull %[A]\n"   // st(0) = %[A]*cos(%[X]),  st(1) = sin(%[X])
7     "faddp\n"        // st(0) = %[A]*cos(%[X]) + sin(%[X])
8     "fstpl %[Y]\n"   // %[Y] = %[A]*cos(%[X]) + sin(%[X]), стек пуст
9
10    : [Y] "=m" (y)
11    : [X] "m" (x), [A] "m" (a)
12    : "cc"
13 );          // y = a*cos(x) + sin(x)

```

Для всех тригонометрических команд операнд считается заданным в радианах и не может быть больше  $2^{63}$  или меньше  $-2^{63}$ . Если операнд выходит за эти пределы, флаг *C2* устанавливается в единицу, значение  $st(0)$  и стек не изменяются.

Частичный арктангенс *fpatan* отличается от математического определения арктангенса тем, что принимает два аргумента, соответствующие координатам некоторой точки  $\begin{cases} st(0) = x \\ st(1) = y \end{cases}$  и возвращает результат в диапазоне  $[-\pi, \pi]$ , равный азимуту заданной точки  $(x, y)$ . Соответственно, знаки аргументов определяют квадрант результата.

Чтобы получить результат в диапазоне  $(-\frac{\pi}{2}, \frac{\pi}{2})$ , то есть в соответствии с математическим определением  $arctg(x)$ , необходимо задать точку в первом или четвёртом квадрантах, то есть с положительной абсциссой —  $\begin{cases} st(0) = 1 \\ st(1) = x \end{cases}$ .

Остальные обратные тригонометрические функции можно получить с помощью команды *fpatan* и арифметических команд, используя основное тригонометрическое тождество и задавая координаты соответствующих точек.

### 5.3.7. Сравнение вещественных чисел

FPU включает несколько семейств команд сравнения вещественных чисел. Все они сравнивают приёмник  $st(0)$  с некоторым источником *src*. По аналогии с командой целочисленного сравнения можно сказать, что анализируется знак разности  $st(0) - src$ . Так как приёмником является  $st(0)$ , поведение команд сравнения не различается для GAS и Intel.

Некоторые из них помещают результат в слове состояния *sw*, откуда его надо вручную копировать в регистр флагов *flags* (при этом осмысленное значение приобретают *ZF, CF, PF*). Они поддерживаются, но считаются устаревшими. Более современные команды сравнения помещают результат непосредственно во флагах *ZF, CF, PF* регистра *flags*.

Также система команд FPU включает *fхам*, которая определяет вид значения в  $st(0)$  в соответствии с разделом 5.3.1.



Знак нуля при сравнении не учитывается, то есть считается, что  $-0 = +0$ .

## Команды сравнения

Все команды сравнения вещественных чисел [71] сравнивают вершину стека — приёмник  $st(0)$  с другим операндом — источником  $src$  (таблица 5.22).

### Команды сравнения FPU

Таблица 5.22

Команда	Источник	Флаги	Особенности
$fcom[p[p]]$ $fcom[p] \ src$	$st(1)$ $src$	$C0, C3, C2$	
$fucom[p[p]]$ $fucom[p] \ src$	$st(1)$ $src$		Не генерируется исключения при сравнении с $nan$
$ficom[p] \ smem$	$smem$		$smem$ — целое число
$ftst$	0		
$fcomi[p] \ \%st(i), \%st(0)$	$st(i)$	$CF, ZF, PF$	
$fucomi[p] \ \%st(i), \%st(0)$	$st(i)$		Не генерируется исключения при сравнении с $nan$

По результатам сравнения (в соответствии со знаком разности  $st(0) - src$ ) устанавливается значение трёх флагов: отрицательности, нуля и несравнимости (таблица 5.23). Операнды считаются несравнимыми, если хотя бы один из них — тихое нечисло (любые тихие нечисла как операнды команд обрабатываются, как вещественная неопределённость  $nan$ , поэтому обычно говорят, что операнды несравнимы, если хотя бы один из них равен  $nan$ ).

Действие команд сравнения одинаково для синтаксиса AT&T и синтаксиса Intel. Мнемоника может включать суффикс  $p$ , в этом случае приёмник  $st(0)$  после сравнения выталкивается из стека. Если явный операнд не задан (то есть источником считается  $st(1)$ ), может также использоваться суффикс  $pp$  — в этом случае после сравнения из стека выталкиваются оба операнда,  $st(0)$  и  $st(1)$ .

## Значение флагов при сравнении

Таблица 5.23

Соотношения	Флаги		
	Отрицательности $C0/CF$	Нуля $C3/ZF$	Несравнимости $C2/PF$
$st(0) > src \quad st(0) - src > 0$	0	0	0
$st(0) = src \quad st(0) - src = 0$	0	1	0
$st(0) < src \quad st(0) - src < 0$	1	0	0
$st(0)$ и $src$ несравнимы	1	1	1

Если источник задан явно и находится в памяти, необходимо указывать также суффикс размера по тем же правилам, что и для арифметических команд.

Если хотя бы одно из сравниваемых значений — нечисло, для большей части команд сравнения (без префикса *u*) это недействительная арифметическая операция *#IA*. Если соответствующее исключение не замаскировано (раздел 3.4.2), работа программы прерывается, если замаскировано — устанавливается флаг несравнимости. Команды неупорядоченного сравнения, мнемоники которых включают префикс *u*, считают операцию сравнения с тихим нечислом, в частности, вещественной неопределённостью, действительной, и устанавливают в этом случае флаг несравнимости. Если хотя бы одно из сравниваемых значений — неподдерживаемое значение или сигнальное нечисло, операция сравнения недействительна (*#IA*) для всех команд.

По набору используемых флагов команды сравнения делятся на две группы — часть их выставляет биты слова состояния *sw* (*C3*, *C0* и *C2*), часть — биты регистра *flags* (*CF*, *ZF* и *PF*).

В слове состояния сопроцессора *sw* результат сохраняют команды сравнения оригинального FPU 8087 и 80387. В настоящее время такой способ также доступен в силу преемственности набора команд x86, но неоптимален. Начиная

с Pentium Pro, доступен более быстрый вариант. Современные процессоры включают команды сравнения с суффиксом *i* (*fcomi*, *fcomip*, *fucomi*, *fucomip*), которые напрямую устанавливают флаги *ZF*, *CF*, *PF* в *flags*. Неиспользуемые три флага состояния *flags* сбрасываются в 0; биты *C0*, *C2*, *C3* слова состояния сопроцессора не изменяются.

Если при сравнении целых чисел в регистре *flags* выставляется значение тех же флагов, которые выставляются по результатам арифметических действий, то в FPU флаги, выставляемые устаревшими командами сравнения в слове состояния *sw*, отличаются от тех, что устанавливаются, в частности, при вычитании.

### Анализ результатов сравнения

Условные команды, даже из набора FPU, не могут анализировать флаги слова состояния FPU *sw*. Соответственно, если используется одна из устаревших команд сравнения, сохраняющая результат в *sw*, после её выполнения необходимо вручную перенести его в регистр флагов *flags* основного процессора.

Это выполняется в два этапа:

- слово состояния *sw* выгружается в регистр *ax* командой *fnstsw*;
- старший байт *ax* загружается в младший байт регистра флагов *flags* командой *sahf*.

При загрузке старшего байта *sw* во *flags* флаг отрицательности *C0* помещается во флаг беззнакового переполнения *CF*, флаг нуля *C3* — в аналогичный ему по смыслу *ZF*, а флаг несравнимости *C2* — во флаг чётности *PF*. Другие флаги младшего байта *flags* получают фактически неопределённое значение. Таким образом, результат можно анализировать как результат сравнения беззнаковых целых чисел.

Результат современных команды сравнения, напрямую устанавливающих флаги *ZF*, *CF*, *PF* и обнуляющих остальные флаги состояния в регистре *flags*, можно анализировать как результат сравнения беззнаковых целых чисел без дополнительных действий.

## Определение вида значения

Кроме команд сравнения, анализирующих разность двух значений  $st(0) - src$  как число, набор команд FPU включает также команду `fxam`, которая анализирует тип содержимого вершины стека (нормальное число, ноль, бесконечность, денормализованное число и т. д.). Эта команда достаточно старая, поэтому записывает результат в слово состояния FPU.

Команда `fxam` выставляет в соответствии с значением  $st(0)$  все четыре специальных флага  $C0 - C3$  слова состояния  $sw$  (таблица 5.24).

### Значение флагов при определении вида $st(0)$

Таблица 5.24

Значение в $st(0)$	Флаги			
	$C0$	$C1$	$C2$	$C3$
Недопустимое значение	0	Знаковый бит $st(0)$	0	0
Ноль	0		0	1
Денормализованное число	0		1	1
Нормализованное число	0		1	0
Бесконечность	1		1	0
Неопределённость ( <i>nan</i> )	1		0	0
Пустой регистр	1		0	1

Флаг  $C1$  всегда устанавливается равным знаковому биту  $st(0)$ , даже если в нём находится значение, не имеющее знака (вещественная неопределённость или недопустимое значение). Необходимо отметить, что при загрузке флагов FPU в регистр *flags* бит  $C1$  попадает на зарезервированный разряд и, соответственно, игнорируется.

Команда `fxam` корректно работает даже при пустом стеке. Если регистр данных  $r_{top}$ , соответствующий вершине стека, помечен в регистре тегов  $tw$  как

пустой, флаги  $C0$ ,  $C3$ ,  $C2$  получают значения, указанные в последней строке таблицы 5.24, а  $C2$  — значение знакового бита  $r_{top}$ .

Если  $st(0)$  не пуст, флаги  $C0$ ,  $C3$ ,  $C2$  соответствуют виду содержащегося в нём значения (более подробно виды значений FPU описаны в разделе 5.3.1).

## Контрольные вопросы

1. Какой порядок операндов принят в синтаксисе AT&T?
2. Какие вы знаете команды передачи управления?
3. Какие вы знаете команды пересылки данных?
4. Какие команды используются для обнуления регистра?
5. Какие команды используются для выполнения арифметических операций над целыми числами?
6. Какие команды используются для выполнения арифметических операций над вещественными числами?
7. Какие команды используются для выполнения тригонометрических операций?
8. Какие команды используются для сравнения вещественных чисел?
9. Какие флаги регистра  $flags$  содержат результат сравнения вещественных чисел?

## Глава 6

# Программирование на языке Ассемблера

А потому, после того, как будешь свободен, будь деятелен.

*Коран. 94.7*

Некоторые приёмы программирования на ассемблере существенно отличаются от используемых в языках высокого уровня. При этом GNU Assembler является неотъемлемой частью процесса компиляции этих языков, в первую очередь — C/C++. Соответственно, структура программы на GNU Assembler аналогична C/C++, кроме того, полностью доступна стандартная библиотека `libc`. Данная глава показывает как возможную связь программ на C/C++ и ассемблере, так и различия в реализации алгоритмов.

Если не указано иное, примеры соответствуют тридцатидвухбитной платформе. Вызовы функций описываются в соответствии с тридцатидвухбитным соглашением `cdecl` и без учёта искажения имён (см. раздел 6.2).

### 6.1. Структура программы на ассемблере

Отсюда-то всё и начинается...

*В. Кандинский.*

*Пощёчина общественному вкусу. Видеть*

Программа обязательно должна включать точку входа — адрес, с которого начинается её выполнение. По умолчанию компоновщик GCC ищет точку входа по имени `_start`. Для программ на C/C++ по адресу `_start` находится стартовый код библиотеки `libc`, который, в частности, инициализирует все используемые библиотекой ресурсы и вызывает так называемую головную функцию `main()`.

После этого программа выполняется по определённому алгоритму (в частности, с помощью цикла обработки сообщений можно реализовать событийно-ориентированную модель) и при определённых обстоятельствах должна корректно завершать свою работу.

Минимальная программа запускается и немедленно завершает работу. Также в учебных целях часто описывается программа, выводящая на экран приветствие «Hello, world!».

### 6.1.1. Программирование с использованием `libc`

По умолчанию в GCC программа (как на языке C/C++, так и на языке ассемблера) собирается с поддержкой стандартной библиотеки `libc`. Соответственно, стартовой (главной) функцией программы является `int main(int argc, char * argv[])`.

Стартовая функция `main()` может находиться как в модуле на языке C/C++, так и в ассемблерном модуле.

#### Минимальная программа с использованием `libc`

Приведём код минимальной программы на ассемблере GAS с использованием стандартной библиотеки `libc` (листинг 6.1).

#### Листинг 6.1. Минимальная программа с использованием `libc`

```
1 .globl main // головная функция (libc)
2 main:
3     xor %eax, %eax // A ^= A, то есть A = 0
4     ret // return A
```

Команда `ret`, в отличие от оператора C++ `return`, не принимает возвращаемое значение как параметр. Целый результат в соответствии с соглашениями о вызовах (раздел 6.2) всегда подразумевается в регистре `A`. Для тридцатидвухбитного кода это `eax`, поэтому, чтобы вернуть код успешного завершения программы (0), необходимо обнулить регистр `eax`. В данном случае это делается при помощи

побитового исключающего «или». Эта команда компактнее явного копирования нуля в регистр и выполняется быстрее.

Параметры *argc* и *argv* располагаются в соответствии с используемым соглашением о вызове, то есть находятся в стеке для тридцатидвухбитных систем и в регистрах для шестидесятичетырехбитных. Имя функции *main()*, как и имена других функций *libc*, в GNU/Linux не искажается.

Эквивалент программы 6.1 на C++ выглядит следующим образом.

### Листинг 6.2. Минимальная программа с использованием *libc* (C++)

```
1 int main()
2 {
3     return 0;
4 }
```

Неиспользуемые параметры *main()* стандарт позволяет опустить.

### Приветствие миру

Более сложная программа, выводящая на экран строку "Hello, \\_world!\n" с помощью функции *printf()* библиотеки *libc* (листинг 6.3)

### Листинг 6.3. Программа, выводящая на экран приветствие (C++)

```
1 int main()
2 {
3     printf("Hello, \_world!\n");
4     return 0;
5 }
```

на тридцатидвухбитном ассемблере (как сказано в разделе 6.2, в этом случае для функций *libc* независимо от операционной системы используется соглашение *cdecl*) выглядит следующим образом (листинг 6.4).

### Листинг 6.4. Программа, выводящая на экран приветствие

```
1 .data
2     msg:
3     .string "Hello, \_world!\n"
4 .global main // точка входа в программу
```



```
5 main:
6     pushl $msg // Адрес строки в стек
7     call printf
8     popl %eax  // Вычищаем параметр из стека
9     xor %eax, %eax
10    ret
```

Воспользоваться в ассемблерной программе оператором вывода в поток << и потоком стандартного вывода `std::cout` затруднительно из-за декорирования имён. Функции библиотеки `libc` описаны как C-функции, поэтому их имена не декорируются (но в некоторых версиях Microsoft Windows могут дополняться префиксом, как описано в разделе 6.2.7).

Параметры функции `printf()` в соответствии с соглашением `cdecl` передаются через стек (`pushl $msg`) перед вызовом функции, а после завершения её работы вычищаются из стека вызывающей программой (`popl %eax`). Так как нам не нужно извлечённое из стека значение, команду `popl %eax` можно заменить изменением значения указателя стека `addl $4, %esp`. Эта команда быстрее и к тому же не требует приёмника для сохранения неиспользуемого значения.

Так как для вывода приветствия не используются все возможности `printf()`, вместо неё можно использовать более простую функцию `puts()`.

#### Листинг 6.5. Программа, выводящая приветствие с помощью `puts()`

```
1 .data
2     msg:
3     .string "Hello, \world!\n"
4 .global main // точка входа в программу
5 main:
6     pushl $msg // Адрес строки в стек
7     call puts
8     addl $4, %esp // Вычищаем параметр из стека
9     xor %eax, %eax
10    ret
```

### Форматированный вывод

Если необходимо передать функции вывода несколько параметров

**Листинг 6.6.** Программа, выводящая на экран два числа (C++)

```
1 int foo = 13;
2 int main()
3 {
4     printf("Переменные: %d %d\n", 19, foo);
5     return 0;
6 }
```

по соглашению `cdecl` эти параметры передаются в обратном порядке (то есть на вершине стека оказывается первый):

**Листинг 6.7.** Программа, выводящая на экран два числа

```
1 .data
2 fmt: .string "Переменные: %d %d\n"
3 foo: .int 13
4 .global main // точка входа в программу
5 main:
6     pushl foo      // Значение foo в стек
7     pushl $19      // Значение 19 в стек
8     pushl $fmt     // Адрес строки fmt в стек
9     call printf
10    addl $3*4, %esp // Три четырехбайтовых числа из стека
11    xor %eax, %eax
12    ret
```

Очистка стека здесь выполняется модификацией указателя стека *sp*, так как три команды `popl` выполнялись бы дольше.

Необходимо всегда помнить о том, что после вызова функции значения многих регистров (в частности, *A*) меняется (раздел 6.2.3). Если в таких регистрах хранятся данные вызывающей программы, их надо сохранить перед вызовом функции.

**6.1.2. Программирование без `libc`**

Минимальная программа с `libc` (листинг 6.1) после ассемблирования занимает 4704 байт. В этот размер входят библиотечные функции, обеспечивающие

обработку параметров, вызов стартовой функции *main()* и завершение программы после возврата управления из *main()*.

Отключить использование *libc* при сборке позволяет ключ *-nostdlib*. В этом случае взаимодействие с операционной системой, в том числе завершение программы, необходимо осуществлять напрямую, с помощью системных вызовов.

Точкой входа в этом случае будет непосредственно метка *\_start*.

Каждая операционная система имеет свой набор функций и свой способ их вызова (раздел 6.2.8). В большинстве тридцатидвухбитных операционных систем системные вызовы осуществляются с помощью программных прерываний. При этом, если в GNU/Linux, BSD и FreeDOS системные вызовы хорошо документированы, то в Microsoft Windows как механизм их вызова, так и набор функций постоянно меняются и скрыты от прикладного программиста. Вместо прямых системных вызовов под Microsoft Windows предлагается использовать функции библиотеки Windows API.

В GNU/Linux к функциям операционной системы в тридцатидвухбитном режиме можно обратиться с помощью прерывания 0x80. Номер функции указывается в регистре *eax*. Вызов может принимать до шести аргументов в регистрах *ebx*, *ecx*, *edx*, *esi*, *edi*, *ebp*.

### Минимальная программа без *libc*

Рассмотрим минимальную программу, не использующую функции *libc* (в том числе *main()*) для GNU/Linux. Сразу после запуска она должна завершиться с кодом 0. Для завершения программы используется системный вызов с кодом 1 — *sys\_exit()*. Его единственный параметр — код возврата.

### Листинг 6.8. Минимальная программа без *libc*

```
1 .globl _start
2 _start: // точка входа
3     movl $1, %eax // системный вызов №1 - sys_exit
```

```

4     xorl %ebx, %ebx // выход с кодом 0
5     int $0x80 // вызов ядра

```

Если код 6.8 сохранён как файл `nsmin.S`, собрать его без стандартной библиотеки можно командой:

```
1 $ gcc -o nsmin nsmin.S -nostdlib
```

полученный исполняемый файл занимает 600 байт.

## Приветствие миру

Для вывода строки на экран ядром Linux используется системный вызов с кодом 4 — `sys_write()`. Он предназначен для записи в файл; требует трёх параметров — дескриптор файла, указатель на начало записываемых данных и длина этих данных в байтах. Согласно концепции Unix всё есть файл; для вывода на экран используется специальный дескриптор 1 (`stdout`).

### Листинг 6.9. Вывод приветствия при помощи системных вызовов Linux

```

1 .data
2     msg:
3     .ascii "Hello, \world!\n"
4     len = . - msg // символу len присваивается длина строки
5 .global _start // точка входа в программу
6 _start:
7     movl $4, %eax // системный вызов №4 - sys_write
8     movl $1, %ebx // поток №1 - stdout
9     movl $msg, %ecx // указатель на выводимую строку
10    movl $len, %edx // длина строки
11    int $0x80 // вызов ядра
12    movl $1, %eax // системный вызов №1 - sys_exit
13    xorl %ebx, %ebx // выход с кодом 0
14    int $0x80 // вызов ядра

```

Для вывода более сложных данных, в частности, чисел, необходимо вручную сформировать в программе выводимую в файл `stdout` строку. Такой код, предназначенный для взаимодействия с системными вызовами, займёт существенный объём, так что в итоге выигрыш от неиспользования `libc` может оказаться несущественным.

## 6.2. Подпрограммы и функции

Рассказ есть зодчество из слов.

Зодчество из «рассказов» есть сверхповесть.

*В. Хлебников. Зангези*

Как сказано в разделе 4.1.6, одним из способов соединения кода на различных языках программирования является вызов внешних функций, описанных в модулях, соединяемых с головным на этапе компоновки.

Функции языка высокого уровня являются частным случаем **подпрограмм** — последовательностей команд, завершающихся командой возврата. Для вызова подпрограммы используется команда `call`, которая помещает в стек адрес возврата, а затем передаёт управление на начало вызываемой подпрограммы; для возврата управления вызывающей программе — команда `ret`, которая передаёт управление адресу, снятому со стека.

Таким образом, при вызове внешних функций необходимо решить четыре основные задачи. Первые две связаны с возможностью связать вызов функции из одного модуля с её описанием в другом на стадии компоновки; третья и четвёртая требуются для корректного взаимодействия вызывающей программы и функции на этапе выполнения.

1. Имена функций (как вызываемых из данного модуля, так и тех, которые описаны в данном модуле и могут быть вызваны извне) должны быть видимы для компоновщика.

Для этого используются ключевое слово `extern` в C++ и директива `.globl` в языке ассемблера.

2. Имя одной и той же функции на этапе компоновки должно выглядеть одинаково как в том модуле, где она описана, так и в том, где она вызывается. Для этого необходимо отказаться от такой возможности языка C++, как перегрузка (она приводит к декорированию имён) с помощью дополнитель-

ной строки "C" для ключевого слова `extern`. Кроме того, многие версии операционной системы Microsoft Windows требуют от компиляторов дополнительно искажать имена; это необходимо компенсировать вручную при помощи макросов.

3. Параметры должны помещаться вызывающей программой именно туда, где их будет искать вызываемая функция; возвращаемое значение функции также должно оказаться на том месте, где его ожидает вызывающая программа.

Для языков высокого уровня протокол взаимодействия вызывающей и вызываемой программ называется соглашением о вызовах. Используемое соглашение определяется платформой, операционной системой, языком высокого уровня, компилятором, а также специальными ключевыми словами при описании функции. Соответственно, функцией можно назвать подпрограмму, следующую необходимому соглашению о вызове.

4. Функция перед возвратом должна удалить из стека все те данные, которые она поместила поверх адреса возврата, и ни в коем случае не удалять больше, чем поместила.

Баланс стека должен быть сохранён, иначе во время возврата из функции управление будет передано не туда (а именно — по адресу, равному значению в текущей вершине стека), что приведёт к некорректной работе программы и, возможно, к её краху.

Рассмотрим процесс взаимодействия вызывающей программы и функции подробнее.

### **6.2.1. Требования к вызовам функций**

К механизму вызова подпрограммы (в различных языках используется также термины «функция», «метод», «процедура») можно сформировать ряд требований.

1. Возможность передачи управления на произвольный адрес.

2. Возврат управления назад после завершения подпрограммы.
3. Вложенные вызовы подпрограмм.
4. Сохранение и восстановление регистров вызывающей программы.
5. Передача заданного количества аргументов.
6. Возврат значения.
7. Выделение и освобождение памяти под локальные переменные подпрограмм.

В системе команд x86 реализованы только первые три из них. Обеспечить выполнение остальных можно только в том случае, если вызывающая и вызываемая программа «договорятся», где будут находиться передаваемые аргументы и локальные переменные.

В некоторых источниках считается, что функцией можно назвать только подпрограмму, написанную на языке высокого уровня. При этом подпрограмма, написанная на ассемблере и соответствующая используемому в данном языке высокого уровня соглашению о вызове, может быть вызвана наравне с написанными на ЯВУ. Таким образом, логичнее считать термин «функция» либо синонимом подпрограммы, либо обозначать им подпрограмму, соответствующую одному из общепринятых соглашений.

### 6.2.2. Механизм вызова подпрограммы

В системе команд x86 для реализации механизма подпрограмм используются всего две команды:

- команда вызова подпрограммы `call`, единственным аргументом которой является адрес начала подпрограммы;
- команда возврата из подпрограммы `ret`.

Пусть следующая команда, расположенная по адресу  $c_i$  — `call f` (рис. 6.1, а). Команда `call` помещает в стек адрес следующей по порядку команды  $c_{i+1}$  — **адрес возврата**, после чего в указатель команд  $ip$  помещается адрес  $f$ , так что эта команда становится следующей для исполнения процессором (рис. 6.1, б). Когда

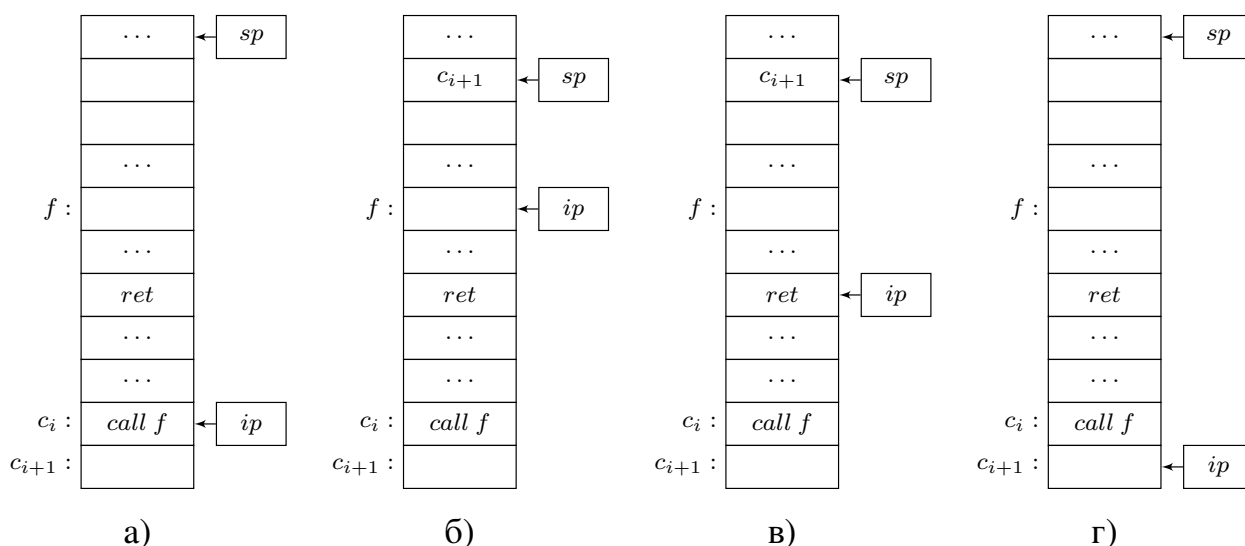


Рис. 6.1. Изменение указателя стека командами вызова  
и возврата

в процессе исполнения подпрограммы *f* встретится команда *ret* (рис. 6.1, в), из стека извлекается верхнее машинное слово — там должен быть адрес возврата — и помещается в указатель команд *ip* (рис. 6.1, г). Соответственно, выполнение вызывающей программы продолжится со следующей за *call* команды.

### Локальные переменные

Как уже говорилось в разделе 3.2, локальные переменные подпрограммы хранятся в стеке; также программист или оптимизирующий компилятор может поместить часть локальных переменных в регистрах общего назначения. Это не предписано системой команд, но является общепринятым.

Так как память под локальные переменные подпрограммы выделяется подпрограммой после передачи управления на её начало, эти переменные будут расположены в стеке над адресом возврата (так как стек растёт вниз — по меньшим адресам). Для выделения и освобождения памяти под переменные служат специальные фрагменты кода в начале и в конце подпрограммы — пролог и эпилог соответственно.



Современные компиляторы помещают в пролог команду уменьшения указателя стека  $sp$  на величину, равную общему объёму локальных переменных, помещаемых в стек (рис. 6.2, а) и б). В эпилоге указатель  $sp$  увеличивается на

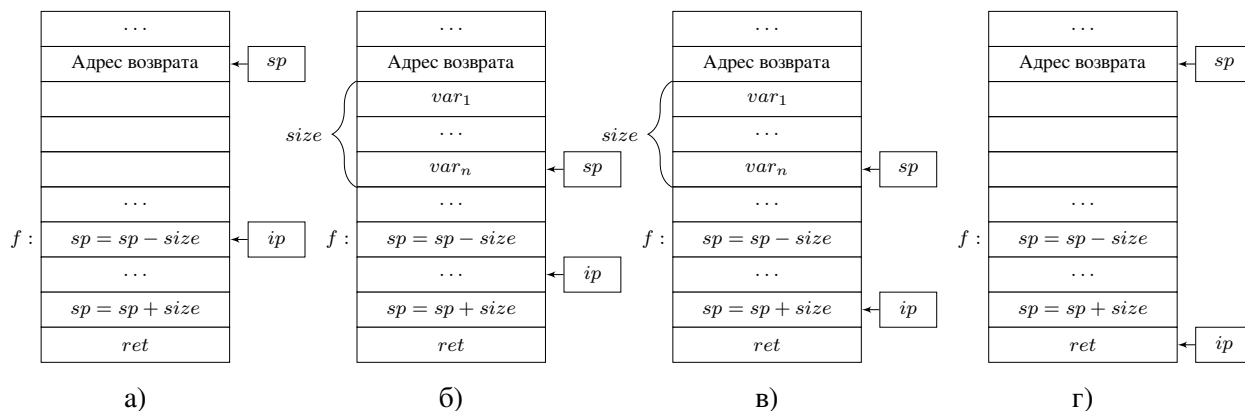


Рис. 6.2. Размещение локальных переменных в стеке  
оптимизирующим компилятором

ту же величину (рис. 6.2, в) и г), так что к моменту выхода из подпрограммы на вершине стека окажется адрес возврата (рис. 6.2, г).

Адреса локальных переменных можно выразить через  $sp$ . В процессе выполнения программы указатель  $sp$  может меняться, в этом случае необходимо соответственно корректировать смещения переменных относительно  $sp$ . Также, если код подпрограммы нарушает баланс стека, может понадобиться скорректировать величину, добавляемую к  $sp$  в эпилоге.

Ранние компиляторы сохраняли в регистре  $bp$  значение  $sp$  до резервирования памяти под локальные переменные. Это позволяло адресовать переменные относительно  $bp$  и не корректировать смещение, а в эпилоге восстановить значение  $sp$  из  $bp$  (рис. 6.3, а)–д). Это удлиняло пролог и эпилог за счёт необходимости сохранять и восстанавливать исходное значение  $bp$ . Кроме того, при этом невозможно использовать  $bp$  для других целей.

Размещение локальных переменных происходит внутри вызываемой подпрограммы и не затрагивает данные вызывающей программы. При ручном про-

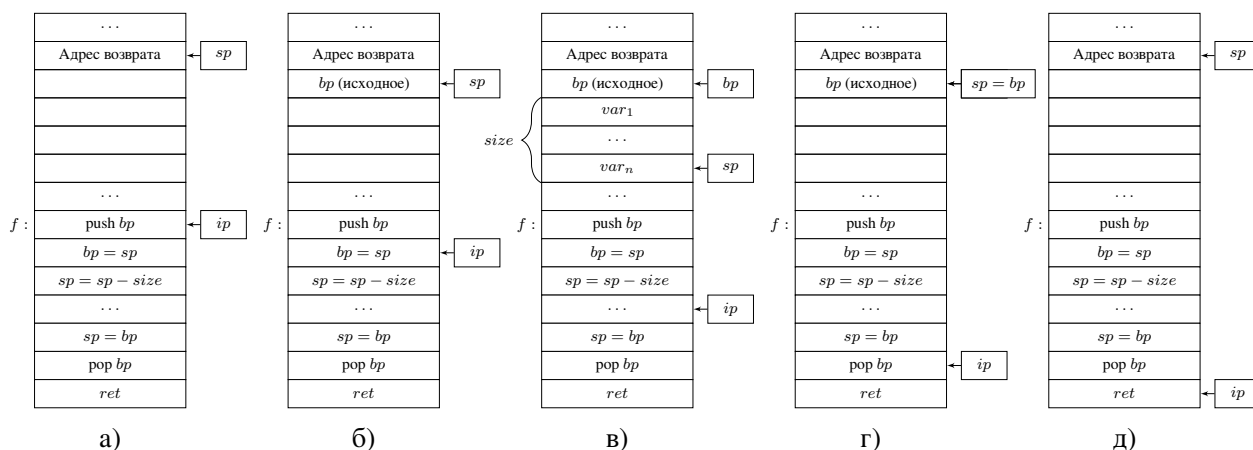


Рис. 6.3. Размещение локальных переменных в стеке  
(устаревший вариант)

граммировании можно использовать как современную, так и устаревшую форму пролога и эпилога; если все локальные переменные размещены в регистрах, пролог и эпилог вообще не требуются.

Следует отметить два момента:

- при компиляции с языка высокого уровня порядок локальных переменных в стеке может не совпадать с порядком их объявления;
- содержимое резервируемой в стеке памяти, как и начальное значение регистров, не определено, поэтому локальные переменные обязательно нужно инициализировать.

## Параметры и возвращаемое значение

Параметры также могут передаваться в подпрограмму через стек или регистры. Вызывающая программа должна разместить параметры в условленных местах до того, как управление будет передано подпрограмме. Соответственно, параметры, передаваемые через стек, окажутся под адресом возврата, то есть будут иметь большие адреса.

Возвращаемое значение функции не может передаваться вызывающей подпрограмме через стек, так как при выполнении команды `ret` в стеке не должно

остаться ничего после адреса возврата. Соответственно, возвращаемое значение может передаваться в вызывающую подпрограмму только через регистр.

Иногда значение, которое, согласно синтаксису языка высокого уровня, является возвращаемым, не может быть размещено в регистре (в частности, это может быть объект). В этом случае зарезервированное для него место (или его адрес) фактически передаётся как ещё один параметр.

Таким образом, «настоящее» возвращаемое значение может быть только числом. Если это целое число или указатель, в программах для x86 для возврата используется регистр *A*. Вещественное значение возвращается через вершину стека математического сопроцессора *st(0)* или через *x/y/zmm0*.

### 6.2.3. Соглашения о вызовах

Соглашение о вызовах определяет протокол взаимодействия вызывающей и вызываемой программ; в частности, необходимо согласовать следующие правила.

1. Способ передачи параметров (через стек, через регистры, смешанный; а также используемые регистры и их порядок).
2. Порядок размещения параметров в стеке (порядок Pascal подразумевает, что первый параметр помещается в стек первым, порядок C — что первый параметр помещается последним, непосредственно перед адресом возврата).
3. Как передаётся указатель *this* (для методов объекта).
4. Какие регистры могут изменяться подпрограммой.
5. Кто очищает стек и сохраняет/восстанавливает регистры.
6. Инструкции вызова и возврата из подпрограмм.
7. Возврат значения из подпрограммы (функции).

На платформе x86 для вызова и возврата из подпрограммы используются соответственно команды *call* и *ret*; а значение обычно возвращается через регистр *A*. Параметры обычно передаются либо через стек, либо смешанным спо-

собою: первые из тех, что можно разместить в отведённых регистрах, передаются через регистры, оставшиеся — через стек.

Остальные пункты по-разному реализованы в различных языках, компиляторах, операционных системах и для различной разрядности. Подробно эти различия рассмотрены в исследовании Агнера Фога [9].

В таблице 6.1 приведены наиболее популярные соглашения о вызовах, используемые на 32-битных платформах. Регистры для передачи параметров используются в указанном порядке. Если столбец «Параметры в регистрах» пуст, все параметры передаются через стек. Указатель *this* обычно передаётся первым параметром.

### Тридцатидвухбитные соглашения о вызовах

Таблица 6.1

Соглашение	Параметры в регистрах	Порядок	Очистка стека	Изменяемые регистры	Неизменяемые регистры
<b>cdecl</b>		C	вызывающая программа	<i>eax, ecx, edx, st(0)–st(7), x/y/zmm</i>	<i>ebx, ebp, esi, edi</i>
<b>pascal</b>		Pascal	функция		
<b>winapi (stdcall)</b>		C	функция		
<b>gnu</b>		C	this — функция, остальные — вызывающая программа		
<b>gnu fastcall</b>	<i>ecx, edx</i>	C	функция		
<b>gnu regparm (3)</b>	<i>eax, edx, ecx</i>	C	функция		
<b>Borland fastcall</b>	<i>ecx, edx</i>	Pascal	функция		
<b>Microsoft fastcall</b>	<i>ecx, edx</i>	C	функция		

Для соглашения **gnu regparm** можно указать количество параметров в регистрах (от одного до трёх).

Кроме того, регистры делятся на те, которые подпрограмма может изменять по своему усмотрению (соответственно, если они используются в вызывающей программе, вызывающей программе необходимо сохранить их перед обращением

к подпрограмме и восстановить после того, как подпрограмма закончит работу) и те, которые должны сохранить своё значение (если в подпрограмме потребуется использовать один из таких регистров, то сохранить и потом восстановить их исходное значение должна сама подпрограмма).

Согласно Фогу, в тридцатидвухбитных программах, как в Microsoft Windows, так и в Unix-подобных операционных системах (GNU/Linux, BSD, Mac OS X), подпрограмма может изменять регистры *eax*, *ecx*, *edx*, регистры сопроцессора *st(0) — st(7)* и регистры расширений *xmm/ymm/zmm*. Неприкосновенными должны остаться *ebx*, *ebp* и *esi*, *edi*.

На шестидесятичетырехбитных платформах применяется всего два соглашения о вызовах (таблица 6.2). К сожалению, они несовместимы между собой. Также для разных платформ теперь различается набор регистров, которые могут изменяться в подпрограмме. Регистры для передачи параметров используются в указанном порядке. Указатель *this* передаётся первым параметром.

### Шестидесятичетырехбитные соглашения о вызовах

Таблица 6.2

Соглашение	Параметры в регистрах	Порядок	Очистка стека	Изменяемые регистры	Неизменяемые регистры
<b>Microsoft Windows,</b> компиляторы MinGW, Microsoft, Intel	<i>rcx/zmm0</i> , <i>rdx/zmm1</i> , <i>r8/zmm2</i> , <i>r9/zmm3</i>	C	вызывающая программа	<i>rax, rcx, rdx</i> , <i>r8—r11</i> , <i>st(0)—st(7)</i> , <i>x/y/zmm</i> , кроме младших частей 6—15	<i>rbx, rbp</i> , <i>rsi, rdi</i> , <i>r12—r15</i> , <i>xmm6—xmm15</i>
<b>GNU/Linux, BSD, Mac OS X,</b> компиляторы GCC, Intel	<i>rdi, rsi</i> , <i>rdx, rcx</i> , <i>r8, r9</i> , <i>zmm0—zmm7</i>	C	вызывающая программа	<i>rax, rcx, rdx</i> , <i>rsi, rdi</i> , <i>r8—r11</i> , <i>st(0)—st(7)</i> , <i>x/y/zmm</i>	<i>rbx, rbp</i> , <i>r12—r15</i>

Как видно из таблицы 6.2, в 64-битном режиме под разными операционными системами в подпрограмме необходимо сохранять и восстанавливать разные регистры.

## Вызов подпрограммы в GAS

На тридцатидвухбитной платформе в GCC используются соглашения о вызове `gnu`, `cdecl`, `gnu fastcall`, `gnu regparm (0–3)`. Для внешних функций с отключённым декорированием (`extern "C"`) применяется только `cdecl`, то есть:

- размещение аргументов исключительно *в стеке*, без использования регистров, причём аргументы, меньшие 4 байт, расширяются до 4 байт;
- размещение аргументов в стеке таким образом, что первый аргумент оказывается на вершине стека;
- очистка стека выполняется *вызывающей программой*, так что в функции аргументы должны не сниматься со стека, а копироваться оттуда.

Размещение аргументов в стеке справа налево и очистка стека вызывающей программой позволяет определить функции с переменным количеством аргументов, такие, как `printf` и `scanf` из стандартной библиотеки C (**libc**), но надо помнить о небезопасности таких функций.

При вызове функции в стек сначала помещаются аргументы в соответствии с соглашением о вызовах, а затем команда вызова кладёт сверху адрес возврата. Соответственно, когда функция получает управление, то первые четыре байта по адресу, хранящемуся в `sp`, будут содержать адрес возврата. Далее идут аргументы функции. При использовании соглашения о вызовах `cdecl` непосредственно за адресом возврата (по адресу `sp + 4`) будет находиться первый параметр, за ним идёт второй и т. д. (рис. 6.4)

Регистры `B`, `bp`, `si`, `di` не должны изменяться подпрограммой. Возврат значения по возможности выполняется через регистры:

- `eax`, если результат — указатель или целое число до 4 байт (если результат меньше 4 байт, старшую часть `eax` необходимо обнулить);
- пара регистров `edx : eax`, если результат — целое число размером 8 байт;
- вершина стека сопроцессора, если результат — вещественное число;

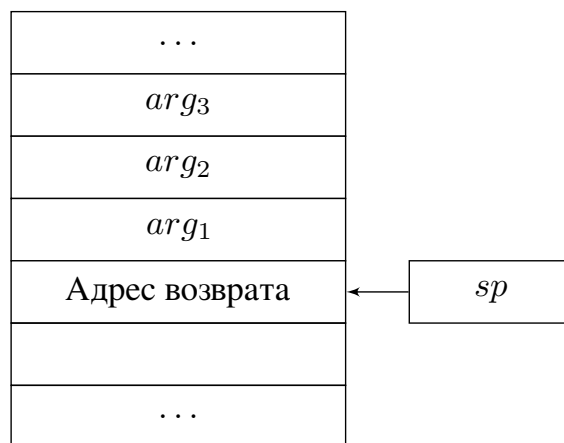


Рис. 6.4. Параметры и адрес возврата в соглашении cdecl

если результат не помещается в регистры, возвращается указатель на него (через *eax*).

На шестидесятичетырехбитной платформе в GCC для любых функций, в том числе внешних с отключённым декорированием (`extern "C"`), используется соглашение, соответствующее операционной системе.

#### 6.2.4. Описание функций на ассемблере

Пусть требуется описать функцию, рассчитывающую для целых беззнаковых чисел  $x, y$  значение  $z = 1 + x/8 + y$ . На языке высокого уровня она будет иметь вид, приведённый в листинге 6.10.

##### Листинг 6.10. Функция на языке C++

```

1 unsigned foo(unsigned x, unsigned y)
2 {
3     return 1 + x/8 + y;
4 };

```

Так как делитель  $8 = 2^3$  является степенью двойки, деление  $x/8$  можно заменить беззнаковым сдвигом  $x >> 3$ .

В принципе, если функция предназначена для использования внутри ассемблерного файла и гарантированно не будет вызываться языком высокого уровня, она может и не соответствовать стандартным соглашениям о вызове. Таким образом, можно реализовать собственные нестандартные соглашения, позволяющие,

в частности, передать параметры через регистры даже на тридцатидвухбитной платформе или вернуть несколько результатов в разных регистрах.

Тем не менее, если нестандартного поведения от функции не требуется, лучше использовать стандартные соглашения, так как в перспективе может понадобиться вызвать функцию из модуля на ЯВУ.

### Описание функции (cdecl)

Пусть используемая платформа тридцатидвухбитна и используется соглашение `cdecl`. Тогда на ассемблере код, рассчитывающий и возвращающий значение  $z = 1 + x/8 + y = x/8 + y + 1$ , в простейшем случае выглядит как в листинге 6.11.

#### Листинг 6.11. Простейшая функция на ассемблере

```
1 foo:
2  movl 4(%esp), %eax    // eax = x
3  shrl $3, %eax         // eax >>= 3
4  addl 8(%esp), %eax    // eax +=y
5  incl %eax             // ++eax
6  ret                  // возврат управления вызывающей программе
```

В первой строке находится метка, показывающая начало функции `foo()`.

Параметры, в соответствии с соглашением `cdecl`, находятся в стеке; при этом стек в GNU/Linux по умолчанию выравнен по *long*, то есть любое не более чем тридцатидвухбитное значение должно занимать 32 бита.

Таким образом, на вершине стека (ячейка `(%esp)`) находится адрес возврата; в ячейке памяти, смещённой на четыре байта относительно вершины стека `sp` (по адресу `sp + 4`, что обозначается как `4(%esp)`) — первый параметр  $x$ , по адресу `sp + 8` (ячейка `8(%esp)`) — второй параметр  $y$ . Оба параметра тридцатидвухбитны.

Команда `movl 4(%esp), %eax` копирует (move) четырёхбайтовый (что обозначается суффиксом *l*, то есть *long*) параметр  $x$  в регистр `eax`. Следующая команда выполняет беззнаковый, или логический сдвиг вправо (shift right) на три



бита, что эквивалентно беззнаковому делению на 8. После этого к *eax* добавляются параметр *y* и единица.

Возвращаемое значение, согласно соглашению, `cdecl`, должно находиться в регистре *eax*; результат вычислений находится именно там. В стеке нет ничего над адресом возврата, поэтому далее следует возврат из функции (`ret`).

### Вызов функции (`cdecl`)

Вызов описанной функции, в частности, расчёт значения `foo(17, 2)`, выглядит следующим образом (листинг 6.12).

#### Листинг 6.12. Вызов функции `foo()`

```
1 pushl $2           // второй параметр
2 pushl $17          // первый параметр
3 call foo           // вызов подпрограммы foo
4 add $8, %esp        // очистка стека от параметров
```

Команда `call`, в отличие от оператора вызова функции на ЯВУ, не позволяет передать параметры и получить возвращаемое значение. Она только помещает в стек адрес следующей после вызова подпрограммы команда, а затем передаёт управление на начало подпрограммы.

Таким образом, перед вызовом функции командой `call` параметры необходимо вручную поместить туда, где их ожидает увидеть подпрограмма. Для соглашения `cdecl` они должны находиться в стеке. Соответственно, фактические параметры 17 и 2 нужно загрузить в стек как четырёхбайтовые целые числа, что и делает команда `pushl`.

После завершения работы такой функции параметры необходимо вручную удалить из стека; для этого к указателю стека добавляется общий размер параметров — восемь байтов.

Возвращаемое значение типа *unsigned int* можно найти, в соответствии с соглашением о вызове, в регистре *A* (*eax*).

## Сохранение изменяемых функцией регистров

Если вызывающая подпрограмма хранит какие-то долгоживущие данные в регистре, изменяемом подпрограммой (для тридцатидвухбитных соглашений это *eax*, *ecx*, *edx*, регистры сопроцессора  $st(0) - st(7)$  и регистры расширений  $xmm/ymm/zmm$ ), их необходимо сохранить перед вызовом функции и восстановить после её завершения.

В частности, в листинге 6.13 сохраняется и восстанавливается регистр *eax*. Так как в этом регистре по соглашению после выхода из функции содержится её возвращаемое значение, оно (если оно требуется вызывающей программе) должно быть скопировано или использовано до восстановления оригинального значения *eax*.

### Листинг 6.13. Вызов функции *foo()* с сохранением *eax*

```
1 z: .long 0
2 ...
3 pushl %eax      // сохранение eax
4 pushl $2        // второй параметр
5 pushl $17       // первый параметр
6 call foo        // вызов подпрограммы foo
7 add $8, %esp    // очистка стека от параметров
8 movl %eax, z    // z = foo()
9 popl %eax       // восстановление eax
```

При этом значения регистров, которые по соглашению могут быть изменены функцией, необходимо сохранять, даже если текущая реализация конкретной функции *foo()* их не меняет (в следующей версии функции они могут быть использованы).

Наилучшим выходом будет использовать изменяемые функциями регистры только как временные и не хранить там долгоживущие переменные.

## Локальные переменные

Пусть требуется использовать в функции *foo()* локальную переменную *t*. Например, зададим вначале  $t = 1$ , затем добавим к этому значению  $x/8$  и  $y$  (хотя

для таких простых вычислений введение переменной в памяти явно избыточно, данная функция не использует даже всех разрешённых по соглашению регистров). Получим листинг 6.14.

#### Листинг 6.14. Функция с локальной переменной

```
1 foo:
2   pushl $1          // пролог: создание t = 1
3
4   movl 8(%esp), %eax // eax = x
5   shrl $3, %eax      // eax >>= 3
6   addl %eax, (%esp)  // t += eax (t = 1 + x/8)
7
8   movl 12(%esp), %eax // eax = y
9   addl %eax, (%esp)  // t += eax (t = 1 + x/8 + y)
10
11  movl (%esp), %eax   // eax = t
12  add $4, %esp        // эпилог: уничтожение t
13  ret
```

У функции появляются пролог и эпилог. В листинге 6.14 показана современная форма пролога и эпилога, **используемая**, когда все локальные переменные и параметры адресуются через *sp*.

Так как после пролога (создания стековой локальной переменной *t*) значение указателя стека *sp* уменьшилось на  $sizeof(t) = 4$ , смещения параметров *x* и *y* относительно *sp* здесь отличаются от листинга 6.11 (увеличились на те же 4 байта).

#### 6.2.5. Импорт функций из модулей на ассемблере в код на C++

Функции, описанные в ассемблерном модуле, необходимо описать в коде C++ как внешние (*extern*). После спецификатора *extern*, согласно стандарту C++, могут быть указаны строки "C++" (подразумевается по умолчанию) или "C" (различные компиляторы могут поддерживать и иные строки) для указания компоновщику, какой язык использовался при написании внешней функции. Конкретные свойства таких функций не описываются в стандарте. На прак-

тике "C++" подразумевает искажение имён функций до неузнаваемости (что подробнее описано в разделе 6.2.7) и, для тридцатидвухбитных систем, передачу параметров по возможности через регистры, что реализуется в разных компиляторах по-разному. Указание "C" уменьшает искажение имён и для тридцатидвухбитных платформ подразумевает соглашение о вызовах `cdecl` (для шестидесятичетырехбитных платформ C-функции, как и C++-функции, следуют соглашению, соответствующему операционной системе).

Рассмотрим функцию `int sqr(int x)`, вычисляющую квадрат целого числа на тридцатидвухбитной платформе GNU/Linux [64]. Текст ассемблерного модуля `sqr.S` с текстом функции приведён в листинге 6.15.

#### Листинг 6.15. Файл `sqr.S`

```
1 .globl sqr
2     sqr:
3         movl 4(%esp), %eax
4         imull %eax, %eax
5         ret
```

Функция `sqr()` копирует единственный параметр  $x$  типа `int` (что для 32-битной платформы обычно эквивалентно 32-битному целому числу со знаком) из стека (соглашение `cdecl`) в регистр `A`, затем умножает его сам на себя. Результат  $x^2$  при этом остаётся в регистре `A`. По соглашению значение функции возвращается именно через этот регистр, так что дополнительных действий для формирования возвращаемого значения предпринимать не нужно; сразу после умножения управление возвращается вызывающей программе.

Директива `.globl` делает функцию `sqr()` видимой для внешних модулей.

Основной файл `main.cpp` содержит импорт функции `sqr()` и её вызов из головной функции `main()`.

#### Листинг 6.16. Файл `main.cpp`

```
1 #include <iostream>
2
```

```
3 using namespace std;
4 extern "C" int sqr(int x);
5
6 int main()
7 {
8     int x=2, y;
9     y = sqr(x);
10    cout << y <<endl;
11    return 0;
12 }
```

Для импорта функции из другого объектного файла здесь она описана как внешняя (`extern "C"`). В GNU/Linux имена внешних C-функций не искажаются, поэтому в модулях `main.cpp` и `sqr.S` используется одно и то же имя `sqr()`.

В шестидесятичетырехбитных системах импорт функций в код на C++ выполняется таким же образом. В ассемблерном файле экспорт имени выполняется так же, как и в листинге 6.15, а вот параметры шестидесятичетырехбитной функции извлекаются не из стека, а из регистров (это связано с тем, что используются соглашения о вызовах, отличные от `cdecl`).

#### 6.2.6. Импорт функций из модулей на C++ в код на ассемблере

Подпрограмма на ассемблере может обращаться не только к другим подпрограммам из того же модуля на ассемблере, но и к внешним, в частности, к функциям из других объектных файлов проекта или к стандартной библиотеке `libc`.

Для того, чтобы функция, описанная на языке C++, была доступна для экспорта в другие модули, используется ключевое слово `extern`, как и для импорта внешних функций (для отключения декорирования также используется `extern "C"`):

#### Листинг 6.17. Файл `bar.cpp`

```
1 extern "C" int bar(int x)
2 {
3     return 3*x+1;
```

```
4 };
```

Для импорта функции в ассемблере не требуется никаких директив, достаточно знать её имя. Для работы с параметрами, если они есть, необходимо также знать соглашение о вызове. Для тридцатидвухбитных систем и C-функций это `cdecl`, то есть параметры передаются через стек.

#### Листинг 6.18. Фрагмент файла `main.S`

```
1 pushl $1
2 call bar
3 add $4, %esp
```

Приведённый фрагмент кода рассчитывает `bar(1)`. Возвращаемое значение типа `int` можно найти, в соответствии с соглашением о вызове, в регистре `A`.

Здесь также во всех модулях используется одно и то же имя `bar`, так как в GNU/Linux имена внешних C-функций не искажаются.

Все функции стандартной библиотеки C (`libc`) описаны как внешние C-функции, поэтому их также можно, аналогично функции `bar()`, вызывать по оригинальному (в GNU/Linux — вообще неискажённому) имени.

При вызове функции с несколькими параметрами на тридцатидвухбитной платформе необходимо помнить, что в соответствии с соглашением `cdecl` они должны находиться в стеке в обратном порядке.

#### Листинг 6.19. Вывод двух чисел на экран

```
1 fmt: .string "Переменные:_%d,_%d\n"
2 foo: .int 13
3 ...
4 // printf(fmt, 19, foo)
5 pushl foo          // Значение foo в стек
6 pushl $19          // Значение 19 в стек
7 pushl $fmt         // Адрес строки fmt в стек
8 call printf
9 addl $3*4, %esp     // Три четырёхбайтовых числа из стека
```

Приведённый код выведет на экран сообщение «Переменные: 19, 13».

На шестидесятичетырехбитной платформе параметры передаются через регистры в соответствии с операционной системой (раздел 6.2.3).

### 6.2.7. Искажение имён при компиляции

Имена функций, задаваемые программистом, в процессе компиляции искажаются; наиболее заметные изменения связаны с процессом **декорирования** (name mangling). Механизм декорирования имён отсутствовал в языке С. Язык С++, в отличие от С, поддерживает перегрузку функций, то есть программа, написанная на С++, может содержать множество функций, носящих одно и то же имя. При этом на этапе компоновки для корректной сборки программы у каждой функции должно быть уникальное имя. Соответственно, на этапе компиляции в имя каждой функции включается информация о всех её параметрах (явных и неявных), причём так, что в изменённом имени используются только допустимые символы.

Алгоритм декорирования не стандартизирован; различные компиляторы для различных платформ используют разные схемы.

Декорирование имён можно отключить вместе с возможностью перегрузки, объявив функцию внешней С-функцией `extern "C"`. Большинство современных операционных систем вообще не искажает имён таких функций (рис. 6.5, а). В Microsoft Windows к имени функции даже при отключённом декорировании может быть приписан префикс или суффикс. Конкретный способ искажения зависит от разрядности, компилятора и версии Windows.

Чаще всего в Microsoft Windows при компиляции к именам добавляется префикс `_` (ведущее нижнее подчёркивание). Имена функций стандартной библиотеки `libc` также подвергаются искажению (то есть `printf` преобразуется в `_printf`). На некоторых (но не на всех) 64-разрядных версиях Microsoft Windows при использовании GCC имена не искажаются.

Если имя функции искажается компилятором, описанный в предыдущих разделах код вызовет ошибку компоновки (рис. 6.5, б).

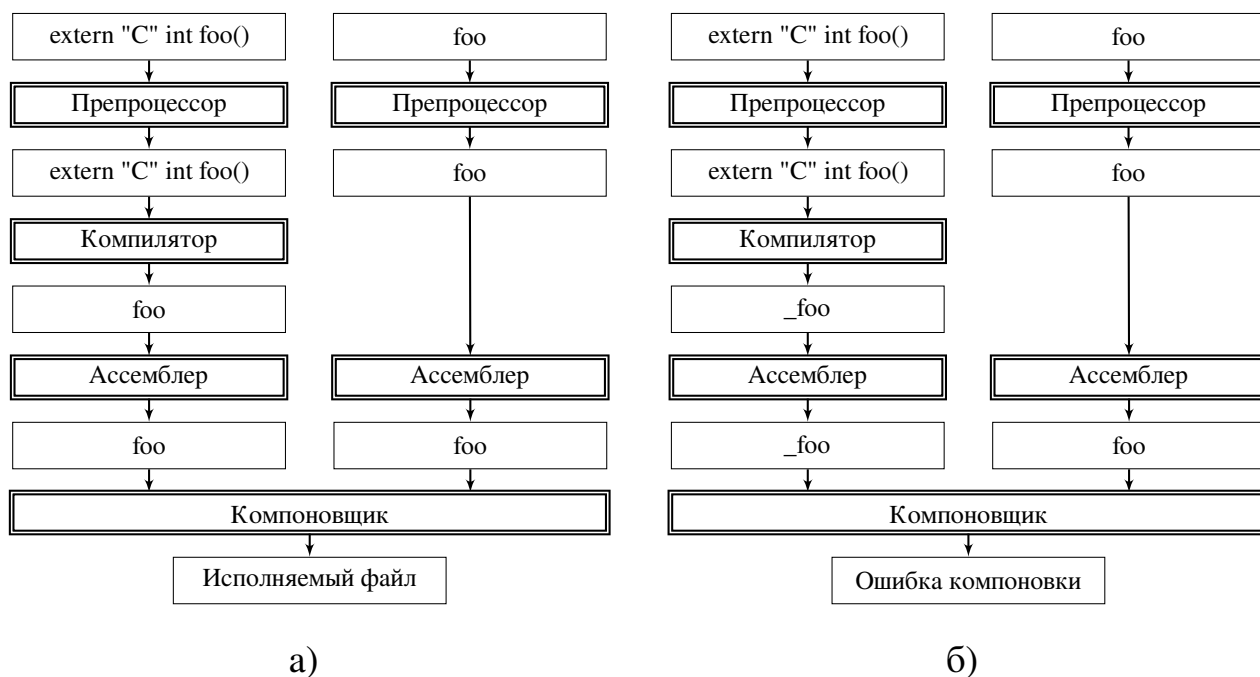


Рис. 6.5. Компиляция C-функции без искажения имён (а)

и с искажением, принятым в большинстве версий

Microsoft Windows (б)

Если используемый компилятор для данной платформы изменяет имена внешних C-функций, например, добавляет префикс `_`, как в случае `gcc` (`mingw`) под Microsoft Windows, то ссылка на внешнюю функцию `extern "C" int sqr(int x)`, упоминаемую в некотором `cpp`-файле, в соответствующем объектном файле будет выглядеть как `_sqr`. Таким образом, для корректной компоновки под Microsoft Windows имя данной функции в объектном файле, полученном из ассемблерного модуля, также должно быть `_sqr`. Так как на этапе ассемблирования имена не искажаются ни на какой платформе, это означает, что в ассемблерном модуле имя также должно выглядеть как `_sqr`.

При этом в других операционных системах имена не искажаются, так что для корректной компоновки в GNU/Linux имя этой же функции должно выглядеть как `sqr`.



Если речь идёт об импорте в C++ функции, описанной на ассемблере, то в ассемблерном файле можно указать оба варианта имени (листинг 6.20).

**Листинг 6.20.** Файл `sqr.S`: экспорт вариантов без искажения и с ним

```
1 .globl sqr
2 .globl _sqr
3 sqr:
4 _sqr:
5     movl 4(%esp), %eax
6     imull %eax, %eax
7     ret
```

В этом случае код на C++ (листинг 6.16) соберётся с модулем 6.20 как под GNU/Linux или BSD, так и под Microsoft Windows, но это не лучшее решение.

В случае импорта в программу на ассемблере функции из модуля на C++, в частности, функции `int bar(int x)`, описанной в листинге 6.17, уже невозможно описать два имени одновременно. Вызов подобной функции из программы на ассемблере будет выглядеть как `call bar` под GNU/Linux или BSD и как `call _bar` под тридцатидвухбитными и некоторыми шестидесятичетырехбитными версиями Microsoft Windows.

Как правило, программа вызывает подобные функции из множества мест, так что править её вручную затруднительно.

Кроссплатформенности в этом случае можно достичь использованием макросов для искажения имён в `.S`-файле вручную (рис. 6.6, а) и б). Опишем макрос `FUNC_NAME` с параметром `s` — именем функции, который либо добавляет к `s` ведущее подчёркивание (тогда его нужно описать как `#define FUNC_NAME(s) _##s`), либо ничего с `s` не делает (`#define FUNC_NAME(s) s`) и вместо имени функции, например, `bar`, будем указывать `FUNC_NAME(bar)`.

В случае тридцатидвухбитных систем подчёркивание необходимо добавлять при работе под Microsoft Windows и не нужно для остальных операционных

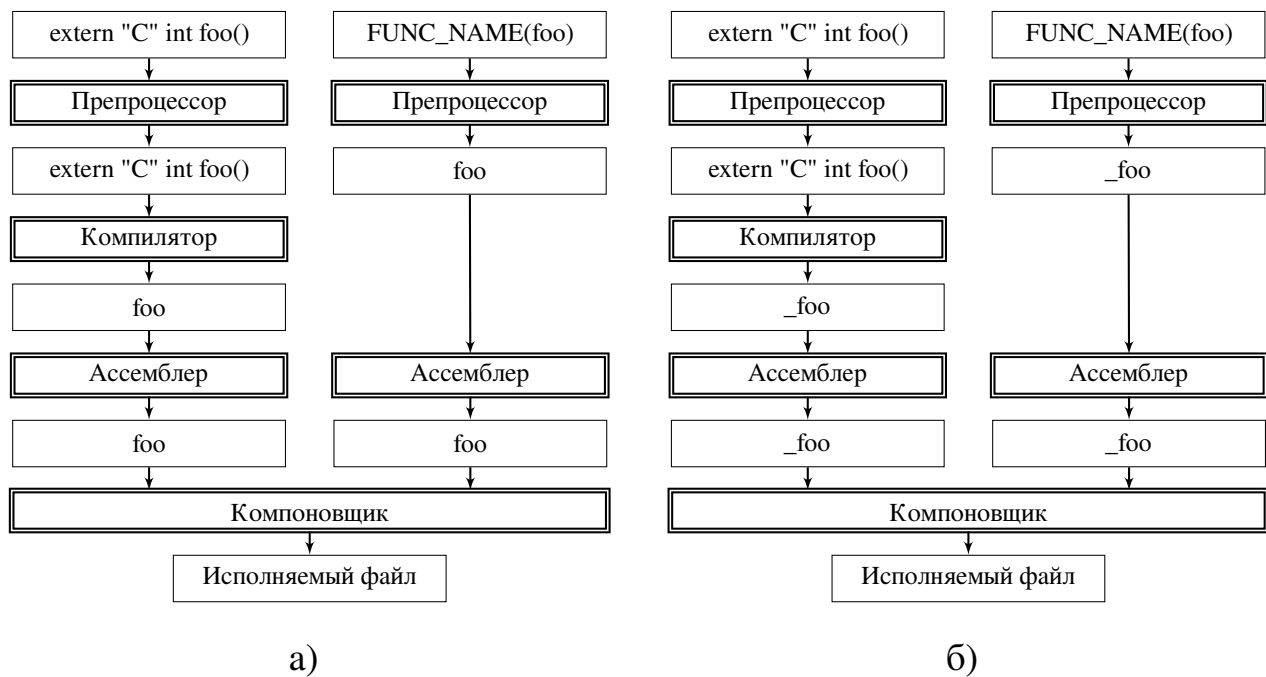


Рис. 6.6. Компиляция C-функции без искажения имён (а)  
и с компенсацией искажения (б)

систем. Для этого можно воспользоваться предопределённым макросом препроцессора `_WIN32`:

```

1 #ifdef _WIN32
2 #define FUNC_NAME(s) _##s
3 #else
4 #define FUNC_NAME(s) s
5 #endif

```

Данное определение можно поместить в начало ассемблерного файла с расширением `.S` или в файл, включаемый в него директивой `#include`.

После этого макрос `FUNC_NAME` можно применять для **компенсации** искажения имён во всех случаях — при экспорте имени функции (листинг 6.21):

#### Листинг 6.21. Файл `sqr.S`: кроссплатформенный экспорт

```

1 .globl FUNC_NAME(sqr)
2 FUNC_NAME(sqr):
3     movl 4(%esp), %eax
4     imull %eax, %eax

```

```
5    ret
```

при импорте (листинг 6.22):

**Листинг 6.22.** Фрагмент файла `main.S`: кроссплатформенный импорт

```
1 pushl $1
2 call FUNC_NAME(bar)
3 add $4, %esp
```

или при вызове библиотечных функций (листинг 6.23).

**Листинг 6.23.** Вывод двух чисел на экран

```
1 pushl foo           // Значение foo в стек
2 pushl $19           // Значение 19 в стек
3 pushl $fmt          // Адрес строки fmt в стек
4 call FUNC_NAME(sprintf)
5 addl $3*4, %esp     // Три четырёхбайтовых числа из стека
```

Шестидесятичетырехбитный перенос функций из GNU/Linux в Microsoft Windows или наоборот сложнее. Во-первых, не все шестидесятичетырехбитные версии Microsoft Windows искажают имена, так что нужный вариант макроса `FUNC_NAME`, возможно, придётся выбирать вручную. Во-вторых, если на тридцатидвухбитных платформах для внешних C-функций использовалось соглашение `cdecl`, то здесь соглашения для разных операционных систем разные, так что для достижения кроссплатформенности придётся описать два варианта копирования параметров из регистров, воспользовавшись директивами условной компиляции.

### 6.2.8. Системные вызовы

Системный вызов (`system call`) — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции. В принципе, системные вызовы соответствуют определению подпрограмм, но, так как ядро системы работает в привилегированном режиме, нельзя давать программе возможность передать управление на произвольный фрагмент кода ядра. Соответственно, количество функций, выполняемых ядром, ограничено. Сами эти функции, как правило, пронумерованы и для обращения к ним необходим номер,

а не адрес. Кроме того, для системных вызовов используются другие соглашения; в частности, вместо `call/ret` применяются другие команды вызова и возврата.

Для обращения к ядру используются следующие способы:

1. Программное прерывание (команда `int`) — этот способ доступен на всех x86-совместимых системах. Как правило, для вызова всех функций ядра используется какой-то один номер прерывания, а номер самой функции передаётся через один из регистров.
2. Быстрый вызов ядра в тридцатидвухбитном режиме выполняется командой `sysenter`.
3. Быстрый вызов в шестидесятичетырехбитном режиме — командой `syscall`.

## Системные вызовы различных ОС

Как и для функций, соглашение о вызовах описывает команду вызова, а также передачу параметров и возврат значения. Ядра различных операционных систем поддерживают различные соглашения.

- к ядру Linux на платформе x86 можно обратиться по программному прерыванию `int 0x80`, на x86-64 — с помощью команды `syscall`, номер функции и параметры передаются в регистрах;
- BSD на x86 также использует `int 0x80`, номер функции в регистре `eax`, параметры в стеке;
- FreeDOS предоставляет большую часть функций через `int 0x21`, также используются `int 0x20` (завершение программы), `int 0x29` (печать символа), `int 0x2E` (выполнение команды).

В большинстве операционных систем функции, выполняемые ядром, документированы и могут быть напрямую вызваны прикладной программой. Операционная система Microsoft Windows, напротив, скрывает их и меняет от версии к версии как сами функции, так и механизм их вызова. Прикладным програм-

мам предлагается использовать обёртки для обёрток над обёртками системных вызовов ядра Windows — функции Windows API из разделяемых библиотек.

Тем не менее, в различных источниках сообщается, что линейка Microsoft Windows NT/2000/XP/2003/Vista использует прерывание `int 0x2E`, а в Microsoft Windows XP/7/8 для обращения к ядру используется команда `sysenter` в тридцатидвухбитной версии и `syscall` в шестидесятичетырехбитной.

## Системные вызовы Linux

Для обращения к ядру Linux в тридцатидвухбитном режиме используется вызов программного прерывания `int 0x80`, при этом в регистре *eax* должен быть номер функции. В частности, номеру 1 соответствует вызов `sys_exit()`, 2 — `sys_fork()`, 3 — `sys_read()`, 4 — `sys_write()` и т. д.

Системный вызов принимает до шести аргументов в регистрах *ebx*, *ecx*, *edx*, *esi*, *edi*, *ebp*. В частности, при завершении программы системным вызовом `sys_exit()` (*eax* = 1) в *ebx* должен находиться единственный параметр этого вызова — код завершения программы. Если требуется передать функции ядра от семи параметров и выше, из них формируется структура, адрес которой передаётся в *ebx*.

Результат помещается в *eax*.

В шестидесятичетырехбитном режиме обращение к ядру Linux выполняется командой `syscall`. Номер вызова также должен быть в регистре *rax*, но сами номера иные, чем для тридцатидвухбитного ядра: 0 — `sys_read()`, 1 — `sys_write()`, 2 — `sys_open()`, 3 — `sys_close()` и т. д.

Параметры передаются в регистрах *rdi*, *rsi*, *rdx*, *r10*, *r8*, *r9*.

Возвращаемое системным вызовом значение помещается в *rax*.

### 6.3. Программирование нелинейных алгоритмов и взаимодействие со структурами данных

Слѡва — нет, есть движения в пространстве и его части —  
точек, площадей.

*В. Хлебников. Зангези*

В языках высокого уровня существуют операторы цикла и условные операторы, используемые для реализации нелинейных алгоритмов. Система машинных команд предлагает только команды условного и безусловного перехода (хотя в наборе команд x86 есть команда `loop`, она не является полноценным аналогом цикла, к тому же не рекомендуется к использованию из-за медленной работы).

Тем не менее, с помощью команд перехода можно реализовать все нелинейные алгоритмические конструкции, а режим косвенной адресации позволяет обращаться к элементам составных структур данных. Для визуализации нелинейных алгоритмов воспользуемся схемами программы [30].

#### 6.3.1. Условие с операторами в одной ветви

Пусть необходимо заменить значение целой знаковой переменной  $x$  нулѐм, если оно отрицательно.

$$x = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \quad (6.1)$$

На языке C++ код, решающий эту задачу, будет использовать условный оператор (листинг 6.24).

#### Листинг 6.24. Условие (6.1), C++

```
1 if (x < 0)
2 {
3     x = 0;
4 }
```

Так как условный оператор включает только ветвь «да», а в ней — только простой оператор присваивания, эту конкретную задачу можно решить с помощью команд условной пересылки. Начиная с Pentium Pro, это наиболее быстрый способ присваивания по условию.

Кроме того, можно воспользоваться командой условного перехода для обхода кода ветви «да» в случае, когда условие не выполнено. Этот способ не только поддерживается начиная с оригинального 8086, но и универсален. Так можно реализовать условный оператор со сколь угодно сложным кодом внутри.

### Условная пересылка

Сравним  $x$  с нулём (оценим знак разности  $x - 0$ ). Присваивание необходимо выполнить в случае, когда  $x < 0$  как знаковое. Это соответствует условию `l` (*if less*) или, что то же самое, `nge` (*if not greater or equal*). С учётом размера типа `int` можно также добавить суффикс размера `l` (листинг 6.25).

#### Листинг 6.25. Условие (6.1), `cmovcc`

```
1 int x = -6;
2 asm
3 (
4     "cmpl_$0, %[X] \n"
5     "cmovnge_l %[Zero], %[X] \n"
6     : [X] "+r"(x)
7     : [Zero] "rm"(0)
8     : "cc"
9 );
```

Так как источником для команды `cmovcc` не может быть константа, вводится входной параметр `[Zero]`, расположенный в памяти или регистре и равный нулю.

### Обход части операторов с помощью команд условного перехода

Построим схему алгоритма (рис. 6.7, а) для решения этой задачи.

В отличие от операторов `if` большинства языков высокого уровня, ветвь, где должны выполняться операторы, лучше сделать ветвью «нет» (таким образом, по ветви «да» будет выполняться обход части программы).

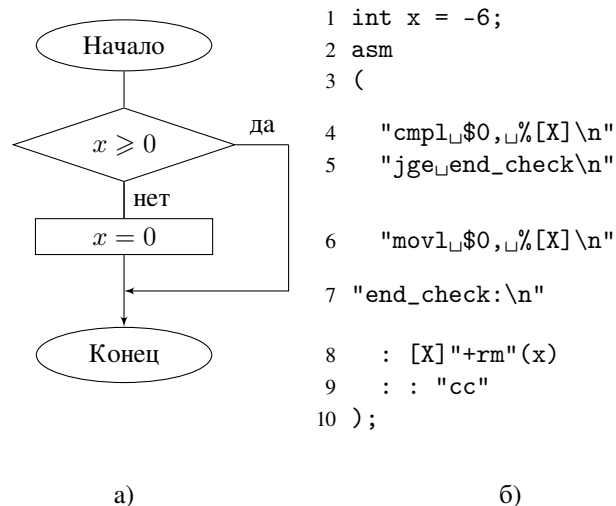


Рис. 6.7. Алгоритм и реализация ветвления с операторами  
в одной ветви

Построим по данной схеме программу (рис. 6.7, б). Основная вертикаль схемы будет соответствовать последовательности команд в памяти, потоки, отходящие от вертикали — командам передачи управления, потоки, входящие в основную вертикаль — меткам, на которые передаётся управление. Если команда не является командой передачи управления, после неё выполняется непосредственно следующая, что соответствует движению вниз по основной вертикали.

Каждому блоку процесса здесь будет соответствовать линейный фрагмент кода (строка 6); блок решения включает установку флагов и завершится условным переходом по метке (строки 4-5); точка соединения двух потоков соответствует метке на строке 7.

Подобным способом можно реализовать условный оператор со сколь угодно объёмным кодом только в одной ветви. Необходимый код помещается на место строки 7, между командой условного обхода ветви и соответствующей меткой.



### 6.3.2. Условие с операторами в двух ветвях

Пусть в зависимости от условия необходимо выполнить один из двух операторов:

$$y(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \quad (6.2)$$

На языке C++ код, решающий эту задачу, будет использовать условный оператор с двумя ветвями (листинг 6.26).

#### Листинг 6.26. Условие (6.2), C++

```
1 if (x >= 0)
2 {
3     y = 1;
4 }
5 else
6 {
7     y = 0;
8 }
```

Из-за специфического кода ветвей данную задачу можно решить на ассемблере тремя способами.

#### Условная установка байта

Так как  $y$  равен либо нулю, либо единице, задачу можно решить с помощью команды условной установки байта.

Единичное значение  $y$  соответствует случаю, когда  $x \geq 0$  как знаковое (*if greater or equal*), то есть мнемоника условия может быть записана как *ge*.

#### Листинг 6.27. Условие (6.2), setCC

```
1 int x = -7, y;
2 asm
3 (
4     "cml_1_$0, _1%[X]\n"
5     "movl_1_$0, _1%[Y]\n"
6     "setgeb_1%[Y]\n"
7     : [Y] "=m" (y)
8     : [X] "rm" (x)
```

```

9      : "cc"
10 );

```

Здесь мы записываем ноль в четырёхбайтовую переменную `[Y]`, находящуюся в памяти (это не изменяет флагов), а затем устанавливаем по условию её младший байт (так как платформа x86 использует порядок байтов Intel, адрес младшего байта `[Y]` совпадает с адресом `[Y]`). Оба возможных значения `y` неотрицательны, поэтому дополнение нулями является корректным расширением и для знаковой, и для беззнаковой их интерпретации.

Также можно было использовать для хранения параметра `[Y]` регистр `A`, установить по условию байт `al` и специальными командами расширить его вначале до `ax`, а затем до четырёхбайтового `eax`.

При использовании для `y` однобайтового типа `char` вместо четырёхбайтового `int` расширение не будет нужным.

### Условная пересылка

Данную задачу также можно решить с помощью команд условной пересылки. Для подобного задания такая реализация будет компактнее и быстрее.

#### Листинг 6.28. Условие (6.2), `cmovcc`

```

1 int x = 10, y;
2 asm
3 (
4     "cmpl_$0, %[X]\n"
5     "movl_$1, %[Y]\n"
6     "cmovll %[Zero], %[Y]\n"
7     : [Y] "+r"(y)
8     : [X] "rm"(x), [Zero] "rm"(0)
9     : "cc"
10 );

```

Здесь параметр `[Y]` объявлен как выходной с начальным значением, хотя, казалось бы, его исходное значение нам не требуется. Это сделано, чтобы компилятор не совместил его с входным параметром `[Zero]` (совмещение с входным пара-

метром  $[X]$  не повредит выполнению, так как значение  $[X]$  используется только для сравнения с литералом  $\$0$ , до первого присваивания  $[Y]$ ). Если  $[Y]$  будет совмещён с  $[Zero]$ , мы всегда получим на выходе  $y = 1$ .

### Обход части операторов с помощью команд условного перехода

Построим схему алгоритма (рис. 6.8, а) для решения этой задачи и затем выстроим блоки линейно вдоль одной вертикали (рис. 6.8, б).

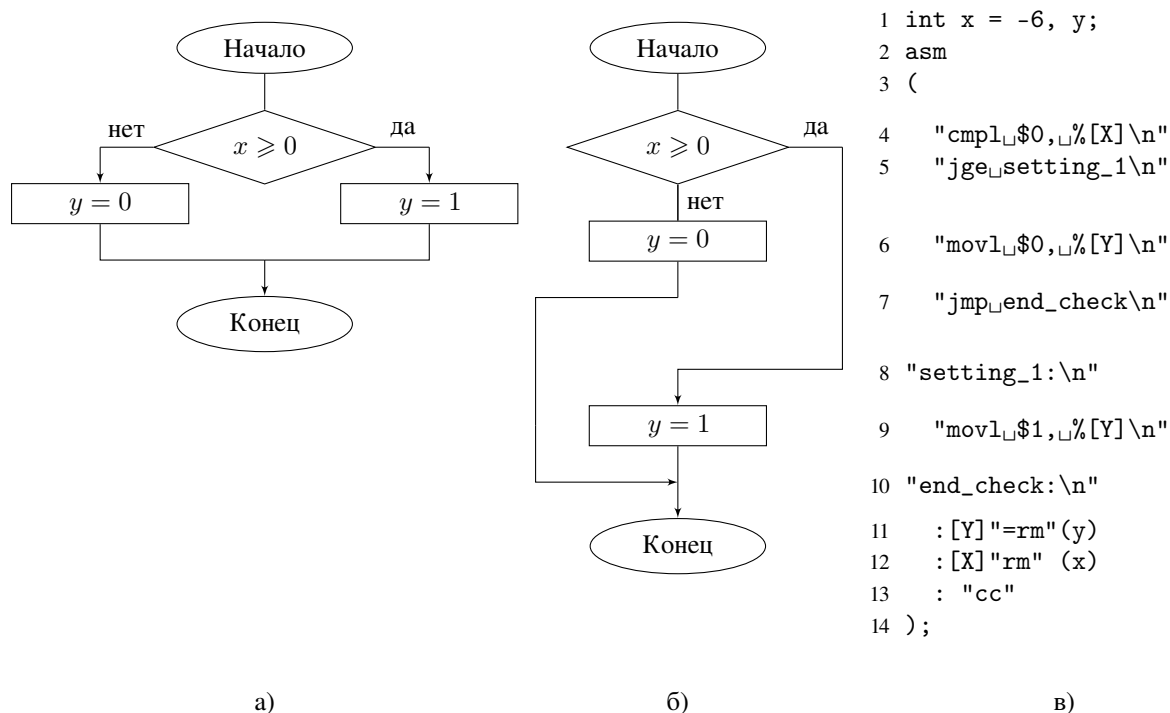


Рис. 6.8. Алгоритм и реализация ветвления

Схема на рис. 6.8, б) не вполне соответствует ЕСПД (стандарт предписывает минимизировать количество изломов и не удлинять линии потоков без необходимости, так как схемы алгоритмов предназначены в основном для чтения человеком) но при таком расположении можно однозначно сопоставить расположение блоков на схеме и расположение команд в памяти.

У схемы на рис. 6.8, а) два возможных варианта «вертикализации» — изображённый на рис. 6.8, б) и тот, где ветвь «да» окажется выше ветви «нет». Эти варианты равноправны и оба приведут к одинаково корректным, но различным

между собой программам. В данном разделе рассматривается вариант рис. 6.8, б).

Сопоставим каждому блоку одну или несколько команд ассемблера. В этом случае излом потока (отход от базовой вертикали без ветвления) будет соответствовать безусловному переходу, блок решения включает вычисление условия и условный переход, а точки входа боковых потоков-переходов в основной вертикальный поток соответствуют меткам (рис. 6.8, в).

Таким образом, код ветви «нет», расположенной на базовой вертикали первой, обходится с помощью команды условного перехода, если условие  $x \geq 0$  верно (ge, строка 5), а в конце, после собственно кода ветви, вставлена команда безусловного перехода к первой команде, которая будет выполнена независимо от условия, чтобы после ветви «нет» выполнение естественным путём не перешло к ветви «да» (переход к метке `end_check:`, строка 7). Сразу после этого начинается код ветви «нет» (метка `setting_1`, строка 8).

Описанный способ наиболее универсален и позволяет реализовать условный оператор со сколь угодно объёмным кодом ветвей. Необходимый код ветвей «нет» и «да» заместит соответственно строки 6 и 9.

### 6.3.3. Цикл

Цикл — единственная алгоритмическая конструкция высокого уровня, позволяющая передать управление назад. Соответственно, цикл на ассемблере можно реализовать только с помощью команд передачи управления.

Пусть необходимо найти сумму двоичных цифр беззнакового числа  $x$ .

Построим схему алгоритма для решения этой задачи, не используя парный блок цикла и выстроив блоки линейно вдоль одной вертикали (рис. 6.9, а), реализован цикл с предусловием).

При соединении блоков ещё немного отступим от требований ЕСПД и отделим точку, где боковой поток случая  $CF \neq 0$  входит в основную вертикаль, от точки, где поток отходит от вертикали, чтобы перейти назад к началу итера-

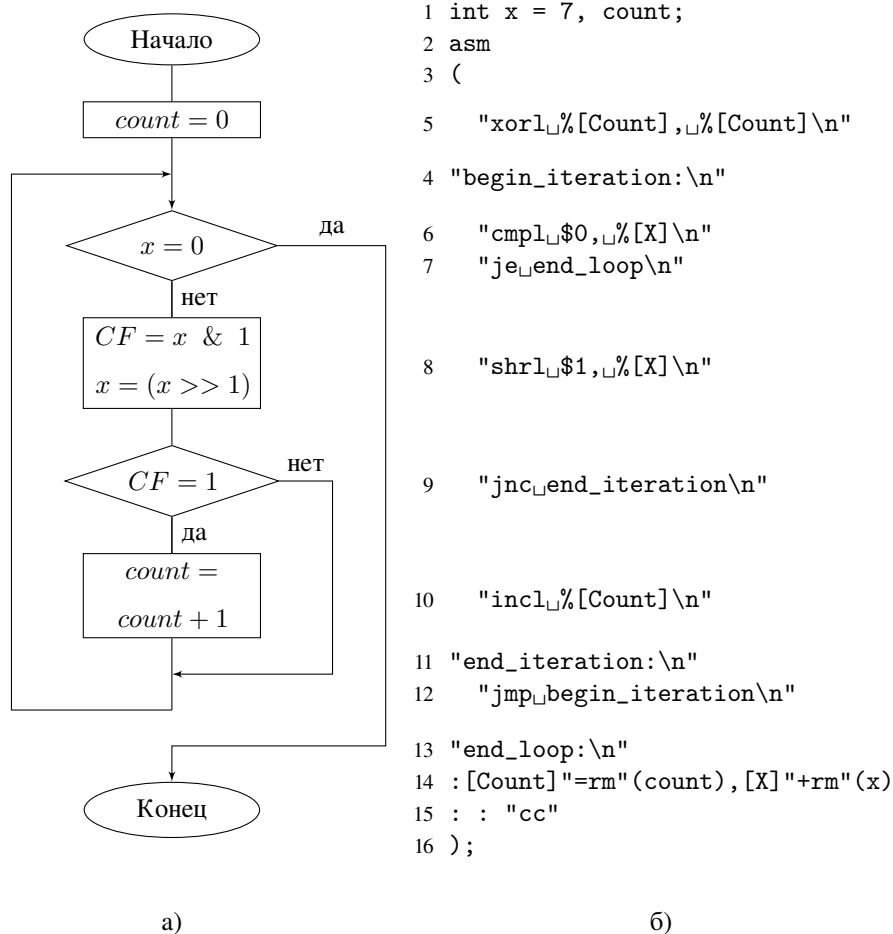


Рис. 6.9. Алгоритм и реализация цикла

ции. Расположение этих точек выберем так, чтобы поток основной вертикали на каждом её участке шёл в естественном направлении — сверху вниз (это не только позволяет не рисовать стрелку для обозначения направления, но и соответствует выполнению кода, не включающего команд передачи управления).

Теперь сопоставим каждому блоку одну или несколько команд ассемблера. Отход потока от базовой вертикали будет соответствовать команде перехода, точки входа боковых потоков в основной вертикальный — меткам (рис. 6.9, б).

Цикл с постусловием будет отличаться только на этапе построения схемы алгоритма.

Цикл с параметром можно реализовать либо как цикл с предусловием, как и в C/C++, либо как цикл с постусловием.

При этом, если счётчик цикла не является одновременно индексом массива в памяти, имеет смысл инициализировать счётчик максимальным значением  $i = i_{max}$  и уменьшать его на каждом шаге  $i = i - 1$ . Это позволит избавиться от команды сравнения, так как команды декремента и вычитания не только изменяют операнд  $i$ , но и выставляют флаги. При переходе от ненулевого значения к 0 получим  $ZF = 1$ , при переходе от неотрицательного значения, в частности нуля, к отрицательному —  $CF = 1$  и  $SF \neq OF$ .

### 6.3.4. Массивы

Массив — структура данных в виде набора однородных компонент (элементов массива), расположенных в памяти непосредственно друг за другом (независимо от настроек выравнивания). Элемент с наименьшим индексом располагается по младшему адресу. Доступ к отдельным элементам массива осуществляется с помощью индексации, то есть через ссылку на массив с указанием номера (индекса) нужного элемента.

В языке C++ наименьший индекс массива всегда равен 0, а имя массива является константным указателем на его начало. Таким образом, адрес  $i$ -го элемента массива  $M$  равен  $M + i \cdot size$ , где  $size$  — размер одного элемента и, в случае, когда длина массива не равна нулю, может быть определён как `sizeof(M[0])`. Для большинства простых типов (`char`, `bool`, `short`, `int`, `long`, `long long`, `float`, `double`, `size_t`, `ptrdiff_t`, `void*`) этот размер как на 32-, так и на 64-битной платформе равен 1, 2, 4 или 8.

Соответственно, в ассемблере для получения элемента массива (если элементы массива имеют размер 1, 2, 4 или 8) будут использованы три из четырёх компонент эффективного адреса `displacement(base, index, multiplier)` — база (адрес начала массива  $M$ ), индекс и масштаб (размер элемента).

В частности, адрес  $i$ -го элемента массива  $M$  из чисел типа `int` на 32-разрядных и многих 64-разрядных платформах равен  $M + i \cdot 4$ , и элемент будет

записан как  $(M, i, 4)$ . При этом база  $M$  и индекс  $i$  должны быть 32-разрядны (на соответствующей платформе) и располагаться в регистрах.

**Листинг 6.29.** Инициализация четырёхбайтового элемента массива  $M[i]$

```

1 const int N = 8;
2 int M[N], i = 0;
3 asm
4 (
5 "movl_$0,_(%[M],_[I],_4)\n"
6 : [I] "+r"(i)
7 : [M] "r"(M)
8 : "memory"
9 );

```

Так как мы модифицируем во вставке элементы массива (а не сами параметры  $[I]$  и  $[M]$ ), необходимо указать в списке перезаписываемых элементов специальное значение "memory".

Для инициализации массива в целом необходим цикл. В листинге 6.30 показана инициализация нулями массива из элементов типа *int* (на большинстве платформ он четырёхбайтовый) циклом с предусловием, аналогичным циклу `for` языка C++.

**Листинг 6.30.** Инициализация массива  $M$  из четырёхбайтовых элементов

```

1 const int N = 8;
2 int M[N], i;
3 asm
4 (
5 "xorl_%,_[I],_[I]\n"
6 "begin_iteration:\n"
7 "cmpl_[M_len],_[I]\n"
8 "jge_end_loop\n"
9 "movl_$0,_(%[M],_[I],_4)\n"
10 "incl_[I]\n"
11 "jmp_begin_iteration\n"
12 "end_loop:\n"
13 : [I] "=&r"(i)
14 : [M_len] "i"(N), [M] "r"(M)

```

```

15 : "cc", "memory"
16 );
17
18 for(i = 0; i < N; ++i)
19 {
20     cout << M[i] << "_";
21 }
22 cout << endl;

```

Действительно, при выводе массива M мы увидим нули:

```
0 0 0 0 0 0 0 0
```

Если в массиве гарантированно есть хотя бы один элемент, можно сократить код, используя постусловие (листинг 6.31).

### Листинг 6.31. Инициализация непустого массива M

```

1 asm
2 (
3 "xorl_ %[I], %[I]\n"
4 "begin_iteration:\n"
5 "movl_ $0, _(%[M], %[I], 4)\n"
6 "incl_ %[I]\n"
7 "cmpl_ %[M_len], %[I]\n"
8 "jnge_ begin_iteration\n"
9 : [I] "=&r"(i)
10 : [M_len] "i"(N), [M] "r"(M)
11 : "cc", "memory"
12 );

```

В приведённом выше коде на тип `int` у элементов массива указывают как суффикс `l` у команды `mov`, так и масштаб `4` при вычислении адреса. Обе характеристики важны: попытка опустить суффикс команды приведёт к ошибке, так как ни один из операндов команды `mov` здесь не является регистром и, следовательно, не имеет определённого размера.

Также суффикс и вычисление адреса должны соответствовать друг другу: хотя команды `movw $0, (%[M], %[I], 4)` и `movl $0, (%[M], %[I], 2)` синтаксически корректны и не вызовут ошибок компиляции, обе они при обработке



массива из элементов типа `int` некорректны по смыслу. Команда `movw $0, (%[M], %[I], 4)` запишет по адресу  $M[i]$  16-битный ноль, который инициализирует только младшие два байта из четырёх; таким образом, значение элемента  $M[i]$  останется неопределённым. Команда `movl $0, (%[M], %[I], 2)` перезапишет не  $M[i]$ , а либо элемент  $M[i/2]$  (для чётного  $i$ ), либо два старших байта одного элемента и два младших следующего (для нечётного  $i$ ).

Избавиться от явного указания суффикса и масштаба можно, используя модификаторы параметров:

**Листинг 6.32.** Инициализация элемента массива  $M[i]$  размера `el_size`

```

1 const int N = 8;
2 short M[N];
3 int i = 3;
4 asm
5 (
6 "mov%z[el_type]_,$0,_(%[M],_[I],_#[el_size])\n"
7 : [I] "+r"(i)
8 : [M] "r"(M), [el_size] "i"(sizeof(M[0])), [el_type] "m"(M[0])
9 : "memory"
10 );
```

Такой код будет компилироваться и выполняться корректно для любого типа элементов массива  $M$ , причём для типа `short` выбирается тот из синонимичных суффиксов, который не вызовет неоднозначности с командой `movs`. К сожалению, это потребовало введения двух новых входных параметров: константа `[el_size]` для масштаба `sizeof(M[0])` и `[el_type]`, равный  $M[0]$ , для определения суффикса размера (так как ни один из ранее использованных параметров — ни указатель  $M$ , ни индекс  $i$  — в общем случае не совпадает по размеру с элементом массива).

В листинге 6.33 показана инициализация нулями массива из элементов целого типа.

**Листинг 6.33.** Инициализация массива  $M$  из элементов 2-8 байт

```

1 const int N = 8;
2 short M[N];
3 int i;
4 asm
5 (
6 "xorl_%[I],_%[I]\n"
7 "begin_iteration:\n"
8 "mov%z[el_type],_$0,_(%[M],_%[I],%c[el_size])\n"
9 "incl_%[I]\n"
10 "cmpl_%[M_len],_%[I]\n"
11 "jnge_begin_iteration\n"
12 : [I] "=&r"(i)
13 : [M_len] "i"(N), [M] "r"(M),
14 [el_size] "i"(sizeof(M[0])),
15 [el_type] "m"(M[0])
16 : "cc", "memory"
17 );
18 for(i = 0; i < N; ++i)
19 {
20     cout << M[i] << "_";
21 }
22 cout << endl;

```

Тип *short* в листинге 6.33 может быть заменён на *long* или *long long*. Не вызовет ошибки компиляции и замена на однобайтовое целое *char*, но вывод будет выглядеть пустым. При выводе на экран числа типа *char* стандартными средствами C++ отображается символ ASCII, код которого равен числу. Нулевой символ отображается как конец строки, то есть никак. Если заменить в команде `mov` константу `$0` на номер печатного символа ASCII, в частности, на `$'y'`, вывод на экран покажет  $N$  заданных символов.

Масштаб, равный 1, может быть опущен. Таким образом, если размер элемента равен одному байту, то адрес  $i$ -го элемента массива  $M$  равен  $M + i$  и сам элемент может быть записан и как  $(M, i, 1)$ , и как  $(M, i)$ .

Рассмотрим инициализацию массива кодами строчных латинских букв, начиная с 'a' (листинг 6.34).

**Листинг 6.34.** Инициализация массива строчными латинскими буквами

```

1 const int N = 8;
2 int i, x;
3 char M[N];
4 asm
5 (
6     "movb_$'a',_b[X]\n"
7     "xorl_%[I],_%[I]\n"
8     "begin_iteration:\n"
9     "cmpl_%[M_len],_%[I]\n"
10    "jge_end_loop\n"
11    "movb_%b[X],_(%[M],_%[I])\n"
12    "incb_%b[X]\n"
13    "incl_%[I]\n"
14    "jmp_begin_iteration\n"
15    "end_loop:\n"
16    : [I] "&r"(i), [X] "&q"(x)
17    : [M_len] "i"(N), [M] "r"(M)
18    : "cc", "memory"
19 );

```

Вывод инициализированного таким образом массива приведёт к следующему результату:

```
a b c d e f g h
```

Если размер *size* элемента отличен от 1, 2, 4 или 8, он не может быть масштабом при вычислении адреса; таким образом, смещение *i*-го элемента относительно начала массива

$$offset = i \cdot size$$

необходимо вычислить отдельно и затем получить элемент как `(M, offset)`. Так как элементы массива, как правило, обрабатываются в цикле, это можно сделать последовательным сложением с *size* на каждой итерации.

Один из возможных вариантов инициализации массива из элементов типа *long double* (в GCC число *long double* имеет размер 10 байт, а выделяемая под него память может занимать как 12, так и 16 байт) показан в листинге 6.35.

**Листинг 6.35.** Инициализация массива 80-битных вещественных чисел

```

1 const int N = 8;
2 int i;
3 long double M[N], *p;
4 asm
5 (
6 "movl_%[M_len],_%[rev_idx]\n"
7 "movl_%[M],_[][%[el_addr]]\n"
8 "begin_iteration:\n"
9 "fldpi\n"
10 "fstpt_[][%[el_addr]]\n"
11 "addl_%[el_size],_%[el_addr]\n"
12 "decl_%[rev_idx]\n"
13 "jnz_begin_iteration\n"
14 : [rev_idx]="&r"(i), [el_addr]="&r"(p)
15 : [M_len]"i"(N), [M]"r"(M), [el_size]"i"(sizeof(M[0]))
16 : "cc","memory"
17 );

```

Каждый элемент получает значение  $\pi$ .

Для инициализации массива нулями можно воспользоваться тем, что нулевое вещественное значение состоит из одних нулей (листинг 6.36).

**Листинг 6.36.** Инициализация массива вещественных чисел нулями

```

1 asm
2 (
3 "xorl_%[I],_%[I]\n"
4 "begin_iteration:\n"
5 "movl_$0,_%[M],_%[I],4)\n"
6 "incl_%[I]\n"
7 "cmpl_%[M_len],_%[I]\n"
8 "jnge_begin_iteration\n"
9 : [I]="&r"(i)
10 : [M_len]"i"(N*sizeof(M[0])/4), [M]"r"(M)
11 : "cc","memory"
12 );

```

Память, отведённая под массив  $M$  ( $N * \text{sizeof}(M[0])$  байтов) целиком заполняется четырёхбайтовыми нулевыми блоками. В результате каждый эле-

мент  $M[i]$ ,  $i \in [0, N - 1]$  получает нулевое значение, что можно увидеть, выведя  $M$  на экран поэлементно.

## Многомерные массивы

Если массивы с одним индексом естественно отображаются на одномерное адресное пространство памяти, то о расположении элементов двумерного массива необходимо условиться дополнительно.

В большинстве ЯВУ элементы статических многомерных массивов располагаются в памяти так, что при движении от начала массива по возрастанию адресов быстрее всего меняется последний индекс. После того, как последний индекс достигнет максимального значения, увеличивается предпоследний и так далее.

В частности, статические двумерные массивы (матрицы) развёрнуты в одномерный по строкам — сначала идёт вся нулевая строка, затем вся первая и так далее, то есть в массиве

```
1 const int I = 8, J = 8;  
2 int M[I][J];
```

адрес элемента  $M[i][j]$  равен  $M + (i \cdot J + j) \cdot size$ , где  $size = \text{sizeof}(M[0][0])$ .

При обработке всех элементов матрицы можно рассматривать её как одномерный массив длины  $I \cdot J$ , так как все её элементы однородны и расположены в памяти непосредственно друг за другом. В этом случае индекс элемента  $M[i][j]$  в этом массиве  $index = i \cdot J + j$ .

Обратное преобразование (расщепление при необходимости эффективного индекса на номера строки и столбца) выглядит следующим образом:

$$\begin{cases} i = index / J \\ j = index \bmod J \end{cases}$$

и может быть выполнено одной командой беззнакового деления.

Если требуется выполнить одно и то же действие, в частности, инициализацию, над всеми элементами матрицы, достаточно выполнить один проход, как по массиву  $M[I \cdot J]$ .

### Динамические массивы

Динамические массивы, память под которые выделяется из кучи с помощью оператора *new[]* или функций *\*alloc()* и освобождается *delete[]/free()*, могут быть только одномерными. При необходимости размещения в куче многомерного массива программист вручную либо разворачивает его в длинный одномерный, либо размещает в древоподобной структуре данных из нескольких небольших одномерных массивов. Способ обращения к элементу в таком случае зависит от способа организации данных.

Обращение к одномерному динамическому массиву, после того, как адрес его начала помещён в регистр, ничем не отличается от обращения к одномерному статическому.

### 6.3.5. Структуры и объекты

#### Выравнивание данных

Хотя оперативная память — устройство с произвольным доступом, то есть возможно читать значения по любым адресам в любом порядке, время доступа различается в зависимости от расположения данных. Конкретные особенности временных характеристик обращения к оперативной памяти зависят от особенностей процессора и чипсета.

Тем не менее, есть несколько общих правил, позволяющих не потерять в производительности слишком сильно.

1. Выравнивание. Фактически процессор не работает с данными, взятыми напрямую из оперативной памяти. При чтении данные поступают в сверхоперативную память (кеш); изменения вначале фиксируются в кеше, затем попадают в оперативную память. Обмен между памятью и кешем произ-

водится пакетами, длина которых составляет от 32 до 128 байт. Начало пакета кратно его длине.

Таким образом, если элемент попадает на границу таких блоков-пакетов, для его загрузки потребуется два запроса к памяти [35].

Чтобы избежать таких ситуаций, достаточно (хотя и не всегда необходимо), чтобы граница между элементами в памяти была кратна определённому числу (таблица 6.3) — выравнена.

### Размер выравнивания для данных различных типов

Таблица 6.3

Размер данных	Граница
1 байт (8 бит)	Произвольная
2 байта (16 бит)	Кратная 2 байтам
4 байта (32 бита)	Кратная 4 байтам
8 байт (64 бита)	Кратная 8 байтам
10 байт (80 бит)	Кратная 16 байтам
16 байт (128 бит)	Кратная 16 байтам

По умолчанию в C++ как размер простых типов, так и величина, которой кратен адрес начала такой переменной — величина выравнивания (кроме *long double*) соответствуют этим значениям.

Десятибайтовый тип *long double* может иметь размер (*sizeof*) как 16, так и 12 байт; в последнем случае он выравнивается на 4 байта (а компиляторы из коллекции Microsoft Visual Studio полагают *long double* = *double*, таким образом и размер, и величина выравнивания там равны 8).

- Обход последовательно расположенных элементов в порядке возрастания адресов выполняется быстрее, чем в обратном.

## Поля структур

Структуры и объекты в C++ сочетают в себе несколько в общем случае разнородных компонент (полей), расположенных в определённом порядке. Доступ к отдельным полям структуры на ЯВУ осуществляется по имени. При обработке на языке ассемблера придётся использовать смещение поля относительно начала структуры, которое будет зависеть не только от состава структуры, но и от компилятора и его настроек.

Обычно поля следуют в порядке объявления и начало поля кратно некоторой величине, значение которой для конкретного поля/типа, а также текущей версии и настроек в GCC можно получить с помощью оператора `__alignof__` (синтаксис аналогичен `sizeof`). При этом между началом одного поля и концом предыдущего может образоваться промежуток, также промежуток может образоваться после последнего элемента структуры. Соответственно, размер структуры может быть больше суммы размеров её полей; также размер структуры может меняться от перестановки полей между собой.

Изменить максимальную кратность выравнивания (часто называемую просто выравниванием) в GCC можно с помощью флага компиляции `-fpack-struct[=n]`. Также GCC для совместимости с компиляторами Microsoft поддерживает набор директив препроцессора `#pragma pack`, позволяющих задать различную кратность выравнивания для различных определений типов:

- `#pragma pack(n)` просто устанавливает новое значение выравнивания;
- `#pragma pack()` возвращает выравнивание по умолчанию (возможно, заданное `-fpack-struct[=n]`);
- `#pragma pack(push[,n])` сохраняет текущее выравнивание во внутреннем стеке и, при заданном  $n$ , устанавливает новое значение
- `#pragma pack(pop)` восстанавливает выравнивание из вершины внутреннего стека (и удаляет эту запись оттуда).



Рассмотрим расположение полей и размер структур при различных настройках выравнивания (листинг 6.37).

### Листинг 6.37. Структуры при различных значениях выравнивания

```

1 const int N = 10;
2 struct TSomeStruct
3 {
4     char Tag;
5     int  Val;
6 }
7 s1, a1[N];
8
9 #pragma pack (push, 1)
10 struct TSqueezedStruct
11 {
12     char Tag;
13     int  Val;
14 }
15 s2, a2[N];
16
17 #pragma pack (pop)
18 struct TAnotherStruct
19 {
20     char Tag;
21     int  Val;
22 }
23 s3, a3[N];
24
25 TSqueezedStruct s20, a20[N];
26
27 #define PRINT(I) cout << reinterpret_cast<char *>(&s##I.Tag) \
28     - reinterpret_cast<char *>(&s##I) << " " \
29 << reinterpret_cast<char *>(&s##I.Val) \
30     - reinterpret_cast<char *>(&s##I) \
31 << " " << sizeof(s##I) << " " << sizeof(a##I) << endl;
32
33 PRINT(1)           // 0 4 8 80
34 PRINT(2)           // 0 1 5 50
35 PRINT(3)           // 0 4 8 80
36 PRINT(20)          // 0 1 5 50

```

Первое поле *Tag* всегда имеет нулевое смещение. Второе поле *Val* для структур *s1* и *s3* (выравнивание по умолчанию) смещено на четыре байта, так как адрес переменной типа *int* для наилучшей производительности должен быть кратен 4. В структурах *s2* и *s20* — при максимальной кратности выравнивания 1 —

поле *Val* следует непосредственно за *Tag*. Следует отметить, что настройки выравнивания задаются при определении типа *TSqueezedStruct*, а не конкретных переменных *s2* и *s20*.

Следует также отметить, что в массиве элементы всегда следуют друг за другом без промежутков.

На практике не рекомендуется изменять настройки выравнивания (особенно директивой `#pragma pack`, приводящей к несовместимости одинаково описанных структур), так как это может замедлить работу программы или даже нарушить её работоспособность.

При необходимости записи структуры в файл для избавления от дыр неопределённого размера лучше воспользоваться покомпонентной записью.

Для придания размеру дыр определённости необходимо по возможности описывать поля структуры в таком порядке (и, возможно, добавить ещё несколько неиспользуемых полей), чтобы границы между полями независимо от настроек выравнивания совпадали с рекомендуемыми значениями таблицы 6.3.

### Обращение к полю структуры

Адрес поля структуры равен сумме адреса структуры *base* и смещения нужного поля *displacement*. Для обращения к полю структуры необходимо разыменовать его адрес  $*(base + displacement)$ .

Это соответствует косвенной адресации с двумя компонентами — базой *base* и смещением *displacement*, что в GAS обозначается `displacement(base)`.

В листинге 6.38 показана инициализация полей *Tag* и *Val* структуры *TSomeStruct*, описанной в листинге 6.37.

### Листинг 6.38. Инициализация структуры с выравниванием на 4 байта

```
1 TSomeStruct s;  
2 asm  
3 (  
4 "movb_$'a',_4(%[S])\n"  
5 "movl_$13,_4(%[S])\n"
```

```

6 :
7 : [S] "r" (&s)
8 : "memory"
9 );
10 cout << s.Tag << " " << s.Val << endl;

```

Вывод программы показывает корректность инициализации

a 13

Так как расположение полей зависит от настроек компиляции, более надёжно передавать смещения полей как параметры вставки (листинг 6.39).

### Листинг 6.39. Инициализация структуры с неизвестным выравниванием

```

1 asm
2 (
3 "movb$_'a',$_%c[tag_disp] (%[S])\n"
4 "movl$_13,$_%c[val_disp] (%[S])\n"
5 :
6 : [S] "r" (&s),
7   [tag_disp] "i" (reinterpret_cast<char *>(&s.Tag)
8     - reinterpret_cast<char *>(&s)),
9   [val_disp] "i" (reinterpret_cast<char *>(&s.Val)
10     - reinterpret_cast<char *>(&s))
11 : "memory"
12 );

```

Полученный код не зависит от настроек выравнивания, но очень тяжело читается.

Если код оформляется не как ассемблерная вставка, а как функция, принимающая структуру, описанную на C++, необходимо либо передавать смещения полей как параметры функции и рассчитывать адреса вручную (что сильно замедлит выполнение), либо задаться конкретными значениями (возможно, при помощи макросов препроцессора), либо, что лучше всего, описывать структуру таким образом, чтобы при любых настройках выравнивания дыры между полями отсутствовали и смещения полей не менялись.

### Листинг 6.40. Структура, выравненная вручную

```
1 struct TRobustStruct
2 {
3     char Tag;
4     char dummy[3]; // неиспользуемые поля
5     int Val;
6 };
```

Листинг 6.40 показывает подобное описание.

## Контрольные вопросы

1. Какая команда передаёт управление подпрограмме?
2. Какая команда возвращает управление вызывающей программе?
3. Что такое адрес возврата?
4. Какие вы знаете соглашения о вызове?
5. Как импортировать ассемблерную функцию в проект на C++?
6. Как, согласно ЕСПД, изображается блок «терминатор»?
7. Как, согласно ЕСПД, изображается блок «процесс»?
8. Как, согласно ЕСПД, изображается блок «решение»?
9. Как располагаются в памяти элементы массива?
10. Как найти размер массива, зная размер элемента и их количество?
11. Что такое выравнивание полей структуры?
12. Зачем нужно выравнивание данных?

## Глава 7

# Программирование на языке высокого уровня: C++

Два программиста быстро находят общий язык.

Как правило, это C++.

*Программистский фольклор*

Для разработки программного обеспечения часто используется язык общего назначения C, а также разработанный на его основе объектно-ориентированный язык C++.

Программированию на C++ посвящён отдельный курс. Вспомним некоторые особенности этого языка, полезные при исследовании содержимого оперативной памяти или сочетании программы на C++ с ассемблерными функциями или вставками.

Синтаксис языков C и C++ стандартизирован. В настоящее время существует четыре стандарта языка C:

- **C89 (ANSI C)** ANSI X3.159-1989;
- **C90** ISO/IEC 9899:1990;
- **C99** ISO/IEC 9899:1999, последняя правка от 2007-11-15;
- **C11** ISO/IEC 9899:2011 от 2011-12-19.

Доступ к текстам стандартов платный. Последний бесплатно доступный черновик C11 — **n1570** от 2011-04-12.

Также существует четыре стандарта C++. Последний, C++17, **опубликован в декабре 2017 г. Начато обсуждение пятой версии — C++20.**

- **C++98** ISO/IEC 14882:1998;
- **C++11** ISO/IEC 14882:2011;
- **C++14** ISO/IEC 14882:2014;

- **C++17 ISO/IEC 14882:2017** (последний бесплатно доступный черновик — **n4659** от 2017-03-21).

Не все компиляторы поддерживают последние стандарты в полном объёме. В данном пособии рассматриваются основные возможности языка, реализованные для всех компиляторов.

## 7.1. Структура программы

Но вот Эм шагает в область сильного слова «Могу».

Слушайте, слушайте моговест мощи!

*В. Хлебников. Зангези*

Выполнение программы на C или C++ начинается с функции *main()*. Она называется головной, стартовой или главной функцией программы.

Функция *main()* описана в разделе [**basic.start.main**] стандарта C++. Она должна быть определена как *int main()* или как *int main(int, char \*\*)*. Обычно используются обозначения *int main(int argc, char \* argv[])*.

Функцию *main()* иногда называют точкой входа в программу, но это не совсем так. С точки зрения компоновщика, точка входа — это метка *\_start*. Для программ на C/C++ по адресу *\_start* находится стартовый код библиотеки *libc*, который, в частности, инициализирует все используемые библиотекой ресурсы и глобальные объекты, готовит параметры для стартовой функции *main()*, вызывает её, а после возврата управления из *main()* завершает программу.

При вызове исполняемого файла программы ему часто передаются так называемые параметры командной строки. Обычно это имя обрабатываемого файла (например, `pdflatex paper.tex`) или настройки для работы (`ls -la`). В частности, щелчок мыши в графической оболочке по файлу с данными эквивалентен запуску какой-либо программы с именем щёлкнутого файла в качестве параметра.

Параметры командной строки разделяются пробелами; например, здесь

```
grep str */*.tex *.tex
```

программа `grep` (поиск в текстовых файлах) вызывается с тремя параметрами — `str` (искомая строка), `*/*.tex` и `*.tex` (имена файлов для поиска). Нулевым параметром командной строки считается само имя исполняемого файла.

Общее количество параметров, включая имя исполняемого файла — это *argc*, первый аргумент функции *main()*. Массив строк *argv* содержит сами эти параметры.

Возвращаемое значение функции *main()* — код завершения программы. Он равен нулю в случае успешного завершения и ненулевому коду ошибки в противном случае.

Корректный код завершения позволяет оболочке учитывать итоги выполнения программы. В частности, следующий однострочный скрипт оболочки

```
pdflatex paper && bibtex paper
```

запускает программу `bibtex` (сборка библиографии) только в том случае, если программа `pdflatex` (сборка текста) корректно завершилась.

## 7.2. Типы данных

На данные свои взирая объективно,

Задумал типы я и идеал создал...

*К. П. Прутков. Безвыходное положение*

Базовые типы C++ описаны в разделе `Fundamental types` ([**basic.fundamental**]) стандарта C++ [25]. Раздел `Types` ([**basic.types**]) описывает общие характеристики хранения данных в памяти.

### 7.2.1. Целые типы

Существует пять стандартных **знаковых целых типов**:

- `signed char`,
- `signed short int` (синонимы: `short`, `signed short`),
- `signed int` (синонимы: `int`, `signed`),
- `signed long int` (синонимы: `long`, `signed long`),
- `signed long long int` (синонимы: `long long`, `signed long long`)

и пять соответствующих **беззнаковых целых типов** (каждый из них имеет тот же размер и те же требования к выравниванию, что и соответствующий знаковый):

- `unsigned char`,
- `unsigned short int` (синонимы: `unsigned short`),
- `unsigned int` (синонимы: `unsigned`),
- `unsigned long int` (синонимы: `unsigned long`),
- `unsigned long long int` (синонимы: `unsigned long long`).

Тип `char`, в зависимости от реализации, может быть знаковым или беззнаковым. Типы `char`, `signed char` и `unsigned char` имеют один размер и одинаковые требования к выравниванию.

Стандарт C++ не содержит явных значений разрядности типов.

Согласно стандарту, `char`, `signed char` и `unsigned char` занимают 1 байт. При этом, если байт на используемой программно/аппаратной платформе не восьмибитен, то и `char` *однобайтовый*, но не *однооктетный*, то есть занимает не 8 бит.

Таким образом, всякий объект любого типа (обозначим его `T`) может быть скопирован в массив `char [sizeof(T)]`.



В ряду целых типов каждый следующий тип имеет размер (и диапазон значений) не меньше предыдущего:

$$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \leq \text{sizeof(long long)}$$

Размеры стандартных целых типов C++ также должны соответствовать ограничениям раздела 5.2.4.2.1 стандарта C [20]. В этом разделе описаны значения, которые обязательно должны быть включены в диапазоны значений соответствующих типов (при этом указанные значения не обязательно должны быть граничными). Анализ этих значений приводит к следующим выводам о минимально допустимой разрядности стандартных типов (таблица 7.1).

### Минимальная разрядность стандартных целых типов

Таблица 7.1

Тип	Разрядность, бит (не менее)
char	8
short	16
int	16
long	32
long long	64

Тип `int` должен соответствовать «естественной» разрядности архитектуры (расплывчатость этой формулировки и то, что по умолчанию разрядность данных в шестидесятичетырехбитном режиме равна 32, приводит к тому, что практически на 64-битной платформе тип `int` чаще всего 32-разряден).

Типы `char`, `signed char` и `unsigned char` называются ещё **узкими (narrow) символьными типами**, так как они могут быть интерпретированы не только как числа, но и как символы; соответственно их размер должен быть таким, чтобы хранить представление любого символа из **базового набора**.

Единственное отличие узких символьных типов от других целых (кроме размера) — то, что операторы ввода/вывода в поток для них перегружены так, что отображают не значение переменной, а символ, код которого равен этому значению. Отображение чисел, не равных кодам символов ASCII, не определено и может быть разным для различных реализаций,

Все арифметические операции для `char`, `signed char` и `unsigned char` выполняются точно так же, как и для любого другого целого типа.

Для типа `unsigned char` каждой возможной комбинации разрядов должно соответствовать отдельное число. Для других типов это не обязательно.

Для представления **расширенного набора символов** введён специальный тип `wchar_t`, имеющий такой же размер, знаковость и требования к выравниванию, что и один из целых типов.

Тип `bool` может хранить только два значения — `true` и `false`.

### Практическая реализация

Всё, что явно не прописано в стандарте C++, может быть реализовано по-разному на различных платформах.

Всё, написанное в этом разделе и в аналогичных разделах ниже, описывает в основном платформу x86 и наиболее популярные компиляторы. На других программно/аппаратных платформах (в частности, при использовании экзотического компилятора) эти закономерности могут быть нарушены.

Для представления беззнаковых чисел используется натуральный двоичный код (см. раздел 2.4 настоящего пособия); знаковые представляются дополнительным кодом (см. разделы 2.4–2.5). Таким образом, для всех целых типов, в частности, для `unsigned char`, каждой возможной комбинации разрядов соответствует отдельное число.

Для знаковых и беззнаковых типов по-разному реализованы некоторые арифметические и битовые операции, в частности, умножение (оператор `*`),

деление (оператор получения частного / и оператор получения остатка от деления %), битовые сдвиги (операторы <<, >>), расширение при присваивании (если приёмник больше источника).

Беззнаковые целые типы должны подчиняться циклической арифметике по модулю  $2^N$ .

### **char, wchar\_t**

Тип `char` — восьмибитный и знаковый. При этом `char` и `signed char` не являются синонимами, хотя вычисления с их использованием компилируются в одинаковые конструкции. Именно, при перегрузке `f(char)` и `f(signed char)` считаются разными функциями и их имена декорируются по-разному (для сравнения, `f(int)` и `f(signed int)` не различаются и декорируются одинаково).

Базовым набором символов является ASCII. При интерпретации переменной типа `char` как символа значение этой переменной трактуется как ASCII-код. Таким образом, 64, 0x40 и '@' — это разные формы записи одного и того же числа (ASCII-код символа «собака» равен шестидесяти четырём).

Символы не из ASCII, в частности, русские буквы, в зависимости от реализации, могут быть представлены одной переменной типа `char` (кодировки koï8, cp1251, cp866; в этом случае с учётом знаковости `char` коды русских букв — 128–255 — трактуются как отрицательные, то есть 'ы' < 0 < 's') или цепочкой из нескольких `char`'ов (кодировка UTF-8; в этом случае можно описать строку из русских букв как `char []`, но невозможно описать одну такую букву как `char`).

Понятие расширенного набора символов и тип `wchar_t` возникли с появлением Unicode. Широкий символьный тип `wchar_t` может содержать любое количество байтов. Как правило, при хранении символьной информации в `wchar_t` используются такие юникодные кодировки или их части, в которых каждый логический символ занимает не менее одного `wchar_t` (UTF-32, или, если поддерживается только часть набора символов Unicode — UTF-16). Расширенный

набор символов, соответственно — та часть набора символов Unicode, которая поддерживается и может быть записана одним `wchar_t`.

Литералы, соответствующие широким строкам и символам, предваряются префиксом `L`. Для ввода/вывода широких символов и строк используются те же операторы, что и для узких, но другие потоки (`wcin/wcout`). Поток `cout` при выводе `wchar_t` отображает число, при выводе `wchar_t*` — адрес в шестнадцатеричном виде.

### **int, short, long, long long**

Тип `int` на 16-разрядных платформах занимал 16 бит, на 32-разрядных — 32 бита. На 64-битной платформе `int` чаще всего 32-разряден.

Тип `short` 16-разряден на 16-, 32- и 64-разрядных платформах.

Тип `long` на 16- и 32-разрядных платформах занимал 32 бита. На 64-битной платформе — 64.

Тип `long long`, если поддерживается, 64-разряден.

Таким образом в C++ не существует гарантированно 32-разрядного фундаментального целого типа.

Кроме фундаментальных типов, описанных в стандарте C++, поддерживаются также типы, описанные в стандарте C [20]. Современный стандарт языка C включает типы фиксированной разрядности, в частности, тридцатидвухразрядный `int32_t`.

### **7.2.2. Вещественные типы**

Существует три стандартных **вещественных типа**:

- `float`,
- `double`,
- `long double`.

тип `double` обеспечивает не меньшую точность, чем `float`, `long double` — не меньшую точность, чем `double`.

Множество значений типа `float` является подмножеством множества значений типа `double`; множество значений типа `double` является подмножеством множества значений типа `long double`:

$$\text{float} \subseteq \text{double} \subseteq \text{long double}$$

Все они хранят данные с плавающей запятой.

### Практическая реализация

Вещественные числа представляются в форматах с плавающей запятой различной точности, в соответствии со стандартом двоичной арифметики с плавающей точкой IEEE 754 [14, 80].

Процессоры семейства x86 имеют два набора команд для работы с вещественными числами с плавающей запятой. Это команды математического сопроцессора FPU и команды расширений XMM и YMM.

Модуль операций с плавающей запятой процессоров семейства x86 (floating point unit, FPU) поддерживает три типа вещественных чисел

- одинарной точности (32 бита),
- двойной точности (64 бита),
- с двойной расширенной точностью (80 бит, внутренний нестандартный формат FPU — 15 разрядов отводится под порядок, 64 под мантиссу).

Расширения XMM и YMM поддерживает два типа вещественных чисел — одинарной и двойной точности.

Типу `float` соответствует число одинарной точности, типу `double` — двойной.

Типу `long double` чаще всего соответствует 10-байтовое число расширенной точности. Размер выделяемой под переменную `long double` памяти при этом в зависимости от флагов компиляции (`-m96bit-long-double` и `-m128bit-long-double` в GCC), может быть равен 12 или 16 байт. Исполь-

зуются только первые 80 бит (10 байт), остальное — неиспользуемая память (заполнение).

В Microsoft Visual Studio типу `long double` соответствуют числа двойной точности (64 бита), хотя `long double` не считается синонимом `double`.

### 7.2.3. Специальные типы

Множество значений типа `void` пусто. Он, в частности, используется для описания функций, которые не возвращают значения, так как в C++ нет отдельного понятия процедуры или подпрограммы, не возвращающей значения.

Любое выражение может быть приведено к типу `void`.

Указатель `void*` считается нетипизированным. К типу `void*` может быть преобразован любой указатель.

### 7.2.4. Указатели

Каждому, простому или сложному, типу данных `T` в C++ соответствует свой тип указателя `T*` — адрес участка памяти, где лежит переменная типа `T` или массив таких переменных.

Все указатели — целые беззнаковые величины одного размера. По разрядности указателей в настоящее время определяется разрядность системы, так что на 32-битной платформе указатели занимают 32 бита, на 64-битной — 64.

Любой указатель может быть разыменован как массив, причём с использованием произвольного целого индекса. Таким образом, программист должен сам следить за количеством элементов по адресу, хранящемуся в переменной-указателе.

В C++ указатели на различные типы сами считаются различными типами. Соответственно, механизм типизации не позволяет неявно преобразовать указатели на различные типы друг в друга (в частности, `int*` в `char*`). Возможно только приведение к нетипизированному указателю `void*`. Это логично для строго типизированного языка высокого уровня, но не всегда удобно для низкоуровневого исследования данных.

Тем не менее, физически все указатели имеют одно строение — ячейка, где хранится адрес в памяти. Таким образом, указатели на разные типы могут быть преобразованы друг в друга с помощью оператора преобразования `reinterpret_cast` либо в два этапа через `void*`. Вначале исходный тип приводится с помощью `static_cast` к нетипизированному указателю, затем также с помощью `static_cast` нетипизированный указатель приводится к требуемому типу.

Также с указателями связано два целых типа той же разрядности. Это знаковая разность указателей — `ptrdiff_t` и беззнаковая длина массива — `size_t`.

В C/C++ нет фундаментального строкового типа, хотя есть строковые литералы. Функции стандартной библиотеки C обрабатывают как строки указатели на массивы чисел типа `char`.

### 7.3. Преобразование типов

И может быть, ещё ничего не сдвинулось:  
только ты сдвинулся.

*В. Кандинский.*

*Пощёчина общественному вкусу. Видеть*

В C++ есть четыре оператора явного преобразования (приведения) типов: `const_cast`, `static_cast`, `dynamic_cast` и `reinterpret_cast`. Кроме того, для совместимости поддерживается приведение в стиле C [72, 77].

**`const_cast`** убирает (или добавляет, но это редко используется) так называемые cv-спецификаторы (cv qualifiers), то есть `const` и `volatile`. Спецификатор `volatile` встречается редко, так что `const_cast` обычно применяется для снятия `const`. Если приведение типов не удалось, выдаётся ошибка на этапе компиляции. При использовании остальных приведений типов cv-спецификаторы останутся неизменными.

**Листинг 7.1.** Снятие спецификатора `const`

```

1 int i;
2 const int * pi = &i;
3 // *pi имеет тип const int,
4 // но pi указывает на int, который константным не является
5 int* j = const_cast<int *> (pi);

```

**static\_cast** преобразует выражения одного статического типа в объекты и значения другого статического типа. Может быть использован везде, где допустимо неявное преобразование типов (в частности, преобразования чисел вроде `int i = 1.3;` или указателя произвольного типа в нетипизированный `void *up = &i;`), а также для преобразования:

- любого типа к типу `void` (допустимое, но обычно ненужное на практике преобразование);
- указателя `void*` в указатель произвольного типа;
- базового класса к ссылке на производный класс (допустимо, но опасно, если объект на самом деле не того производного класса; в этих случаях часто лучше использовать `dynamic_cast`);
- указателя на базовый класс в указатель на производный класс (аналогично, надёжнее использовать `dynamic_cast`, если это возможно);
- интегральных типов в перечисляемые.

Если приведение типов не удалось, выдается ошибка на этапе компиляции.

**dynamic\_cast** — безопасное приведение по иерархии наследования, в том числе виртуального. Используется для преобразования

- указателя на базовый класс в указатель на производный класс:

```
1 dynamic_cast<deriv_class *>(base_class_ptr_expr)
```

если приведение невозможно, будет возвращён `NULL`;

- базового класса к ссылке на производный класс:

```
1 dynamic_cast<deriv_class &>(base_class_ref_expr)
```

если приведение невозможно, будет выброшено исключение `bad_cast`.



Для корректного преобразования классы должны быть полиморфными, то есть в базовом классе должна быть хотя бы одна виртуальная функция. Если это условие не соблюдено, ошибка возникнет на этапе компиляции. Если приведение невозможно, это станет ясно только на этапе выполнения программы.

**reinterpret\_cast** — приведение без проверок. Не может быть приведено одно значение к другому значению. Обычно используется, чтобы привести указатель к указателю, указатель к целому, целое к указателю. Умеет также работать со ссылками.

Возможные варианты использования

### Листинг 7.2. Варианты использования reinterpret\_cast

```
1 reinterpret_cast<whatever *>(some *)
2 reinterpret_cast<integer_expression>(some *)
3 reinterpret_cast<whatever *>(integer_expression)
```

Например, допустимо:

### Листинг 7.3. Корректное использование reinterpret\_cast

```
1 double x = 1;
2 int i = -1;
3 char *pc = reinterpret_cast<char *>(&x);
4
5 // refu -- ссылка на то же место в памяти, где расположена
   переменная i,
6 // но интерпретируется этот фрагмент памяти уже как unsigned
7 unsigned &refu = reinterpret_cast<unsigned &>(i);
8
9 // pu указывает на то же место в памяти, где расположена
   переменная i (аналогично)
10 unsigned *pu = reinterpret_cast<unsigned *>(&i);
11
12 // u -- новая переменная, инициализированная текущим значением i
   в беззнаковой интерпретации (0xFFFFFFFF)
13 unsigned u = reinterpret_cast<unsigned &>(i);
14
15 // lox ссылается на первые (в x86 -- младшие) 4 байта x и
   интерпретирует их как беззнаковое целое
16 unsigned &lox = reinterpret_cast<unsigned &>(x);
17 unsigned &hix = *(&lox + 4);
```

Но нельзя выполнить

```
1 int i;  
2 unsigned u = reinterpret_cast<unsigned>(i);
```

**Приведение в стиле C** (C-style cast) — самое медленное преобразование, так как последовательно перебираются следующие вызовы:

1. `const_cast`
2. `static_cast`
3. `static_cast + const_cast`
4. `reinterpret_cast`
5. `reinterpret_cast + const_cast`

Допустимо во всех случаях, но не рекомендуется из-за внешнего вида.

Примеры:

#### Листинг 7.4. Приведение в стиле C

```
1 double x = 1;  
2 int i = -1;  
3 char *pc = (char *)&x;  
4 unsigned &refu = (unsigned &)i;  
5 unsigned *pu = (unsigned *)&i;  
6 unsigned u = (unsigned)i;
```

## 7.4. Литералы C++

И оставлю кругом следы.

*В. Хлебников. Око*

Литералы (символические константы) — фиксированное значение в коде программы (3, 0xFF, 7.8). Литералы могут использоваться как для инициализации именованных констант и переменных (в этом случае тип литерала приводится к типу соответствующей константы или переменной), так и непосредственно в теле программы (магические числа).

### 7.4.1. Целые

Целочисленные литералы начинаются с цифры или знака (+ или −) и не содержат десятичной запятой и показателя степени.

Для тех чисел, которые соответствуют кодам ASCII для печатных и некоторых управляющих символов, есть альтернативная форма записи — соответствующий символ в одинарных кавычках, например `'\0'` равен 0, `'\t'` — 9, `'2'` — 50, `'R'` — 82, `'r'` — 114. Для чисел, соответствующих номерам Unicode, добавляется префикс `L`, так, `L'ы'` равен 1099, или `0x044B`.

#### Префиксы системы счисления

Целочисленные литералы могут предваряться префиксом, обозначающим систему счисления:

**0x, 0X** — шестнадцатеричная;

**0** (ведущий ноль) — восьмеричная;

**0b, 0B** — двоичная;

по умолчанию (без префикса) используется десятичная система. Так, одно и то же число может быть записано как 13, 015, 0xD и 0b1101.

Знак может быть поставлен только перед префиксом системы счисления, но не после него.

#### Суффиксы знаковости и размера

Целый литерал без суффикса имеет тип `int` (если значение выходит за пределы `int`, то используется минимальный знаковый тип, в который литерал помещается; ASCII-коды имеют тип `char`, Unicode-номера — `wchar_t`).

Для указания беззнакового типа литерала (без суффикса размера это тип `unsigned`) используется суффикс `u` или `U`.

Для указания размера используются следующие суффиксы:

**l, L** — `long`;

**ll, LL** — `long long`.

Так, `2ul` — беззнаковое число типа `unsigned long` (возможна также запись `2lu`).

Если значение литерала не помещается в тип, соответствующий суффиксу, выбирается более ёмкий.

К альтернативной символьной записи целых чисел суффиксы неприменимы.

### 7.4.2. Вещественные

Литералы с плавающей запятой задают значения, которые должны иметь дробную часть (возможно, нулевую). Эти значения содержат разделитель целой и дробной частей (в соответствии с западной традицией — точку,  $.$ ) и/или показатели степени:  $34.56$  ( $34 + \frac{56}{100}$ ),  $0.12$  ( $\frac{12}{100}$ ),  $1.$  (вещественное число 1),  $1e4$  ( $10^4$ ),  $5e-4$  ( $5 \cdot 10^{-4}$ ),  $2.12e+2$  ( $2,12 \cdot 10^2 = 212$ ),  $0x1p10$  ( $1 \cdot 2^{10} = 1024$ ),  $0xF.Fp4$  ( $F, F_{16} \cdot 2^4 = FF_{16} = 255$ ),  $0x.8p0$  ( $0,8_{16} = \frac{1}{2}$ ),  $0x.8p-1$  ( $0,8_{16} \cdot 2^{-1} = \frac{1}{4}$ ).

### Системы счисления

Вещественные литералы могут быть заданы в двух системах счисления — десятичной и шестнадцатерично-двоичной.

Литерал в десятичной системе может иметь вид

$$M[eS] = M[\cdot 10^S] \quad (7.1)$$

где  $M$  — значащие цифры, записанные в виде десятичного числа (возможно, со знаком и/или с дробной частью),  $S$  — десятичный порядок, записанный в виде целого десятичного числа. Порядок может быть опущен вместе с символом  $e$ . Регистр символа  $e$  может быть любым ( $e/E$ ).

Литерал в шестнадцатерично-двоичной системе предваряется шестнадцатеричным префиксом  $0x$  или  $0X$  и обязательно включает двоичный порядок

$$0xMpS = M \cdot 2^S \quad (7.2)$$

$M$  — значащие цифры, записанные в виде беззнакового шестнадцатеричного числа (возможно, с дробной частью),  $S$  — двоичный порядок, записанный в виде целого десятичного числа. Так как символ  $e/E$  является корректной шестнадцатеричной цифрой, порядок отделяется символом  $p/P$  (регистр может быть

любым). В шестнадцатерично-двоичном вещественном литерале должны присутствовать обе компоненты  $M$  и  $S$ , даже если  $M = 1$  или  $S = 0$ .

Знак может быть поставлен перед префиксом  $0x$ .

### Суффиксы размера

Вещественный литерал без суффикса имеет тип `double` (если значение выходит за пределы `double`, то — `long double`). Для указания размера используются следующие суффиксы:

**f, F** — `float`;

**l, L** — `long double`.

Если значение литерала не помещается в тип, соответствующий суффиксу, выбирается более ёмкий.

#### 7.4.3. Строки

Строковые литералы заключаются в двойные кавычки. Перед «широкими» строками ставится префикс `L`.

Русские строковые литералы могут быть как узкими, так и широкими. При этом представление полученной константы в памяти зависит от реализации. Если используются «узкие» строки, то количество символов в полученной строке, как правило, превышает количество букв в литерале за счёт того, что используется восьмибитный байт и кодировка UTF-8, так что одна русская буква представляется двумя элементами типа `char`.

Строковый литерал имеет тип `const char*` или `const wchar_t*` и является адресом соответствующего массива символов, завершающегося нулём — строки, расположенной в памяти.

Таким образом, узкая строка `"test"` занимает в памяти не четыре байта, а пять. Объявления

```
char s[] = { 't', 'e', 's', 't', '\0' };
```

и

```
char s[] = "test";
```

эквивалентны.

## 7.5. Средства автоматизации C++

Но если существует один кусок жизни числа, одна ветка,  
то существует и всё дерево чисел.

*В. Хлебников. Азбука Неба*

Языки C и C++ строго типизированы. Это позволяет избежать множества ошибок, но иногда требуется выполнить какое-нибудь одно действие над данными различных типов.

Для этого в языке C++ доступны два средства — шаблоны C++ и унаследованные от языка C макросы препроцессора.

Шаблоны раскрываются на этапе компиляции; при формировании нужной реализации учитываются фактические типы и значения параметров шаблона. Макросы — текстовая замена, производящаяся на этапе препроцессинга без учёта семантики текста.

### 7.5.1. Шаблоны C++

В языке C++ для выполнения единообразных действий над данными различных типов используются шаблоны. Существует два основных вида шаблонов — шаблонные функции и шаблонные классы.

Перед заголовком соответствующей функции или класса указывается дополнительный заголовок шаблона, который начинается с ключевого слова `template`, после которого в угловых скобках через запятую перечисляются параметры шаблона. Для каждого параметра указывается его тип и имя. Тип параметра шаблона может быть перечислимым (в частности, `int`, `char`, любое описанное пользователем перечисление) или описан ключевым словом `typename` или `class` (раздел **[temp.param]** стандарта утверждает, что семантика этих

ключевых слов не различается), тогда параметр представляет собой имя типа C++.

### Листинг 7.5. Шаблонная функция

```
1 template<typename T>
2 int foo(T x)
3 {
4     T y = x;
5     ...
6 }
```

Внутри шаблонной функции или шаблонного класса имя параметра может быть использовано как целочисленная именованная константа или как имя типа C++, в соответствии с типом параметра.

### Листинг 7.6. Шаблонный класс

```
1 template <typename T, int Rows, int Cols>
2 class TMatrix
3 {
4     T data[Rows][Cols];
5     ...
6 };
```

При вызове шаблонной функции достаточно просто передать ей набор параметров — `j = foo(1)`. Компилятор сформирует по шаблону нужную реализацию и подставит её вызов. При необходимости можно явно указать вызываемую реализацию, подставив после имени функции список фактических параметров шаблона — `foo<int>(1.7)`.

Можно явно описать реализацию шаблона для какого-либо конкретного типа (специализацию).

### Листинг 7.7. Частная реализация шаблона для типа long

```
1 template <typename T> T f(T x)
2 {
3     return x;
4 }
```

```
5 template <> long f(long x)
6 {
7     return 2*x;
8 }
```

В этом случае при вызове шаблонной функции  $f()$  с аргументом любого типа, кроме `long`, в том числе с аргументом типа `int`, она вернёт значение переданного аргумента  $x$ , а если вызвать  $f()$  с аргументом типа `long` — его удвоенное значение  $2x$ .

### 7.5.2. Макросы препроцессора C/C++

Для выполнения единообразных действий над величинами разных типов также могут использоваться макросы препроцессора, описанные в разделе 4.2.

Так как макрос является по сути текстовой заменой, описанные в нём действия могут быть применены к данным любого типа. Невозможно описать отдельную реализацию для параметров определённого типа, имени, длины или вида.

При этом в макросе, в отличие от шаблонной функции, возможно обработать не только значение или тип переменной, но и её имя. В частности, макрос `PRINT`, описанный в листинге 7.8, выводит на стандартный вывод имя переменной и её значение.

#### Листинг 7.8. Определение макроса отладочной печати переменной

```
1 #define PRINT(val) cout << #val << "_=" << val << "_";
```

Таким образом, строка `PRINT(x)PRINT(y)` будет преобразована препроцессором в:

```
1 cout << "x" << "_=" << x << "_";cout << "y" << "_=" << y
   << "_";
```

и, если переменные  $x$  и  $y$  существуют, выведет их значение. В частности, при  $x = 5,7$  и  $y = 0$  будет **напечатано** `x = 5.7 y = 0`.



## 7.6. Ввод-вывод

Человек ведёт переписку со всем земным шаром, а через  
печать сносится даже с отдалённым потомством.

*К. П. Прутков.*

*Плоды раздумья. Мысли и афоризмы*

В языке C++ доступно два варианта платформонезависимого ввода-вывода — полиморфные операторы ввода-вывода в поток и функции стандартной библиотеки C.

### 7.6.1. Ввод-вывод в поток

В библиотеке C++ описаны шаблонные классы потоков — объектов, куда может быть направлен ввод или откуда взят вывод. В качестве операторов ввода и вывода используются перегруженные операторы битового сдвига << и >>. Если левым операндом является поток, то оператор << выводит туда поток левый операнд и возвращает ссылку на этот поток. Аналогично перегружен оператор >>.

Используемые для вывода в поток перегруженные операторы << и >> реализованы для разных типов по-разному. Типы `char`, `signed char` и `unsigned char` отображаются в потоке как символы, код которых равен значениям переменных. Вывести значение такой переменной как число можно, преобразовав её в другой целый тип, в частности, `int` или `unsigned`, для чего можно использовать `static_cast`:

**Листинг 7.9.** Вывод первого байта  $x$  в поток как числа со знаком

```
1 char *p = reinterpret_cast<char *>(&x);  
2 cout << static_cast<int>(*p);
```

или

**Листинг 7.10.** Вывод первого байта  $x$  в поток как числа без знака

```
1 unsigned char *p = reinterpret_cast<unsigned char *>(&x);
```

```
2 cout << static_cast<unsigned>(*p);
```

оба эти листинга выводят в поток `cout` первый байт объекта `x`, первый — как знаковое число, второй — как беззнаковое.

Прочие целые типы отображаются как значение в десятичной, восьмеричной или шестнадцатеричной системе счисления. Используемую систему счисления можно изменить, используя манипуляторы `hex`, `oct`, `dec` или `setbase()`. Манипуляторы `hex`, `oct` и `dec` меняют вывод целых чисел (но не `char`) на шестнадцатеричный, восьмеричный и десятичный соответственно. Манипулятор `setbase (int base)` позволяет задать основание; фактически можно выбрать только 8, 10 и 16. Теми же манипуляторами можно задать систему счисления для ввода чисел.

Знак отображается только в десятичной форме вывода; восьмеричная и шестнадцатеричная формы при выводе знаковых чисел представляют собой двоичное представление числа, приведённое к соответствующей системе счисления. Так, число `-1` (так как в литерале не указан суффикс типа, число имеет тип `int`) в восьмеричной и шестнадцатеричной формах соответственно выглядит как `37777777777` и `ffffff`.

Регистр шестнадцатеричных цифр `A..F` (а также символа `E` в экспоненциальной форме вывода вещественных чисел) задаётся манипуляторами `uppercase` и `lowercase`.

Манипулятор `setfill(int ch)` устанавливает символ заполнения равным `ch`. В частности, `setfill('0')` указывает, что числа нужно дополнять до ширины, указанной манипулятором `setw()`, не пробелами, а ведущими нулями.

Действие манипуляторов `hex`, `oct`, `dec`, `setbase()`, `uppercase`, `lowercase` и `setfill()` не прекращается после вывода/ввода одного числа и длится до изменения другим аналогичным манипулятором.

Ширина вывода устанавливается манипулятором `setw(int w)` только для следующего выводимого значения. Если выводимое значение не помещается в `w` знакомест, оно выводится целиком.

Таким образом, строка

### Листинг 7.11. Вывод нескольких чисел в поток

```
1 cout << hex << setfill('0')
2   << setw(4) << 10 << " " << 20 << " "
3   << setw(3) << " " << setw(8) << 30 << " "
4   << setw(2) << 257 << endl;
```

поместит в поток `cout`

```
1 000a 14 00 0000001e 101
```

манипулятор `hex` действует на все целые числа; `setfill('0')` — на все выводимые данные, для которых ширина поля вывода превышает ширину данных (включая пробел, для которого установлена ширина поля вывода в 3 знакоместа); манипулятор `setw(4)` действует только на число 10 (0xA), `setw(3)` — только на выводимый после него пробел, `setw(8)` — только на число 30 (0x1E); `setw(2)` не влияет на вывод числа 257 (0x101), так как для его вывода нужно три знакоместа.

Порядок использования манипуляторов не важен.

### 7.6.2. Ввод-вывод с помощью `libc`

Стандартная библиотека языка C (`libc`) включает множество функций ввода-вывода. Как и для любых функций стандартной библиотеки, подробную информацию о них можно найти в третьем разделе ман-страниц GNU/Linux. В частности, команда `$ man 3 printf` выводит в консоль данные о функции `printf()`, а также об используемых ею форматных полях.

#### Вывод

Простой вывод строки `s` на стандартный вывод можно выполнить с помощью функции `int puts(const char *s)`.

Форматированный вывод данных различных простых типов осуществляется с помощью функции `int printf(const char *format, ...)`. Первый аргу-

мент этой функции — форматная строка, содержащая некоторый набор форматных полей, а следующие — выводимые данные простых типов. Каждому полю обычно соответствует один параметр, выводимый функцией *printf()* на экран, для некоторых видов полей также требуется один или два дополнительных целых параметра, уточняющих формат вывода.

Несмотря на то, что второй и последующие аргументы могут иметь различные типы, функция *printf()*, как и *scanf()*, не является ни шаблонной, ни перегруженной. Соответственно, её имя не декорируется и изменяется по правилам C-функций.

Форматное поле начинается с символа % и заканчивается буквой, обозначающей обобщённый вид формата, в соответствии с которым выводится значение (таблица 7.2). Символ процента можно вывести с помощью поля %, которому не должно соответствовать никакого параметра функции *printf()*.

Между символом процента и форматом вывода могут присутствовать (но не обязательно) символы, уточняющие формат. Они располагаются в следующем порядке:

- флаги (таблица 7.3; флагов может быть несколько, один или ни одного);
- минимальная ширина поля вывода — число, звёздочка \* (значение задаётся параметром) или другое описанное в документации значение;
- точность вывода (для целых форматов — общее количество выводимых цифр, для вещественных — после запятой); точность описывается аналогично минимальной ширине поля — число, звёздочка \* или другое описанное в документации значение и отделяется от минимальной ширины точкой;
- модификатор размера.

Любой из уточняющих символов может быть опущен.

Таким образом, строка

```
1 printf("%+4d_03x\n", 19, 13);
```

поместит на стандартный вывод

Основные форматы вывода *printf()*

Таблица 7.2

Обозначение	Преобразование
<b>Форматы вывода целых чисел</b>	
d, i	Целое число выводится как знаковое десятичное значение
o, u, x, X	Целое число выводится как беззнаковое восьмеричное (o), десятичное (u) или шестнадцатеричное (x, X) значение без префикса
c	Целое число преобразуется к типу <code>unsigned char</code> , после чего выводится символ с соответствующим кодом
<b>Форматы вывода вещественных чисел</b>	
e, E	Вещественное число выводится в десятичном экспоненциальном формате
f, F	Вещественное число выводится в формате с десятичной запятой
g, G	Вещественное число выводится в десятичном экспоненциальном формате или формате с десятичной запятой, так, чтобы результат был наиболее компактным
a, A	Вещественное число выводится в шестнадцатеричном экспоненциальном формате с двоичным порядком с префиксом
<b>Форматы вывода указателей</b>	
s	Байты по указанному адресу выводятся как строка, завершающаяся нулём
p	Указатель выводится как шестнадцатеричное значение

i 1900d

Регистр шестнадцатеричных цифр A—F соответствует регистру буквенного обозначения формата.

Если целому аргументу функции *printf()* поставить в соответствии вещественный формат вывода или наоборот, к нужному типу приводится не сам аргумент, а его адрес. То есть число, выведенное на экран, не будет соответствовать значению аргумента.

## Основные флаги вывода *printf()*

Таблица 7.3

Обозначение	Действие
#	Значение выводится в «альтернативной форме». В частности, для форматов <i>o</i> и <i>x/X</i> добавляется префикс системы счисления (0 и 0 <i>x</i> /0 <i>X</i> соответственно)
0	Если длина значения в заданном формате меньше заданной минимальной ширины поля вывода, значение дополняется ведущими нулями до заданной ширины. При одновременном задании 0 и - флаг 0 игнорируется
-	Если длина значения в заданном формате меньше заданной минимальной ширины поля вывода, значение выравнивается по левому краю поля
< > (пробел)	Перед беззнаковыми числами (в позиции, соответствующей знаку) вставляется пробел. При одновременном задании < > и + флаг < > игнорируется
+	Перед положительными числами указывается знак +

## 7.7. Отладочная печать

Читай запись дел твоих!

Ныне ты сам в состоянии требовать от себя отчёт.

*Коран. 17.15*

В некоторых случаях использование окон отладчика по какой-то причине неудобно, в частности, иногда необходимо сформировать файл протокола, содержащий шестнадцатеричные представления множества объектов.

### 7.7.1. Средства исследования переменных

Язык C++ предоставляет множество средств для исследования структуры объектов во время выполнения программы.

## Идентификация типа

Для получения информации о типе объекта во время исполнения программы (run-time type identification — RTTI, раздел [**expr typeid**] стандарта) в C++ используется оператор `typeid`. Оператор `typeid` принимает в качестве параметра имя типа или переменной и возвращает `const std::type_info`.

Класс `std::type_info` включает метод `name()`, возвращающий строку, характеризующую тип (не имя типа и не формат вывода). В частности, `typeid(int).name()` вернёт "i", `typeid(double).name()` — "d", а `typeid(long double).name()` — "e". Составные типы характеризуются длинными многокомпонентными строками.

## Размер объекта

Размер выделяемой под объект памяти можно узнать, используя оператор `sizeof`. Согласно стандарту C++ (раздел [**expr sizeof**]), оператор `sizeof` возвращает количество байтов, используемое для представления операнда.

Размеры узких символьных типов `sizeof(char)`, `sizeof(signed char)` и `sizeof(unsigned char)` равны 1, для остальных стандартных типов определяется реализацией.

## Дамп памяти

С точки зрения языка высокого уровня, указатели, хранящие адреса объектов различных типов, сами имеют разные типы. Это сделано для защиты от ошибок, чтобы не попытаться рассмотреть в памяти то, чего там нет (и не получить очень странное значение, например, нечаянно прочитав часть вещественного числа как целое) или не испортить соседние переменные, записывая объект большого размера в область, зарезервированную под меньший.

С точки зрения более низкого уровня, адреса объектов различных типов ничем не различаются, и программист сам должен помнить размер, структуру и назначение каждого объекта в памяти. Все адреса имеет один размер, соответ-

ствующий разрядности платформы, и, теоретически, любой адрес может быть преобразован к любому типу указателя.

На практике не любое преобразование указатель-указатель имеет смысл. Так как размер любого типа кратен размеру `char`, адрес любого объекта `x` может быть преобразован в указатель типа `char *`. Таким образом мы получим доступ к байтам, составляющим объект, как к массиву `char`'ов; размер этого массива — количество байтов в `x` — равен `sizeof(x)`.

Язык C++ позволяет преобразовать указатели на разные типы только с помощью самого наглого и не портируемого оператора преобразования — `reinterpret_cast`:

```
1 char *p = reinterpret_cast<char *>(&x);
```

В программировании на высоком уровне не рекомендуется использование `reinterpret_cast` вообще и преобразование типов указателей в частности, так как это небезопасно.

Низкоуровневое программирование небезопасно само по себе.

Тем не менее, преобразование адреса объекта в адрес цепочки байт, которая затем выводится в отладочный протокол — весьма эффективное средство исследования структуры этого объекта.

## Вывод в поток

Для формирования файла-протокола можно воспользоваться потоками вывода. Ассоциировав в программе какой-либо файл с потоком типа `fstream`, мы получим возможность записи протокола непосредственно в этот файл. Выводя протокол в стандартный поток вывода (`cout`), мы сможем наблюдать протокол в консоли или сохранить его в файл, используя перенаправление стандартного потока вывода в командном интерпретаторе (в частности, `bash`):

```
1 $ program > /tmp/log.txt
```



(данная команда запускает программу `program` и направляет её стандартный вывод не в консоль, а в файл `/tmp/log.txt`). Второй способ более универсален, поэтому во всех примерах будем рассматривать стандартный поток вывода.

Таким образом, вывести в поток `cout` первый байт по адресу `p` в том, виде, который использован в окне `Memory dump` (две шестнадцатеричные цифры с ведущим нулём), можно следующим образом:

**Листинг 7.12.** Вывод первого байта по адресу `p` в шестнадцатеричном виде

```
1 unsigned char *p;  
2 cout << hex << setfill('0') << setw(2) <<  
   static_cast<unsigned>(*p);
```

используется тип `unsigned char`, чтобы расширение до `unsigned` гарантированно было беззнаковым, и выводимое значение поместилось в два знакоместа.

Адрес следующего элемента (с учётом того, что `p` — указатель на однобайтовый тип — следующего байта) равен `p+1` и так далее. Соответственно, зная адрес начала переменной, можно вывести в поток все составляющие её байты, симитировав функциональность окна `Memory dump`.

Зная размер переменной (`sizeof(x)`), можно вывести на экран её побайтовое представление. Оно может не совпадать с шестнадцатеричным представлением из-за порядка байтов в словах. В шестнадцатеричном представлении цифры выводятся по-арабски, от старшей к младшей; побайтовый вывод показывает реальный порядок байтов в памяти (на платформе `x86` — от младшего к старшему), при этом цифры каждого байта выводятся от старшей к младшей.

### 7.7.2. Автоматизация отладочной печати

Для автоматизации отладочной печати лучше реализовать её в виде отдельной функции, чтобы упростить внесение изменений. Назовём эту функцию `MemoryDump()`. Пусть `MemoryDump()` получает в качестве аргумента исследуемый объект `x`, печатает данные о нём в стандартном потоке вывода и возвращает ничего (`void`).

Чтобы избежать приведения типа аргумента и, соответственно, искажения данных о нём, необходима отдельная реализация `MemoryDump()` для каждого возможного типа аргумента; при этом текст реализаций `MemoryDump()` для различных типов аргументов будет полностью совпадать. Для этого идеально подходит механизм **шаблонов (templates)**. Эта возможность C++ позволяет определить семейство функций, которые могут работать с различными типами данных. Так как нам нужно варьировать только тип аргумента, у шаблона будет один параметр — имя типа аргумента функции:

**Листинг 7.13.** Параметр шаблона — тип печатаемого значения

```
1 template<typename T>
2 void MemoryDump(T ...x)
3 {
4 ...
5 }
```

Для доступа к памяти, где реально находится объект, необходимо передать этой функций указатель или ссылку на него. С точки зрения низкого уровня указатель и ссылка — одно и то же; на уровне C++ передача по ссылке позволит использовать те же синтаксические конструкции, что и для исследования локальной переменной.

**Листинг 7.14.** Заголовок шаблона для печати дампа памяти

```
1 template<typename T>
2 void MemoryDump(T &x)
3 ...
```

Так как планируется не изменение, а только печать аргумента, правила хорошего тона требуют для него спецификатора `const` (соответственно, используемый в тексте `MemoryDump()` указатель `p` тоже должен быть константным). Получим окончательный вариант шаблонной функции отладочной печати дампа памяти в виде листинга 7.15.

**Листинг 7.15.** Шаблон для печати дампа *x*

```

1 template<typename T>
2 void MemoryDump(const T &x)
3 {
4     const unsigned char *p
5         = reinterpret_cast<const unsigned char *>(&x);
6
7     cout << "Type:_" << typeid(x).name() << "_Value:_" << x << endl
8         << "Size:_" << sizeof(x) << endl
9         << "Dump:_" << hex << uppercase << setfill('0');
10
11     for(size_t i = 0; i < sizeof(x); ++i)
12     {
13         cout << setw(2)<< static_cast<unsigned>(*(p+i)) << "_";
14     }
15     cout << dec << endl << endl;
16 }

```

Эта функция позволяет вывести в стандартный поток вывода байты любой переменной в том порядке, в котором они лежат в памяти. В частности, результатом следующего кода:

#### Листинг 7.16. Печать сведений о трёх переменных

```

1 int i = 1;
2 double d = 1;
3 long double ld = 1;
4
5 MemoryDump(i);
6 MemoryDump(d);
7 MemoryDump(ld);

```

будет:

#### Листинг 7.17. Значения, размеры и дампы переменных

```

1 Type: i Value: 1
2 Size: 4
3 Dump: 01 00 00 00
4
5 Type: d Value: 1
6 Size: 8
7 Dump: 00 00 00 00 00 00 F0 3F
8
9 Type: e Value: 1
10 Size: 12
11 Dump: 00 00 00 00 00 00 00 80 FF 3F 00 00

```

Для каждого такого вызова компилятор формирует отдельную функцию — реализацию шаблона `MemoryDump()` для конкретного типа аргумента. Такие реализации перегружают друг друга и имеют одно имя для C++; с точки зрения компоновщика разные реализации имеют разные имена, так как по-разному декорируются.

При вызове шаблонной функции `MemoryDump()` конкретная вызываемая реализация определяется типом передаваемого фактического параметра. Указывать реализацию явно (например, `MemoryDump<int>(i)`) здесь не только не нужно, но и вредно — если указанный тип реализации не совпадёт с настоящим типом передаваемого параметра, результат будет некорректен.

Для указателя данная функция выведет размер и представление в памяти самой переменной-указателя, а не тот фрагмент памяти, куда он указывает.

Если необходимо напечатать именно память, на которую указывает аргумент, необходимо модифицировать шаблонную функцию `MemoryDump()`. Получим листинг 7.18 (`CellCount` — количество ячеек типа `T` по адресу `px`).

#### Листинг 7.18. Шаблон для печати дампа памяти по адресу `px`

```
1 template<typename T>
2 void PointerMemoryDump(T *px, int CellCount)
3 {
4     const unsigned char *p
5         = reinterpret_cast<const unsigned char *>(px);
6
7     size_t BytesCount = sizeof(*px)*CellCount;
8
9     cout << "Type:_" << typeid(px).name() << "_Value:_" << px << endl
10         << "Size:_" << sizeof(px) << endl
11         << "Dump:_" << hex << uppercase << setfill('0');
12
13     for(size_t i = 0; i < BytesCount; ++i)
14     {
15         cout << setw(2)<< static_cast<unsigned>(*(p+i)) << "_";
16     }
17     cout << dec << endl << endl;
18 }
```

Тогда результатом выполнения листинга 7.19, печатающего сведения об указателе *s* и восьмибайтовом значении по этому указателю,

**Листинг 7.19.** Печать сведений об указателе и о значении

```
1 char *s = "abcdef";  
2  
3 MemoryDump(s);  
4 PointerMemoryDump(s,8);
```

будет следующий вывод:

**Листинг 7.20.** Дамп памяти собственно указателя и значения по указателю

```
1 Type: Pc Value: abcdef  
2 Size: 4  
3 Dump: 08 96 04 08  
4  
5 Type: Pc Value: abcdef  
6 Size: 4  
7 Dump: 61 62 63 64 65 66 00 54
```

Таким образом, шаблонная функция `MemoryDump(s)` выводит данные об указателе *s*, а `PointerMemoryDump(s,8)` — о строке *s*, включающей семь узких символов (шесть латинских букв и завершающий строку нулевой символ). Видно, что, кроме семи байт строки, листинг 7.19 выводит ещё один, лишний, байт.

В отладочной печати, как и при изучении содержимого памяти при помощи инструментов интерактивной отладки, необходимо различать те переменные, которые содержат интересующее программиста значение и переменные-указатели, которые содержат адрес интересующего значения.

## Контрольные вопросы

1. Как называется головная функция программы на C++?
2. Какие целые типы языка C++ вы знаете?
3. Какие вещественные типы языка C++ вы знаете?
4. Какие операторы преобразования типов C++ вы знаете?

5. Как записываются целые, вещественные, строковые литералы?
6. Как автоматизировать отладочную печать в C++?

## Заключение

Усердный в службе не должен бояться своего незнания;  
ибо каждое новое дело он прочтёт.

*К. П. Прутков.*

*Плоды раздумья. Мысли и афоризмы*

В данном пособии описана только малая часть безграничных возможностей Ассемблера.

Многие возможности низкоуровневого программирования доступны только на уровне операционной системы и тесно связаны с её архитектурой и особенностями. Соответственно, дальнейшее изучение архитектуры и системы команд x86, а также программирования на языке Ассемблера, неизбежно сопряжено с изучением операционных систем.

Прикладное программирование на ассемблере широко применяется в задачах криптографии. При программной реализации большинства современных алгоритмов использование только команд языков высокого уровня, позволяющих программисту абстрагироваться от конкретного представления данных, неэффективно. Ассемблер, напротив, легко позволяет рассмотреть блок данных одновременно как число и битовую строку. Кроме того, разработчики современных процессоров постоянно вводят новые команды, облегчающие реализацию популярных асимметричных схем.

## Приложение А

# Лабораторный практикум GNU Assembler

### Требования к выполнению лабораторных работ

#### Операционная система и компилятор

Все задания, если это не сказано особо, выполняются под управлением операционной системы GNU/Linux. Задания всех лабораторных работ, кроме Л1 и Л2, если не указан язык, выполняются на ассемблере, в виде вставок в программу на языке высокого уровня либо отдельных модулей.

Программы на чистом C++ (без ассемблерных вставок) должны собираться на любой платформе любым компилятором (недопустимо использование платформозависимых элементов).

Для разработки программ на языке C++ могут быть использованы кросс-платформенные среды Code::Blocks, Qt Creator, TheIDE (Ultimate++), Codelite. Для сборки и отладки программ в среде Microsoft Windows с использованием компилятора Microsoft используется среда Microsoft Visual Studio.

#### Отчёт и оформление

По результатам выполнения лабораторной работы оформляется отчёт в формате plain text, L<sup>A</sup>T<sub>E</sub>X, OpenDocument или PDF, а также программный код. Заголовок отчёта должен включать имя группы и ФИО авторов, а также тему работы. Отчёт должен содержать для каждого задания:

- номер и текст задания;
- номер и текст варианта (если есть);
- результат выполнения задания: результаты измерений с комментариями и ссылки на программы.



Если отчёт оформлен не как комментарии к коду, полный текст программ копировать в отчёт не нужно (текст программ предоставляется отдельно).

Отчёт в формате plain text может быть совмещён с программным кодом (помещён в комментарии соответствующих модулей). Отчёт по лабораторным работам Л4–Л7 *должен* быть оформлен как комментарии к программному коду.

В начале каждого файла с исходным кодом должен находиться *комментарий*, содержащий краткое описание модуля. Описание модуля с головной функцией должно включать *тему лабораторной работы*, а также *тексты задания и варианта*.

### Парная работа

Лабораторная работа выполняется совместно двумя сидящими рядом студентами. Состав пар на разных занятиях может различаться. При нечётном количестве присутствующих допускается не более одной команды, состоящей из трёх или одного человека. Команда, независимо от количества участников, выполняет один вариант задания, соответствующий номеру команды.

Номер пары или команды в группе должен быть уникален. Его можно рассчитать как

$$\text{№ пары} = \left\lfloor \frac{\max(\text{ПК участника})}{2} \right\rfloor \quad (\text{A.1})$$

то есть пара за компьютерами 1 и 2 получит номер 1, 3 и 4 — номер 2 и т. д. В спорных случаях номер пары или команды может быть назначен преподавателем.

### Оценивание

Максимально возможная оценка лабораторной работы — 11 баллов. Работа, выполненная не полностью (в том случае, если не выполнено или выполнено с ошибками менее 25% обязательных заданий), может быть зачтена с оценкой ниже максимальной. Работа, выполненная не самостоятельно, *не может быть зачтена*.

Из оценки лабораторной работы, сданной с опозданием более чем на одно занятие без уважительной причины, *вычитается величина опоздания*, то есть лабораторная работа, сданная с опозданием на два занятия, может быть оценена (без учёта бонусных заданий) максимум на 9 баллов, с опозданием на три занятия — максимум на 8 баллов, и т. д.

Если лабораторные работы выполняются не по порядку, но при этом на каждом занятии (кроме, возможно, первого) выполняется и сдаётся какая-либо работа, то они оцениваются как сданные без опоздания.

Лабораторная работа, сданная на зачётной неделе, оценивается как 1 балл. Не зачтённые лабораторные работы помечаются в ОРИОКС литерой «н».

### **Необязательные (бонусные) задания**

Задания, отмеченные как **«Бонус»**, *необязательны*; за их выполнение начисляются дополнительные баллы. Баллы за необязательные задания добавляются к оценке за лабораторную работу (если это возможно без переполнения) или к колонке «бонус/зачёт».

Бонусные задания могут быть сданы либо одновременно с соответствующей лабораторной работой, либо (при условии, что это не мешает преподавателю принимать у других студентов обязательные задания) после неё. На зачётной неделе бонусные задания не принимаются.

В тексте задания указано *максимальное* количество дополнительных баллов. Количество баллов, начисленных за бонусные задания, зависит только от качества выполнения задания, но не от времени сдачи.

### **Замечания и дополнения**

Замечания и дополнения к данному документу можно отправить в письменном виде по адресу [illinc@inbox.ru](mailto:illinc@inbox.ru).

Принятое замечание/дополнение приносит первому приславшему его студенту от 1 до 8 бонусных баллов.

## Лабораторная работа 1.

### Представление данных в ЭВМ

**Цель работы:** изучить форматы представления целых чисел и символьной информации в ЭВМ.

#### Л1.1. Задание на лабораторную работу

**Задание 1.** Изучите, как интерпретируется одна и та же область памяти, если она рассматривается как знаковое или беззнаковое число, а также как одно и то же число записывается в различных системах счисления.

Необходимо сравнить:

- а) беззнаковую интерпретацию переменной в шестнадцатеричной форме;
- б) беззнаковую интерпретацию в десятичной форме;
- в) знаковую интерпретацию в десятичной форме.

Для этого определите и запишите в отчёт десятичное, двоичное (16 бит) и шестнадцатеричное представления шестнадцатитбитных чисел  $x$  и  $y$ , а также беззнаковую интерпретацию этого представления в десятичном виде.

$(N^o - 1) \% 4 + 1$	Вариант
1	$x = -34, y = 2^{15} + 7$
2	$x = -2, y = 2^{14} + 1$
3	$x = -2^{15}, y = 2^{15}$
4	$x = -7, y = 2^{16} - 2$

Для получения различных интерпретаций одного и того же кода в C++ можно использовать объединения или оператор `reinterpret_cast`. Шестнадцатеричное и десятичное представление можно получить, используя различные форматы вывода. Шестнадцатеричный формат вывода не предусматривает знака,

то есть шестнадцатеричное представление знаковой переменной будет соответствовать её беззнаковой интерпретации.

**Задание 2.** Найдите и выпишите в отчёт минимальное и максимальное 16-битное число со знаком и без знака в формах представления (а), (б), (в) и в двоичной форме (4 числа, каждое из которых представлено в 4 формах).

**Задание 3.** Разработайте программу на языке C++, выполняющую над беззнаковыми шестнадцатитрибитными целыми числами следующие поразрядные операции (результат должен печататься в десятичной и шестнадцатеричной формах):

- бинарные  $x \wedge y$  (конъюнкция),  $x \vee y$  (дизъюнкция),  
 $x \oplus y$  (сложение по модулю два);
- унарные  $\neg x$  (отрицание),  
 $\text{neg}(x)$  (дополнение до двух,  $x + \text{neg}(x) = 2^{\text{разрядность } x}$ );
- $x \ll y$  (логический сдвиг влево),  $x \gg y$  (логический сдвиг вправо).

$(\text{№} - 1) \% 3 + 1$	Вариант	
1	$\begin{cases} x = 0\text{xFFEE} \\ y = 0\text{x0001} \end{cases}$	$\begin{cases} x = 0\text{x000E} \\ y = 0\text{x0003} \end{cases}$
2	$\begin{cases} x = 0\text{x8008} \\ y = 0\text{x1111} \end{cases}$	$\begin{cases} x = 0\text{x0005} \\ y = 0\text{x0002} \end{cases}$
3	$\begin{cases} x = 0\text{x9211} \\ y = 0\text{x0004} \end{cases}$	$\begin{cases} x = 0\text{x0009} \\ y = 0\text{x0013} \end{cases}$

**Задание 4.** Измените в программе из задания 3 тип переменных на знаковый. Объясните изменение (или неизменность) результата.

**Задание 5. Бонус (+1 балл).** Разработайте программу на языке C++ (или дополните программу из задания 3), которая расширяет шестнадцатитрибитное представление числа  $x$  до тридцатидвухбитного, рассматривая числа как

- знаковые (signed);
- беззнаковые (unsigned).

$(\text{№} - 1) \% 5 + 1$	Вариант
1	$x = 0xFF00$
2	$x = 0xF123$
3	$x = 0x8001$
4	$x = 0xABCD$
5	$x = 0xEE22$

Результат должен печататься в десятичной и шестнадцатеричной формах.

**Примечание:** Расширение числа происходит, в частности, при присваивании — если размер приёмника больше размера источника.

### Задание 6. Бонус (+1 балл).

Напишите программу, демонстрирующую переполнение целых чисел со знаком и без знака. Для этого опишите целое число со знаком и инициализируйте его максимально возможным для данного типа значением, результат распечатайте в шестнадцатеричном виде. Затем прибавьте к этому числу единицу, результат распечатайте также в шестнадцатеричном виде. Выполните аналогичные действия для беззнаковой переменной. Выпишите в отчёт распечатанные значения переменной, поясните полученные результаты.

**Задание 7.** Определите и выпишите в отчёт, как хранятся в памяти компьютера:

- целое число `0x12345678`; по результату исследования определите порядок следования байтов в словах для вашего процессора:
  - Little-Endian (от младшего к старшему, порядок Intel);
  - Big-Endian (от старшего к младшему, порядок Motorola);
- строки `"abcd"` и `"абвг"` (массив из `char`);
- «широкие» строки `L"abcd"` и `L"абвг"` (массив из `wchar_t`).

**Задание 8.** При помощи оператора `sizeof` выясните, сколько байтов занимают переменные следующих типов: `char`, `bool`, `wchar_t`, `short`, `int`,

long, long long, float, double, long double, size\_t, ptrdiff\_t, void\*. Результаты оформите в отчёте в виде таблицы, указывая для каждого типа его назначение.

Для выполнения единообразных действий над переменными различных типов используются макросы препроцессора C или шаблоны C++.

Проверьте, соответствуют ли размеры типов современному стандарту C++.

**Задание 9.** Запустите программу (программы) из заданий 7–8 на двух других платформах, доступных на ВЦ — 32- и 64-разрядной версиях Microsoft Windows и повторите измерения.

Для каждого из заданий дополните таблицу результатами новых измерений. Платформы, для которых необходимо провести измерения:

- GNU/Linux Ubuntu, архитектура x86 (компилятор GCC, среда любая);
- 32-разрядная версия Microsoft Windows (ВЦ, среда Microsoft Visual Studio);
- 64-разрядная версия Microsoft Windows (терминал Skylab, среда Microsoft Visual Studio, 64-битная конфигурация).

Результаты однотипных измерений, выполненных на различных платформах, должны быть сгруппированы в таблицу или последовательно описаны в одном разделе.

**Задание 10. Бонус (+2 балла за платформу).** При подготовке к работе соберите и запустите программу (программы) из заданий 7–8, используя платформу и/или компилятор, отличные от доступных на ВЦ. Результаты измерений с пояснениями внести в конспект (дополнить таблицы результатов заданий 7–8).

## Л1.2. Дополнительные бонусные и штрафные баллы

+2 балла — корректная автоматизация с помощью макросов препроцессора C или шаблонов C++.

–4 балла — неудобочитаемый отчёт.

*Пример неудобочитаемого фрагмента отчёта:*

Задание 3.



```
lab1
10
A
Process returned 0 (0x0)   execution time : 0.021 s
Press ENTER to continue.
```

### *Возможная альтернатива:*

Задание 3. Разработайте программу на языке C++, выполняющую над беззнаковыми целыми числами следующие операции (результат должен печататься в десятичной и шестнадцатеричной формах):

`11 & 14 = 10 (0xB & 0xE = 0xA)`

## **Л1.3. Рекомендации**

Для просмотра содержимого памяти используйте возможности отладчика (окно Memory или подобное ему) или отладочную печать. Не стоит использовать для исследования представления объектов в памяти окно Watches, так как оно по определению показывает состояние объекта, абстрагированное от представления.

Перенаправить стандартный вывод программы в файл в большинстве оболочек, в том числе `bash`, возможно с помощью символа `>`.

Скопировать вывод программы из терминала в отчёт можно, используя буфер обмена `primary`. Для этого необходимо выделить мышью необходимый текст в терминале и затем щёлкнуть средней кнопкой мыши в том месте редактора, где планируется вставка. После закрытия текста-источника содержимое буфера `primary` теряется, поэтому вставку желательно проводить непосредственно после выделения фрагмента вывода в терминале.

## Л1.4. Теоретические сведения

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

1.2.1 Единицы измерения

1.2.2 Порядок следования байтов

3.2 Сегменты памяти

2 Представление данных

7 Программирование на языке высокого уровня: C++

4.1.2 Этапы компиляции

## Отладка в интегрированных средах разработки

Различные IDE для отладки программ используют собственные инструменты или (чаще) предоставляют графический интерфейс к внешним отладчикам. Обычно используется отладчик gdb (GNU Symbolic Debugger).

Базовые инструменты различных отладчиков схожи. Рассмотрим инструменты отладки на примере среды Qt Creator (рис. Л1.1) [76].

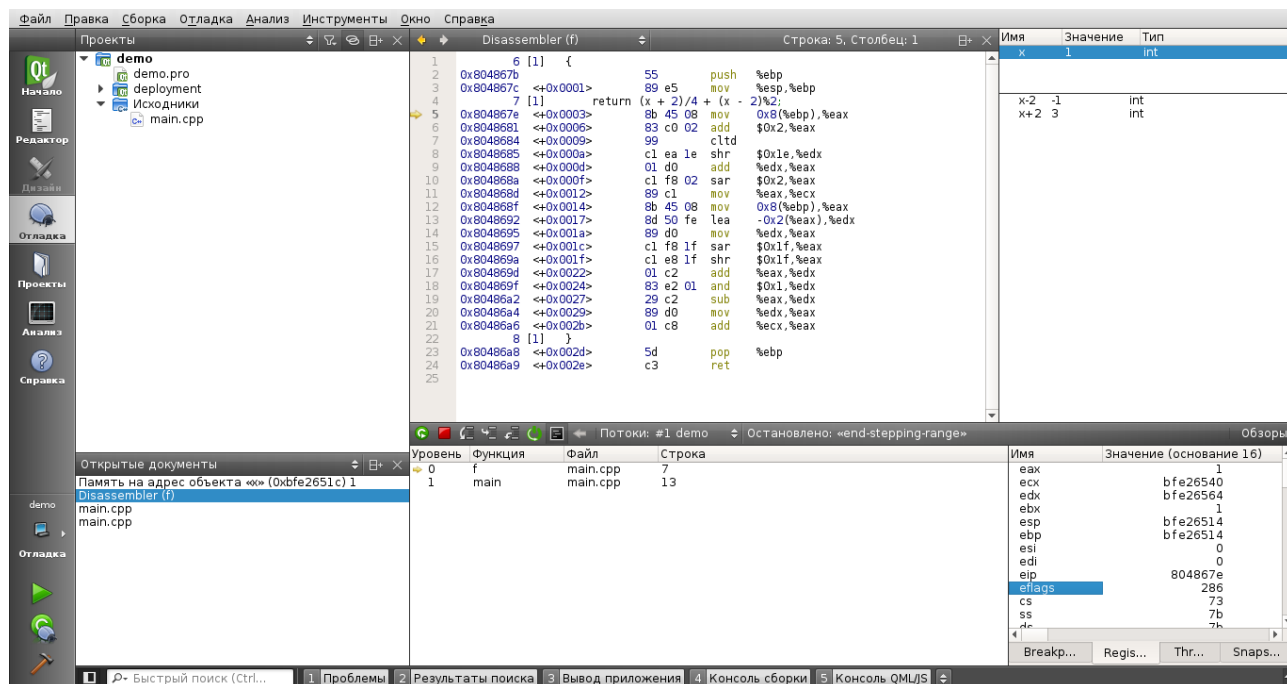


Рис. Л1.1. Отладка с помощью Qt Creator



## Точки останова

Точки останова представляют место или набор мест в коде, которые при выполнении прервут отлаживаемую программу и передадут управление пользователю. Пользователь может просмотреть состояние прерванной программы или продолжить выполнение построчно или непрерывно.

Обычно точки останова связаны с файлом исходных кодов и строкой или началом функции (рис. Л1.2).

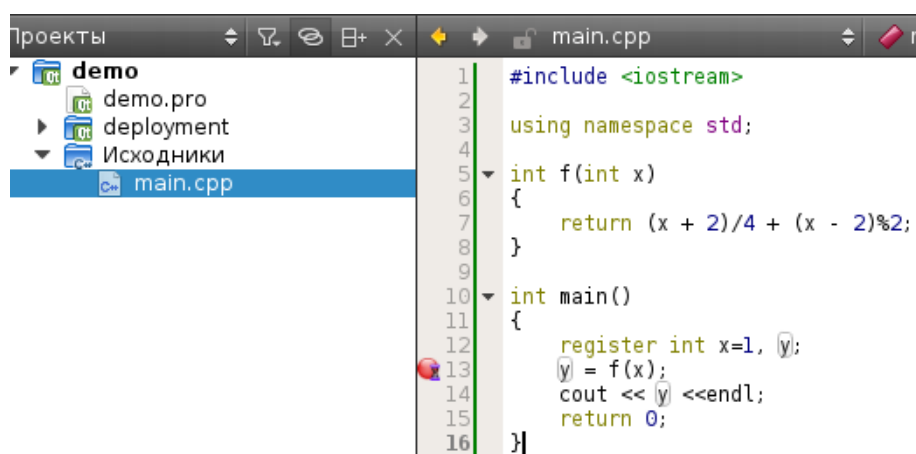


Рис. Л1.2. Точка останова в коде

Если выполнение не останавливается в точке останова, проверьте конфигурацию сборки. В большинстве сред разработки отладка возможна только для отладочной (Debug), но не для оптимизированной (Release) сборки. В Qt Creator возможна пошаговая отладка Release-сборки с использованием окон дизассемблера, регистров и памяти, но точки останова в исходном коде C++ в этом режиме игнорируются.

## Локальные и наблюдаемые переменные

Окно «Локальные и наблюдаемые переменные» отображает информацию о верхнем кадре стека (параметры функции и локальные переменные), а также о заданных выражениях. Окно «Локальные и наблюдаемые переменные» может быть использовано для изменения содержимого переменной простого типа данных (такого как *int* или *float*) во время останова программы (рис. Л1.3).

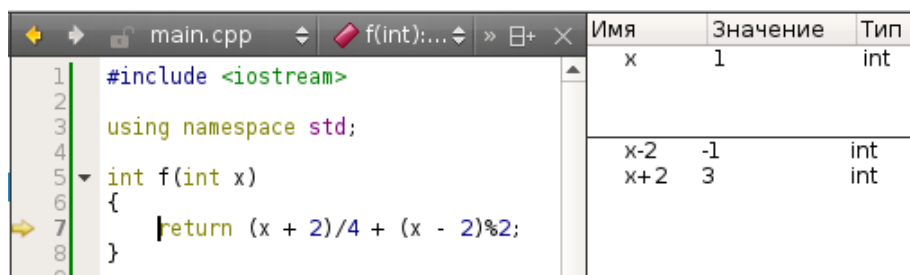


Рис. Л1.3. Локальные и наблюдаемые переменные Qt Creator

В среде Code::Blocks окно просмотра переменных можно открыть, используя меню *Debug* → *Debugging windows* → *Watches* (рис. Л1.4, внизу).

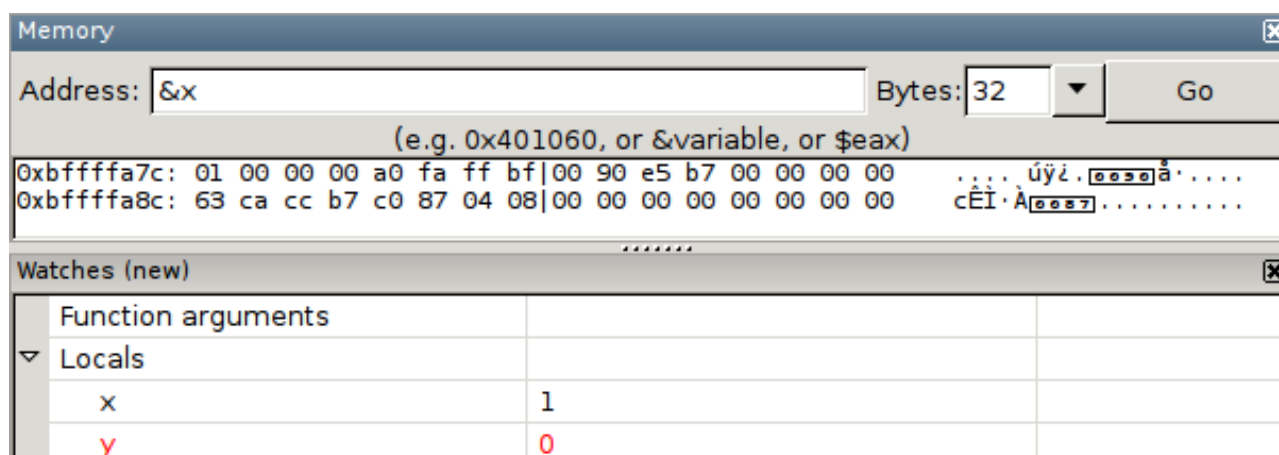


Рис. Л1.4. Отладочные окна Memory и Watches в Code::Blocks

В TheIDE для просмотра переменных используются вкладки *Autos*, *Locals* и *Watches* панели *Debug* (доступна во время отладки внизу экрана, скрывается/показывается по Esc).

### Обозреватель и редактор памяти

Обозреватель и редактор памяти отображают содержимое памяти. В Qt Creator как обозреватель, так и редактор памяти можно открыть для просмотра конкретной переменной, используя контекстное меню переменной в окне «Локальные и наблюдаемые переменные» (рис. Л1.5), при этом подсвечивается не только начало, но и размер объекта.

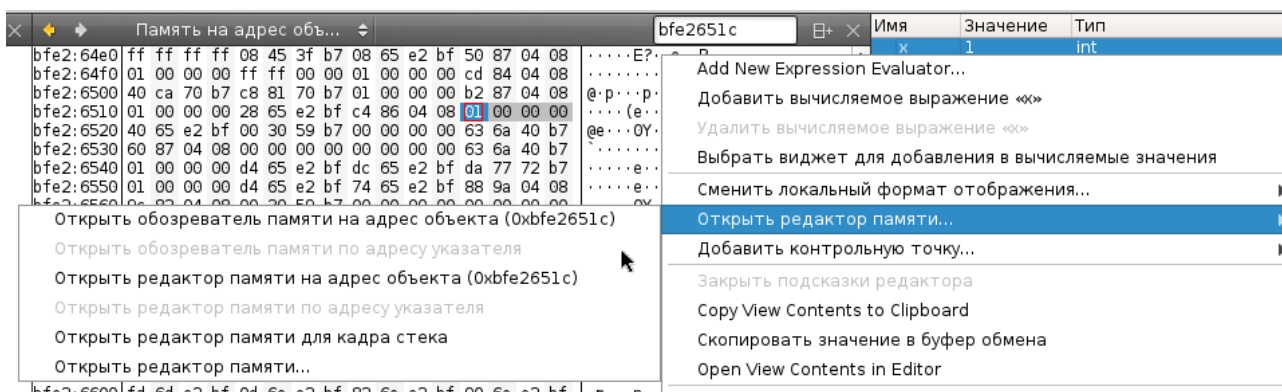


Рис. Л1.5. Обзорщик и редактор памяти Qt Creator

В среде Code::Blocks окно просмотра содержимого фрагмента памяти можно открыть, используя меню *Debug* → *Debugging windows* → *Memory dump* (рис. Л1.4, вверху). Адрес начала отображаемого фрагмента может быть литералом (конкретным адресом в памяти, как 0x401060), адресом переменной (&x), регистром, в котором содержится адрес (\$eax) или выражением C++ (результатом выражения должен быть указатель, например, (char\*)(&x) - 4 — адрес на 4 байта меньше, чем начало переменной x).

В текущей версии TheIDE нет инструмента для просмотра дампа памяти.

Содержимое памяти отображается как последовательность байтов. Каждый байт представлен как беззнаковое целое (в виде двух шестнадцатеричных цифр) и как ASCII-символ (справа). Байты разбиты на строки по 16; слева перед строкой показан адрес первого байта в строке в шестнадцатеричном виде.

## Создание 64-битной конфигурации Microsoft Visual Studio

Для создания 64-битной конфигурации Microsoft Visual Studio запустите Microsoft Visual Studio в терминале Skylab (домен SIPC) и выполните следующие три шага (рис. Л1.6) [34]:

Шаг 1. Запустить менеджер конфигураций (рис. Л1.6, а).

Шаг 2. В менеджере конфигураций выбрать поддержку новой платформы (рис. Л1.6, б).

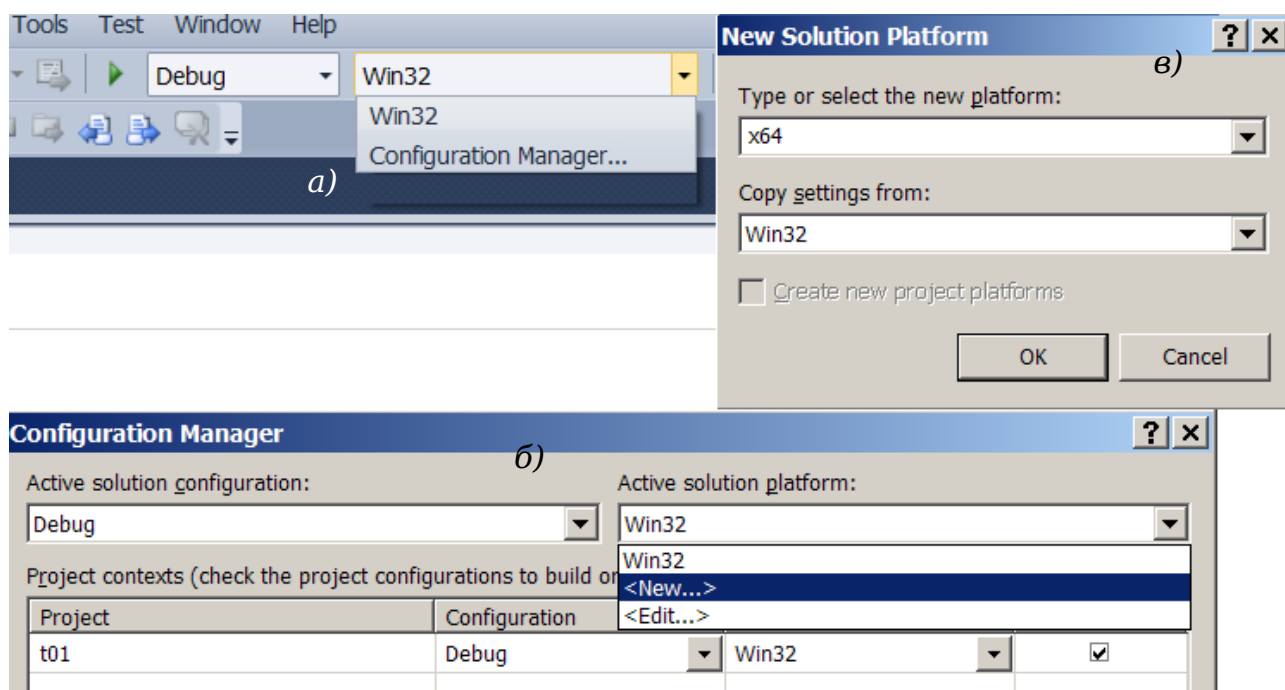


Рис. Л1.6. Создание 64-битной конфигурации для проекта  
в Microsoft Visual Studio

Шаг 3. Выбрать 64-битную платформу (x64), а в качестве основы выбрать настройки от 32-битной версии (рис. Л1.6, в). Те настройки, которые влияют на режим сборки, среда Microsoft Visual Studio скорректирует сама.

### Л1.5. Вопросы

1. Как представляется символьная информация в компьютере в кодах ASCII, расширениях ASCII и различных кодировках Unicode?
2. Как хранятся русские буквы в «классических» и «широких» строках?
3. Как представляются целые числа со знаком и без знака?
4. Как перевести число в дополнительный код?
5. Как выполняются логические операции и сдвиги над строкой битов?
6. Что такое расширение чисел со знаком и без знака? Для чего нужны операции расширения?
7. Для чего нужно знать порядок следования байтов на вашем компьютере?

## Лабораторная работа 2.

### Отладка кода

**Цель работы:** научиться использовать инструменты отладки современных IDE; исследовать генерируемый компилятором код.

#### Л2.1. Задание на лабораторную работу

**Задание 1.** Разработайте программу на языке C++, вычисляющую три целых выражения от целого аргумента (в соответствии с вариантом).

а)

$(N - 1) \% 8 + 1$	Вариант
1	$y(x) = x \cdot 2$
2	$y(x) = -x - 1$
3	$y(x) = x \cdot 3$
4	$y(x) =  x $
5	$y(x) = x \cdot 5$
6	$y(x) = x/2$
7	$y(x) = x \bmod 4$
8	$y(x) = x^2$

б)

$(N - 1) \% 7 + 1$	Вариант
1	$y(x) = x + 17$
2	$y(x) = x \cdot 13$
3	$y(x) = x - 21$
4	$y(x) = x/11$
5	$y(x) = x \cdot 7$
6	$y(x) = x/3$
7	$y(x) = x^2$

$(N - 1) \% 3 + 1$	Вариант
1	$y(x) = \begin{cases} 13, & x \geq 0 \\ 7, & x < 0 \end{cases}$
2	$y(x) = \begin{cases} 0, & x < 7 \\ x, & x \geq 7 \end{cases}$
3	$y(x) = \begin{cases} x, & x > 13 \\ -1, & x \leq 13 \end{cases}$

в)

**Примечание:** Не обязательно вводить  $x$  с клавиатуры. Инициализация локальной переменной «узнаваемым» литералом упростит поиск её адреса.

Сборка всех заданий данной лабораторной работы выполняется одним компилятором при одних и тех же настройках.

Не все изменения кода на C++ приведут к видимым изменениям дизассемблированного кода.

**Задание 2.** Запустите программу и, используя инструменты отладчика (в частности, дизассемблер), изучите ассемблерный код, соответствующий вычислениям (для Code::Blocks — переключитесь на синтаксис AT&T и включите Mixed mode, чтобы в окне дизассемблера перед каждой группой команд, соответствующих одному оператору языка высокого уровня, явно отображался этот оператор).

Занесите ассемблерный код, соответствующий вычислению  $y(x)$ , в отчёт (код, не связанный с вычислением  $y(x)$ , копировать в отчёт не нужно!). Определите и прокомментируйте:

- обращение к переменным  $x$  и  $y$ ;
- арифметические и логические операции — сложение, вычитание, умножение, деление с остатком, деление на  $2^n$  и т. д. (по возможности);
- сравнения и передачу управления в ветвлениях.

**Задание 3.** Внесите в программу из задания 1, а) изменения (либо, что предпочтительнее, добавьте новые фрагменты кода, выполняющие аналогичные

вычисления для других переменных, используя макросы препроцессора или шаблоны C++).

- сделайте переменные глобальными;
- измените тип с `int` на `char`, `short`, `long` и `long long`;
- измените тип с `int` на `long double`.

Опишите в отчёте различия в ассемблерном коде.

**Задание 4.** Оформите вычисления из задания 1, а) как целую функцию от целого аргумента. Опишите в отчёте код вызова функции. Как передаётся аргумент? Как возвращается значение?

**Задание 5.** Измените тип аргумента и результата на вещественный. Опишите в отчёте код вызова функции. Как передаётся аргумент? Как возвращается значение?

**Задание 6. Бонус (+2 балла).** Используйте в функции статическую переменную. Как выглядит обращение к ней?

**Задание 7. Бонус (+2 балла за платформу):** запустите тестовую программу (программы), используя платформу и/или компилятор, отличные от GNU/Linux и GCC. Результаты с пояснениями внести в конспект.

## Л2.2. Теоретические сведения

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

3.2 Сегменты памяти

3.3 Регистры

4.1.2 Этапы компиляции

5.1 Особенности GNU Assembler

5.2.1 Общие команды

## Дизассемблер

Окно дизассемблера отображает дизассемблированный код для текущей функции (рис. Л2.1). В Qt Creator по умолчанию спрятано, перейти в режим работы на уровне инструкций процессора возможно во время отладки.

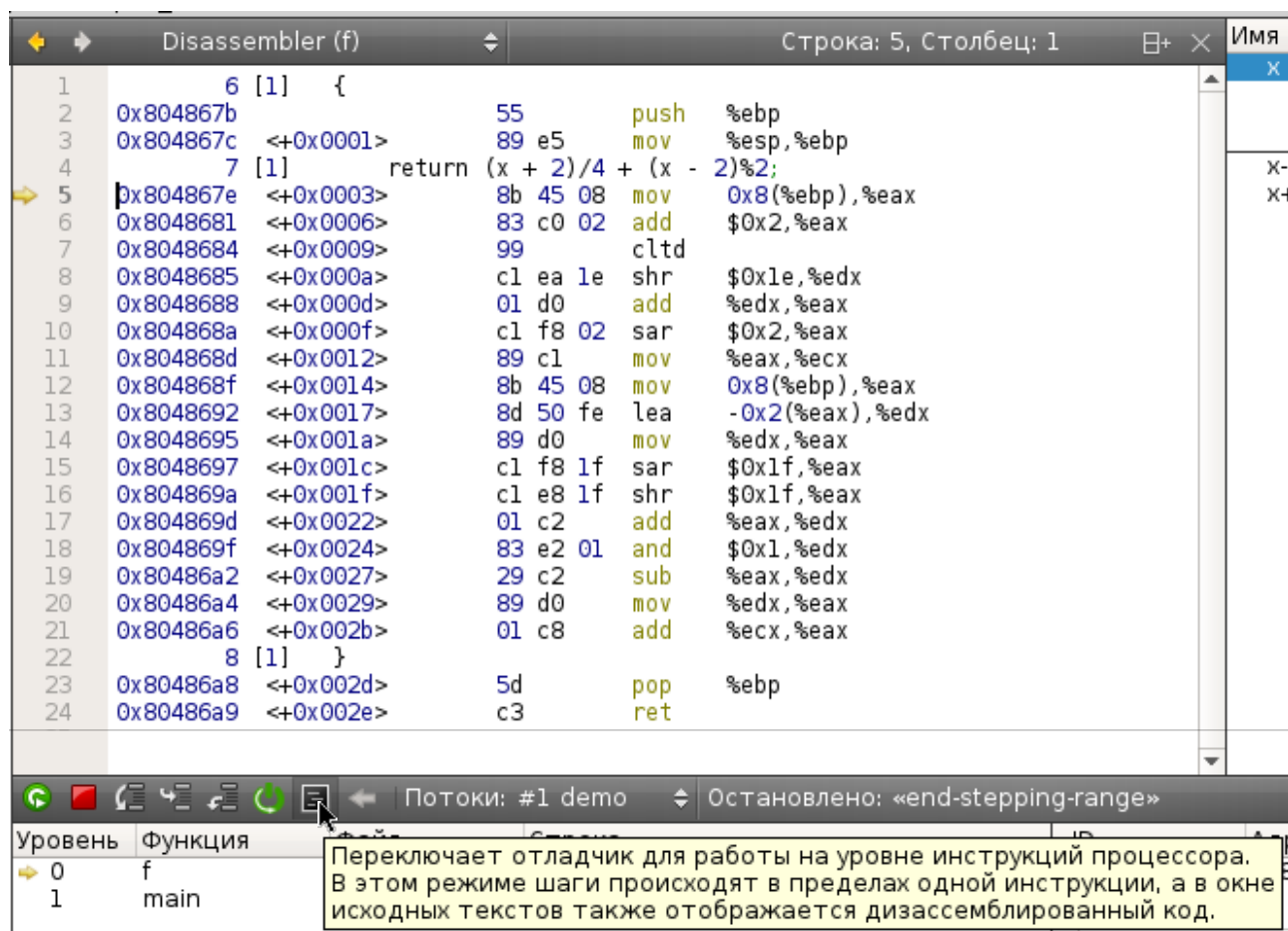


Рис. Л2.1. Дизассемблированный код программы (Qt Creator)

В среде Code::Blocks окно дизассемблера можно открыть, используя меню *Debug* → *Debugging windows* → *Disassembly* (для редактирования кода и для отладки используются разные наборы окон, поэтому открывать дизассемблер необходимо уже в процессе отладки). Опция *Mixed mode* в этом окне включает отображение строки C++ перед соответствующей ей группой команд. Кнопка *Adjust* позволяет перемотать дизассемблированный код к текущей команде. Кнопка *Save to text file* — сохранить его.



Рис. Л2.2 показывает окно дизассемблера среды Code::Blocks с включённой опцией *Mixed mode* (синтаксис AT&T).

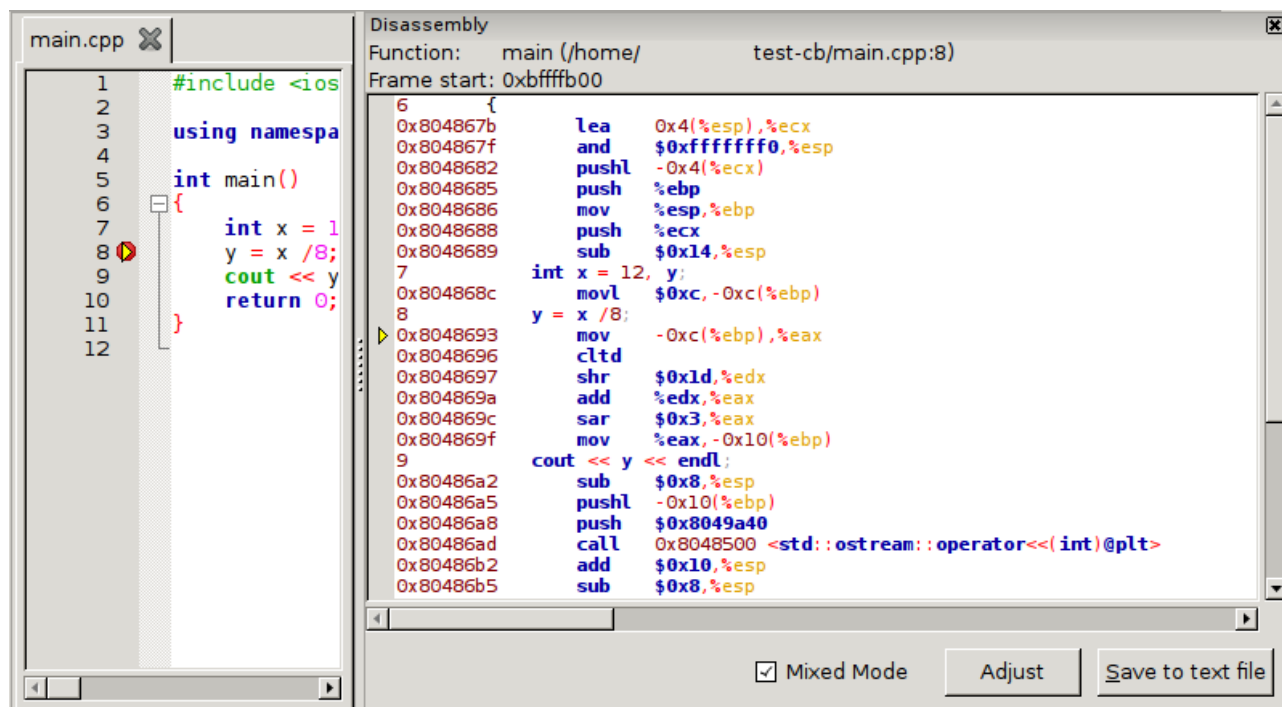


Рис. Л2.2. Дизассемблированный код программы (Code::Blocks)

Иногда при внесении изменений в программу и включённой опции *Mixed mode* ассемблерный код и операторы высокого уровня в окне дизассемблера не изменяются даже после пересборки проекта. В этом случае необходимо выключить и снова включить *Mixed mode* (останавливать отладку и пересобирать проект между этими действиями не надо).

В TheIDE панель дизассемблера всегда доступна во время отладки (справа). Её содержимое можно скопировать в буфер обмена, используя пункт меню *Debug* → *Copy disassembly*.

Содержимое окна дизассемблера в любой среде разработки — именно читабельное представление машинного кода отлаживаемой программы, а не промежуточная стадия компиляции модуля в машинный код, формируемая командой `g++ -S` (промежуточный результат компиляции можно увидеть, запустив ко-

манду `g++ -S <путь к cpp-файлу>` в консоли; в TheIDE можно использовать пункт меню *Build* → *Show assembler code for ...*).

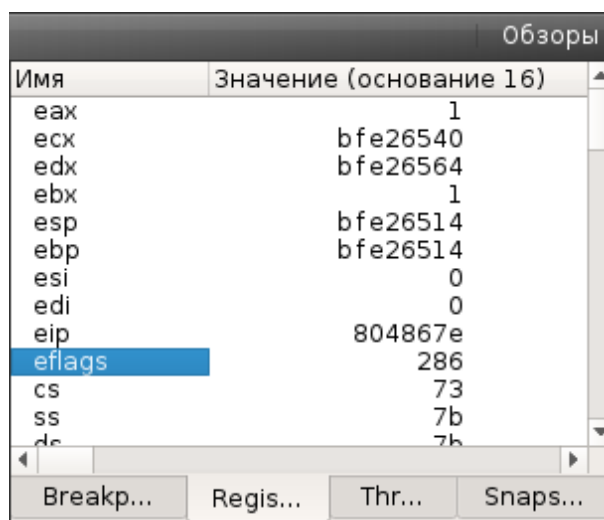
## Настройка синтаксиса ассемблерного кода

В зависимости от используемого отладчика и его настроек может различаться используемый синтаксис. Настроить вид дизассемблированного кода можно в настройках отладчика среды (для Code::Blocks *Settings* → *Debugger* → *GDB/CDB debugger : default* → *Choose disassembly flavor*).

В Qt Creator настроить синтаксис дизассемблера можно в окне параметров (меню *Инструменты* → *Параметры*). Для этого используется переключатель «Стиль Intel для дизассемблера», находящийся в разделе «Отладчик» этого окна на вкладке GDB.

## Регистры

Окно «Регистры» отображает текущее содержимое регистров (рис. Л2.3). В Qt Creator по умолчанию спрятано, включается через контекстное меню или меню «Окно».



Имя	Значение (основание 16)
eax	1
ecx	bfe26540
edx	bfe26564
ebx	1
esp	bfe26514
ebp	bfe26514
esi	0
edi	0
eip	804867e
eflags	286
cs	73
ss	7b
ds	7b

Рис. Л2.3. Регистры (Qt Creator)

В среде Code::Blocks содержимое основных регистров можно увидеть, используя окно *CPU registers* (меню *Debug* → *Debugging windows* → *CPU registers*). Регистры сопроцессора отображаются в окне *FPU status* (ме-

ню *Debug*  $\rightarrow$  *Information*  $\rightarrow$  *FPU status*), но, в отличие от окна основных регистров, открытое окно *FPU status* не позволяет продолжить отладку.

В TheIDE содержимое основных регистров показывается под панелью дисассемблера и всегда доступно во время отладки (справа).

### Л2.3. Вопросы

1. Уметь пользоваться окнами просмотра переменных и содержимого памяти в отладчике используемой вами IDE.
2. Чем различается размещение в памяти локальных, глобальных и статических переменных?
3. Чем различается работа с целыми числами разной разрядности?
4. Чем различается работа с целыми и вещественными числами?
5. Как в функции передаются целые параметры (в исследуемом компиляторе и платформе)?
6. Как в функции передаются вещественные параметры (в исследуемом компиляторе и платформе)?
7. Как в функции передаются три и более параметров (в исследуемом компиляторе и платформе)?
8. Чем различается код, созданный компиляторами одного семейства для различных платформ?
9. Чем различается код, созданный различными компиляторами для одной платформы?

## Лабораторная работа 3.

### Модули и функции на ассемблере

**Цель работы:** изучить процесс компиляции программы на C++; научиться включать в проекты на языке C++ ассемблерные модули; научиться описывать функции и вызывать из программы на языке C++.

#### Л3.1. Задание на лабораторную работу

**Задание 1.** Разработайте ассемблерную функцию, вычисляющую целое выражение от целого аргумента (в соответствии с вариантом), а также головную программу на языке C++, использующую разработанную функцию.

$(N - 1) \% 5 + 1$	Вариант
1	$y(x) = 1 + x/2$
2	$y(x) = x \% 4 - x$
3	$y(x) = 3x + 1$
4	$y(x) = x^3$
5	$y(x) = 4 - 4x$

**Задание 2.** Разработайте программу, целиком написанную на ассемблере, вычисляющую значение  $y(x)$  для  $x = 13$  и выводящую полученное значение на стандартный вывод с использованием библиотеки `libc` (в частности, функции `printf`).

**Задание 3.** Опишите функцию на произвольном языке высокого уровня (включая C/C++) и вызовите её из ассемблерной функции.

$(N^{\circ} - 1) \% 3 + 1$	Вариант
1	Вывод двух параметров на экран с пояснениями
2	Ввод результата с клавиатуры
3	Случайный результат в заданном диапазоне

**Задание 4. Бонус (+2 балла).** Опишите на ассемблере одну подпрограмму с параметрами  $a, b, \dots$  и результатами  $x$  и  $y$  и вызовите её из другой ассемблерной программы.

$(N^{\circ} - 1) \% 2 + 1$	Вариант
1	$\begin{cases} x = a + c \cdot b \\ y = a - c \cdot b \end{cases}$
2	$\begin{cases} x = a^2 - b^2 \\ y = 2ab \end{cases}$

### Л3.2. Теоретические сведения

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

4.1 Компиляция

7.2 Типы данных

6.2 Подпрограммы и функции

6.1 Структура программы на ассемблере

5.2 Основные команды

### Подключение к проекту модулей на ассемблере

#### Code::Blocks

Создать ассемблерный модуль в Code::Blocks можно, используя меню *File*  $\rightarrow$  *New*  $\rightarrow$  *Empty file*. Имя файла обязательно должно иметь расширение *.S* (заглавное; расширение *.s* не воспринимается Code::Blocks как допустимое).

После создания в проекте файла с таким расширением он во время сборки проекта обрабатывается препроцессором и компилируется gcc; полученный объектный файл в дальнейшем используется линкером. Дополнительных настроек делать не нужно.

## Qt Creator

Файл проекта Qt Creator для добавления ассемблерного модуля `sqr.S` необходимо отредактировать вручную, добавив строку `SOURCES += sqr.S`, так как мастер добавления файлов не воспринимает расширения `.S` и `.s` как допустимые для исходного кода.

```
1 TEMPLATE = app
2 CONFIG += console
3 CONFIG -= app_bundle
4 CONFIG -= qt
5
6 SOURCES += main.cpp
7 SOURCES += sqr.S
8
9 include(deployment.pri)
10 qtcAddDeployment()
```

Файл `sqr.S` должен находиться в той же папке, что и проект. Других настроек, кроме редактирования файла проекта, делать не нужно.

## Л3.3. Вопросы

1. Какие вы знаете соглашения о вызове?
2. Какая команда передаёт управление подпрограмме?
3. Какая команда возвращает управление вызывающей программе?
4. Что такое адрес возврата?
5. Какие вы знаете регистры общего назначения?
6. Какие вы знаете команды ассемблера x86?
7. Какие вы знаете флаги?

## Лабораторная работа 4.

### Использование ассемблерных вставок в программах на C++

**Цель работы:** научиться вставлять в программы на языке высокого уровня ассемблерные фрагменты.

#### Л4.1. Задание на лабораторную работу

**Задание 1.** Разработайте и выполните программу, в которой реализуйте при помощи ассемблерной вставки

$(N - 1) \% 6 + 1$	Вариант
1	Сложение целых чисел без знака
2	Сложение целых чисел со знаком
3	Вычитание целых чисел без знака
4	Вычитание целых чисел со знаком
5	Увеличение целого числа без знака на 1 (инкремент)
6	Увеличение целого числа со знаком на 1 (инкремент)

После команды сложения/вычитания проанализируйте соответствующий флаг состояния процессора, и, в зависимости от его значения, выведите результат и строку «Результат верный» (если не было переполнения) или «Результат неверный» (при переполнении).

**Задание 2.** Реализуйте задание лабораторной работы Л3 (см. раздел Л3.1, задание 1) как ассемблерную вставку в программу на C++.

**Задание 3. Бонус (+1 балл).** Реализуйте задание 2, не используя в тексте вставки конкретных имён регистров.

**Задание 4. Бонус (+2 балла).**

Придумайте и запишите в отчёте:

- маску (т.е. второй операнд бинарной побитовой операции) для преобразования кодов ASCII [57] цифр '0'...'9' в их двоичные эквиваленты (однобайтовые числа 0...9) с помощью операции AND. При этом ASCII-код нуля (шестнадцатеричное число 0x30, что также иногда записывается как 30<sub>16</sub>, однако наиболее предпочтительное и самодокументирующееся представление ASCII-кода нуля — '0') должен быть преобразован в однобайтовое число 00, ASCII-код единицы ('1', шестнадцатеричное число 0x31 или 31<sub>16</sub>) — в 01, и т. д.;
- необходимую операцию и маску для преобразования однобайтовых чисел 0...9 в их коды ASCII;
- маску, которая преобразовывает строчные английские буквы в прописные и наоборот с помощью операции XOR.

Коды ASCII приведены в приложении Б.

Реализуйте преобразования ASCII-кодов цифр в значения и обратно как ассемблерную вставку в программу на C++.

**Л4.2. Дополнительные бонусные и штрафные баллы**

—4 балла за неуказание в списке перезаписываемых элементов модифицируемых в ассемблерной вставке регистров общего назначения, памяти, флагов.

Исключение — те регистры, которые текущая версия компилятора не позволяет указать в списке перезаписываемых (для некоторых версий это *esi*, *edi*).

**Л4.3. Теоретические сведения**

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

**7.2 Типы данных**



### 4.3 Ассемблерные вставки в код C++

### 5.2 Основные команды

#### **Л4.4. Вопросы**

1. Каким ключевым словом открывается ассемблерная вставка?
2. Чем различаются базовый и расширенный синтаксис вставок?
3. Где описываются входные параметры ассемблерных вставок?
4. Где описываются выходные параметры ассемблерных вставок?
5. Где описываются элементы, изменяемые в ассемблерных вставках?
6. Где описываются метки ЯВУ, на которые может быть передано управление из ассемблерных вставок?
7. Как из ассемблерной вставки обратиться к локальным переменным?
8. Какие вы знаете ограничения на размещение параметров ассемблерных вставок?

## Лабораторная работа 5.

## Программирование сопроцессора

**Цель работы:** познакомиться с работой сопроцессора и его командами.

## Л5.1. Задание на лабораторную работу

**Задание 1.** <sup>1</sup>Разработайте программу на языке C++, выполняющую вычисления над вещественными числами одинарной точности (тип float).

Проверьте, что программа действительно работает с операндами одинарной точности, а не приводит к типу float окончательный результат.

$(N - 1) \% 3 + 1$	Вариант
1	<p>Найти разность <math>a - b</math> для различных значений <math>a, b</math>:</p> <p><math>a = 123456789; b = 123456788,</math></p> <p><math>a = 123456788; b = 123456787</math></p>
2	<p>Найти значение выражения <math>\frac{(a + b)^2 - (a^2 + 2ab)}{b^2}</math></p> <p>для различных значений <math>a, b</math> (пример Лавренова):</p> <p><math>a = 95; b = 1,</math></p> <p><math>a = 95; b = 0,1</math></p>
3	<p>Не используя умножение, найти суммы</p> $S_1 = \sum_{i=0}^{N-1} \frac{1}{N} \text{ и } S_2 = \sum_{i=0}^{2N-1} \frac{1}{N}$ <p>для различных <math>N: 10^2, 10^4, 10^6, 10^7, 10^8, 10^9</math></p>

Сравните полученный результат с теоретическим. Объясните результат.

Измените тип операндов на double. Объясните результат.

---

<sup>1</sup>Использованы материалы: П. Закляков. Представление чисел в памяти ЭВМ. Ошибка «Циклическая дыра». Системный Администратор, № 10 (119) 2012 г.; В. Яшкардин. IEEE 754 — стандарт двоичной арифметики с плавающей точкой

**Задание 2.** Составьте программу `calc` для вычисления выражения с использованием сопроцессора в соответствии со своим вариантом.

**Задание 3. Бонус (+2 балла).** Оформите вычисления из задания 2 как функцию на ассемблере (вещественную от вещественного аргумента  $x$ ).

### Л5.2. Дополнительные бонусные и штрафные баллы

—2 балла за *каждое* использование команды `finit`.

Инициализация процессора, в том числе и модуля FPU, выполняется операционной системой в процессе загрузки.

—2 балла за несбалансированный стек FPU после завершения вычислений.

—2 балла за потерю точности из-за округления промежуточных расчётов в заданиях 2 и 3.

—4 балла за включение в отчёт заведомо недостоверных цифр (более 2 недостоверных десятичных знаков в округлённом результате).

+1 балл за поясняющие текущее содержимое стека FPU комментарии.

### Л5.3. Варианты заданий для программы `calc`<sup>2</sup>

№	МП-30	МП-34	МП-35
1	$\frac{1.2 + x}{x^2}$	$1.5x^2 - 3x - 1$	$\frac{\sin(x - 1)}{2x}$
2	$\frac{\cos(x)}{\sin(x) + 2}$	$\frac{\sin(1.9x)}{x - 1.5}$	$\sin(x) \cdot \cos(0.9x)$
3	$\sin(2x) + \cos(3x)$	$\sqrt{0.99x} + x^2$	$\sin(0.27x) - \cos(x)$
4	$\ln 2 \cdot (\sin(1.2x))$	$\sqrt{3.45x^2 + 0.1}$	$\frac{\cos(x)}{\sin(x) + 2}$
5	$1.5 + 3.88(x^2 - 1)$	$\cos^3(x) + \operatorname{tg}(x)$	$\sin(0.9x) + \cos(x)$

<sup>2</sup>Использованы: лабораторная работа, подготовленная Л. В. Илюшечкиной

6	$\frac{\cos(x)}{\sin(2x) + 1.5}$	$\sin^2(\frac{x}{2} + 1)$	$x(x^2 + 3x - 5)$
7	$\cos(x + 0.9x^2)$	$3.4x^2 + 1.5$	$\sin(1.5x) + \frac{1}{x}$
8	$\cos(x - 0.8) + x$	$(\frac{x}{2} + 1.5)(x + 1)$	$\sin((\cos(x) + 1.3))$
9	$\sqrt{\sin(2x) + x}$	$\frac{x^2 + 4}{\sin(2x)}$	$\sqrt{\frac{x - 1.2}{2x}}$
10	$2x^2 + x - 5$	$\sqrt{ x - \ln 2 }$	$\frac{x \operatorname{tg}(x)}{2}$
11	$(x + 1)(2x - 1)$	$3x^2 + 2x - 1$	$ 3x^3 - 2.5 $
12	$\sin(2x) \cdot \cos(x)$	$(2x + 0.5)(3x - 2)$	$\frac{7 + x}{2x + 1}$
13	$\sin(2.27x) - \cos(x)$	$\sin(1.88x + 1) \cdot \cos(x)$	$\cos(x + \ln 2)$
14	$\frac{\cos(x)}{\sin(2x + 1)}$	$\sqrt{x^3 - 5} + x$	$\cos(x - 0.8) + x^2$
15	$\sin(0.99x) + \cos(x)$	$\frac{\sin^2(x)}{1 + \operatorname{tg}(x)}$	$1.1x + 0.5x^2$
16	$\sin(2.5x) - \cos(x^2)$	$\cos^2(x) + \operatorname{tg}(1.5x)$	$\cos(x) \cdot (1 + 3x^2)$
17	$\sin(1.5x) + \cos(x)$	$2\sqrt{x + 1}$	$\frac{\sin(1.5x)}{\cos(1.5x)}$
18	$\sin((\cos(x) + 1.3))$	$\frac{\sin(x) + 1}{2}$	$1 + 2x - 3x^2$
19	$\sin(t + 1), x = 2.123t$	$(x + 3)^2 - \sin^2(x)$	$ \operatorname{tg}(\frac{x}{2}) + 1 $
20	$\cos(x) + \sin(2x)$	$\cos(2.1x) - \sin(x)$	$\frac{\sin(0.5x)}{x^2}$
21	$0.9x^2 + 0.7$	$\frac{\sin^2(x)}{1 + \operatorname{tg}^3(x)}$	$\cos^2(x) + \operatorname{tg}(2x)$
22	$0.5 \sin(0.5x + 2)$	$\frac{x}{3x^2 - x + 1}$	$x^2 - x + 8.7$
23	$\frac{\sin(x)}{\cos(x) + 1.1}$	$x^2(2 - 3x)$	$\frac{1}{2.5 + x^2}$

24	$\cos((0.7 + x)x)$	$ \cos(2x + 0.9) $	$\sin(x) \cdot (2.5x^2 + 3)$
25	$ \cos(2 + x^2) $	$\cos(x - 0.8) + x^2$	$\sin(x) + 2\cos(x)$
26	$ 1.5x  + x^2$	$1.1x + 0.5x^2$	$\sin(\frac{x}{2.1}) - \cos(2x)$

#### Л5.4. Теоретические сведения

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

3.4 Математический сопроцессор (FPU x87)

5.3 Команды FPU

#### Л5.5. Вопросы

1. Какие регистры используются в сопроцессоре для хранения операндов?
2. Какие команды используются для выполнения арифметических операций над вещественными числами?
3. Какие команды используются для выполнения тригонометрических операций?
4. Какие команды используются для сравнения вещественных чисел?
5. Какие флаги регистра *flags* содержат результат сравнения вещественных чисел?

## Лабораторная работа 6.

## Программирование ветвящихся и циклических алгоритмов

**Цель работы:** ознакомиться с командами условного и безусловного перехода; научиться использовать их для реализации ветвящихся и циклических алгоритмов.

### Л6.1. Задание на лабораторную работу

**Задание 1.** Вычислите для заданных целых  $x, y$

$(N - 1) \% 3 + 1$	Вариант
1	$f(x) = \begin{cases} x/2, & x \% 2 = 0 \\ 3x + 1, & x \% 2 \neq 0 \end{cases}$
2	$f(x, y) = \begin{cases} x & x + y < 4 \\ y + 2, & x + y \geq 4 \end{cases}$
3	$f(x, y) = \begin{cases} y/x & y \% x = 0 \\ y \cdot x, & y \% x \neq 0 \end{cases}$

**Задание 2.** Вычислите для заданного вещественного  $x$

$(N - 1) \% 11 + 1$	Вариант
1	$y(x) = \begin{cases} 2 \cdot (x/3) + 15, & x \geq 0 \\ 0, & x < 0 \end{cases}$
2	$y(x) = \begin{cases} x + x/2 + x/4 + x/8, & x \geq 0 \\ x - 1, & x < 0 \end{cases}$
3	$y(x) = \begin{cases} 0, & x \leq 0.8 \\ x - 0.8, & x > 0.8 \end{cases}$

4	$y(x) = \begin{cases} x^3 - (x-1) \cdot (x+1), & -4 \leq x \leq 4 \\ -1, & x < -4 \vee x > 4 \end{cases}$
5	$y(x) = \begin{cases} \pi, & x > \pi \\ x, & x \leq \pi \end{cases}$
6	$y(x) = \begin{cases} (x^2)/(x+1), & x > -1 \\ x^3, & x \leq -1 \end{cases}$
7	$y(x) = \begin{cases} 1 - x/3, & x < 13 \\ 1 + x, & x \geq 13 \end{cases}$
8	$y(x) = \begin{cases} 2x \cdot (x-1) &  x-1  < 5 \\ -2x, &  x-1  \geq 5 \end{cases}$
9	$y(x) = \begin{cases} x - 1.28, & x < 0 \\ 0, & x \geq 0 \end{cases}$
10	$y(x) = \begin{cases} x/2, &  x  \geq 1 \\ 3x + 1, &  x  < 1 \end{cases}$
11	$y(x) = \begin{cases} 0, & x \leq e \\ 1, & x > e \end{cases}$

**Задание 3.** Задайте с клавиатуры  $N$  и напечатайте первые  $N$  членов последовательности (целой).

$(N-1)\%2 + 1$	Вариант
1	Числа Фибоначчи: $\varphi_0 = \varphi_1 = 1, \varphi_{i+1} = \varphi_i + \varphi_{i-1}$
2	Арифметическая прогрессия $a_0 = 0, a_{i+1} = a_i + x$ , $x$ ввести с клавиатуры

**Задание 4. Бонус (+2 балла)).** Найдите с заданной точностью  $\varepsilon$  сумму ряда (если это возможно). Если ряд расходится, выведите сообщение об этом.

$(\mathbb{N} - 1) \% 5 + 1$	МП-30	МП-34	МП-35
1	$S = \sum_{i=1}^{\infty} (-1)^i \frac{1}{i}$	$S = \sum_{i=2}^{\infty} (-1)^i \frac{i+1}{i^3-1}$	$S = \sum_{i=1}^{\infty} e^{-i}$
2	$S = \sum_{i=0}^{\infty} (-1)^i \frac{1}{2^i}$	$S = \sum_{i=1}^{\infty} (-1)^i \frac{1}{i^2}$	$S = \sum_{i=1}^{\infty} (-1)^i \frac{1}{\sqrt{i}}$
3	$S = \sum_{i=1}^{\infty} \frac{1}{i}$	$S = \sum_{i=2}^{\infty} \frac{i-1}{i^3+1}$	$S = \sum_{i=1}^{\infty} (-1)^i e^{-i}$
4	$S = \sum_{i=0}^{\infty} \frac{1}{2^i}$	$S = \sum_{i=1}^{\infty} \frac{1}{i^2}$	$S = \sum_{i=1}^{\infty} \frac{1}{\sqrt{i}}$
5	$S = \sum_{i=0}^{\infty} (-1)^{2i} \frac{1}{2^i}$	$S = \sum_{i=1}^{\infty} \frac{\sin(i)}{i}$	$S = \sum_{i=1}^{\infty} \frac{\sin(i)}{i^2}$

**Задание 5. Бонус (+2 балла):** Реализуйте умножение двух целых чисел без знака  $(x \cdot y)$  «в столбик» (то есть не как сумму  $\underbrace{x + \dots + x}_y$ ), не используя mul/imul и команды сопроцессора.

## Л6.2. Дополнительные бонусные и штрафные баллы

−2 балла за *каждое* использование команд loop\*, jcx\* или jecx\*.



Хотя команды `loop*` занимают меньше места, чем комбинация команды `dec` с условным переходом `jz/jnz`, но последняя выполняется вдвое быстрее, чем `loop*`. Кроме того, организация цикла при помощи явной модификации счётчика и условных переходов более гибка и, в частности, позволяет организовать вложенный цикл.

### **Л6.3. Теоретические сведения**

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

5.2.6 Флаги

5.2.7 Условные команды

6.3 Программирование нелинейных алгоритмов и взаимодействие со структурами данных

### **Л6.4. Вопросы**

1. Какие вы знаете флаги?
2. Какие вы знаете команды передачи управления?
3. Какие вы знаете команды условной пересылки?
4. Как, согласно ЕСПД, изображается блок «терминатор»?
5. Как, согласно ЕСПД, изображается блок «процесс»?
6. Как, согласно ЕСПД, изображается блок «решение»?

## Лабораторная работа 7.

### Использование массивов

**Цель работы:** ознакомиться с расположением элементов массива в памяти компьютера; научиться обрабатывать массивы, используя ассемблер.

#### Л7.1. Задание на лабораторную работу

**Задание 1.** Создайте, используя язык высокого уровня, массив  $\alpha[]$  из целых чисел.

Изучите расположение элементов массива (используя возможности языка высокого уровня или IDE).

1. Каков размер элемента (в байтах)?
2. Насколько отличаются адреса соседних элементов массива (в байтах)?
3. Одинаково или различно это расстояние для разных элементов массива?
4. Каков общий размер массива (в байтах)?
5. Как получить адрес элемента массива, зная его номер и адрес начала массива?

**Задание 2.** Обработайте массив целых чисел  $\alpha[]$  (выделение памяти и заполнение массива может быть выполнено на языке высокого уровня).

$(N - 1) \% 3 + 1$	Вариант
1	Найти минимальный элемент массива
2	Найти максимальный элемент массива
3	Найти сумму элементов массива

**Задание 3.** Создайте, используя язык высокого уровня, массив  $\beta[]$  из структур, содержащих целый ключ и вещественное значение.

Определите размер массива и расположение элементов (используя возможности языка высокого уровня или IDE).

1. Каков размер структуры (в байтах)?
2. Каковы размеры полей структуры (в байтах)?
3. Насколько отличаются адреса соседних полей структуры (в байтах)?
4. Каков общий размер массива (в байтах)?
5. Как получить адрес поля элемента массива, зная номер элемента, адрес начала массива и имя поля?

**Задание 4.** Реализуйте задание 2 для значений тех элементов массива  $\beta[]$ , ключ которых равен заданному числу  $k$ . Если таких нет, вывести корректное сообщение об этом.

**Задание 5.** Опишите вставку либо функцию, инициализирующую массив заданной длины  $N$  первыми  $N$  элементами последовательности из задания 3 лабораторной работы Л6.

**Задание 6. Бонус (+2 балла).** Создайте массив структур, содержащих целый ключ и вещественное значение или набор значений. Опишите функцию, обрабатывающую этот массив (реализация варианта в виде вставки, а не функции, принесёт не более +1 балла вместо +2). Адрес и длина массива должны передаваться в функцию как параметры.

$(N - 1) \% 2 + 1$	Вариант
1	<p>Структура содержит ключ и одно вещественное число.</p> <p>Функция получает эталонное значение ключа <math>E</math> и нормирует соответствующие ключу значения. После обработки сумма всех значений с ключом, равным <math>E</math>, должна быть равна 1.</p>
2	<p>Структура содержит ключ и комплексное число <math>(re, im)</math>.</p> <p>Изменить ключ каждой структуры на номер квадранта, где располагается число (или на 0, если число лежит на одной из осей).</p> <p>Вернуть количество точек I квадранта.</p>

## Л7.2. Дополнительные бонусные и штрафные баллы

—2 балла за *каждое* использование команд `lор*`, `јсх*` или `јесх*`.

—4 балла за *каждую* утечку памяти (то есть выделенный, но не освобождённый блок динамической памяти).

## Л7.3. Теоретические сведения

Для выполнения данной лабораторной работы необходимо изучить следующие разделы:

5.1.4 Адресация операндов

6.3 Программирование нелинейных алгоритмов и взаимодействие со структурами данных

## Л7.4. Вопросы

1. Как располагаются в памяти элементы массива?
2. Как найти размер массива, зная размер элемента и их количество?
3. Что такое выравнивание полей структуры?
4. Зачем нужно выравнивание данных?

Приложение Б

Коды ASCII

ASCII CONTROL CODE CHART

b7	0	0	0	0	1	1	1	1
b6	0	0	1	1	0	0	1	1
BITS	CONTROL		SYMBOLS		UPPER		LOWER	
b5	0	1	0	1	0	1	0	1
b4			NUMBERS		CASE		CASE	
b3	0	0	0	0	0	0	0	0
b2	0	0	0	0	0	0	0	0
b1	0	0	0	0	0	0	0	0
0 0 0 0	0 NUL	16 DLE	32 SP	48 0	64 @	80 P	96 ‘	112 p
0 0 0 1	1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
0 0 1 0	2 STX	18 DC2	34 ”	50 2	66 B	82 R	98 b	114 r
0 0 1 1	3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
0 1 0 0	4 EOT	20 DC4	36 \$	52 4	68 D	84 T	100 d	116 t
0 1 0 1	5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
0 1 1 0	6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
0 1 1 1	7 BEL	23 ETB	39 ’	55 7	71 G	87 W	103 g	119 w
1 0 0 0	8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
1 0 0 1	9 HT	25 EM	41 )	57 9	73 I	89 Y	105 i	121 y
1 0 1 0	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
1 0 1 1	11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
1 1 0 0	12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124
1 1 0 1	13 CR	29 GS	45 —	61 =	77 M	93 ]	109 m	125 }
1 1 1 0	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
1 1 1 1	15 SI	31 US	47 /	63 ?	79 O	95 -	111 o	127 DEL

LEGEND:

dec	CHAR
hex	oct

Victor Eijkhout  
Dept. of Comp. Sci.  
University of Tennessee  
Knoxville TN 37996, USA

## Список литературы

1. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. USA, Sunnyvale: Advanced Micro Devices Inc., 2013. Т. 1. 390 с.
2. AMD64 Architecture Programmer's Manual Volume 2: System Programming. USA, Sunnyvale: Advanced Micro Devices Inc., 2013. Т. 2. 690 с.
3. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. USA, Sunnyvale: Advanced Micro Devices Inc., 2017. Т. 3. 681 с.
4. AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions. USA, Sunnyvale: Advanced Micro Devices Inc., 2017. Т. 4. 1045 с.
5. AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions. USA, Sunnyvale: Advanced Micro Devices Inc., 2016. Т. 5. 372 с.
6. Bartlett J. Programming from the Ground Up. USA: Bartlett Publishing, 2003. 326 с.
7. Coleman C. L. Using Inline Assembly With gcc. USA, Boston: Free Software Foundation, Inc, 2000. 25 с.
8. Elsner D., Fenlason J. Using as. USA, Boston: Free Software Foundation, Inc, 2009. 318 с.
9. Fog A. Calling conventions for different C++ compilers and operating systems. Denmark, Copenhagen: Technical University of Denmark, 2014. 57 с.
10. Fog A. Optimizing subroutines in assembly language. An optimization guide for x86 platforms. Denmark, Copenhagen: Technical University of Denmark, 2017. 170 с.

11. Glaser A. History of binary and other nondecimal numeration. USA: Tomash publishers, 1981. 231 с.
12. IBM 7090 Data Processing System. USA, New York: International Business Machines Corporation, 1959. 16 с.
13. IEEE Standard for Binary Floating-Point Arithmetic. USA, New York: IEEE, 1985. 23 с.
14. IEEE Std 754<sup>TM</sup>-2008 (Revision of IEEE Std 754-1985). IEEE Standard for Floating-Point Arithmetic. USA, New York: IEEE, 2008. 70 с.
15. Intel® 64 and IA-32 Architectures Optimization Reference Manual. USA, Santa Clara: Intel Corporation, 2016. 672 с.
16. Intel® 64 and IA-32 Architectures Software Developer's Manual. Basic Architecture. USA, Santa Clara: Intel Corporation, 2017. T. 1. 482 с.
17. Intel® 64 and IA-32 Architectures Software Developer's Manual. Instruction Set Reference, A-Z. USA, Santa Clara: Intel Corporation, 2017. T. 2. 2234 с.
18. Intel® 64 and IA-32 Architectures Software Developer's Manual. System Programming Guide. USA, Santa Clara: Intel Corporation, 2017. T. 3. 1660 с.
19. Intel® 64 and IA-32 Architectures Software Developer's Manual. Model-Specific Registers. USA, Santa Clara: Intel Corporation, 2017. T. 4. 420 с.
20. ISO/IEC 9899:201x. Programming languages — C. Committee Draft — April 12, 2011. ISO/IEC, 2011. 701 с.
21. Juan Caramuel Lobkowitz: The Last Scholastic Polymath / Под ред. P. Dvořák, J. Schmutz. Czech Republic, Prague: Filosofia, 2008. 408 с.
22. Neumann J. v. First Draft of a Report on the EDVAC: Tech. rep.: 1945.
23. Rojas R. Konrad Zuse's Legacy: The Architecture of the Z1 and Z3 // Annals of the History of Computing, IEEE. 1997. — apr-jun. Vol. 19, no. 2. P. 5–16.
24. Stallman R. M., the GCC Developer Community. Using the GNU Compiler Collection For GCC version 5.4.0. USA, Boston: GNU Press, 2015. 810 с.

25. Working Draft, Standard for Programming Language C++. ISO/IEC, 2014. 1368 с.
26. Абель П. Язык Ассемблера для IBM PC и программирования. М.: Высшая школа, 1992. 447 с.
27. Александров Е. К., Грушвицкий Р. И., Купрянов М. С., Мартынов О. Е. Микропроцессорные системы. СПб: Политехника, 2002. 935 с.
28. Бурдаев О. В., Иванов М. А., Тетерин И. И. Ассемблер в задачах защиты информации. М.: Кудиц-образ, 2004. 538 с.
29. Воройский Ф. С. Информатика. Новый систематизированный толковый словарь. М: Физматлит, 2003. 760 с.
30. ГОСТ 19.701-90 Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения. М: Стандартиформ, 2010. 158 с.
31. ГОСТ 34.003-90 Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Термины и определения. М: Стандартиформ, 2009. 16 с.
32. Ершов А. П., Шура-Бура М. Р. Становление программирования в СССР. Новосибирск: Сибирское отделение РАН Институт систем информатики им. А. П. Ершова, 2016. 79 с.
33. Зубков С. В. Assembler для DOS, Windows и UNIX. М: ДМК-Пресс, 2017. 638 с.
34. Илюшечкина Л. В. Курс лабораторных работ по ArchVM.
35. Касперски К. Техника оптимизации программ. Эффективное использование памяти. СПб: БХВ-Петербург, 2003. 464 с.
36. Касперски К., Рокко Е. Искусство дизассемблирования. СПб: БХВ-Петербург, 2009. 896 с.
37. Кушнерев Н. Т., Неменман М. Е., Цагельский В. И. Программирование для ЭВМ «Минск-32». М: Статистика, 1973. 248 с.



38. Малашевич Б. М. Неизвестные модулярные супер-ЭВМ // PCWeek Russian Edition. 2005. № 9. С. 44–45.
39. Малашевич Б. М. Модулярная арифметика и модулярные компьютеры // История информационных технологий в СССР. Знаменитые проекты: компьютеры, связь, микроэлектроника. М.: Книма, 2016. С. 228–257.
40. Орлов С. А., Цилькер Б. Я. Организация ЭВМ и систем. Фундаментальный курс по архитектуре и структуре современных компьютерных средств. 3-е издание. СПб: Питер, 2014. 688 с.
41. Пухальский Г. И., Новосельцева Т. Я. Цифровые устройства: Учебное пособие для втузов. СПб: Политехника, 1996. 885 с.
42. Ревич Ю. В., Малиновский Б. Н. Информационные технологии в СССР. Создатели советской вычислительной техники. СПб: BHV, 2014. 336 с.
43. Садыхов Р. Х., Поденок Л. П., Отвагин А. В. и др. Средства параллельного программирования в ОС Linux. Мн.: ЕГУ, 2004. 475 с.
44. Смоленцев М. Программирование на языке Ассемблера для 32/64-разрядных микропроцессоров семейства 80x86. Иркутск: ИрГУПС, 2009. 192 с.
45. Таненбаум Э., Остин Т. Архитектура компьютера. СПб: Питер, 2014. 816 с.
46. Федотов А. М. Современные проблемы информатики и вычислительной техники. Новосибирск: Новосибирский государственный университет, 2010. 43 с.
47. Фомин С. В. Системы счисления. М: Наука, 1987. 48 с.
48. Фролов А., Фролов Г. Аппаратное обеспечение персонального компьютера. М: Диалог-МИФИ, 1997. 304 с.
49. Шелихов А. А., Селиванов Ю. П. Вычислительные машины. Справочник. М: Энергия, 1973. 216 с.
50. Юричев Д. Reverse Engineering для начинающих. China: PTPress publisher, 2017. 1063 с.
51. Юров В. И. Архитектура компьютера. СПб: Питер, 2010. 637 с.

52. A Brief Tutorial on GCC inline asm (x86 biased) [Электронный ресурс]. URL: <http://www.osdever.net/tutorials/view/a-brief-tutorial-on-gcc-inline-asm> (дата обращения: 07.07.2017).
53. Asmworld. Программирование на ассемблере для начинающих и не только [Электронный ресурс]. URL: <http://asmworld.ru/> (дата обращения: 07.07.2017).
54. AT&T Syntax bugs [Электронный ресурс]. URL: [http://sourceware.org/binutils/docs/as/i386\\_002dBugs.html](http://sourceware.org/binutils/docs/as/i386_002dBugs.html) (дата обращения: 07.07.2017).
55. CSCI 241 Intermediate Programming in C++ Spring 2015 (The C++ compilation process) [Электронный ресурс]. URL: <http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html> (дата обращения: 07.07.2017).
56. David John Wheeler [Электронный ресурс]. URL: [http://www.thocp.net/biographies/wheeler\\_david.htm](http://www.thocp.net/biographies/wheeler_david.htm) (дата обращения: 07.07.2017).
57. Eijkhout V. ASCII control code chart [Электронный ресурс]. URL: <http://mirror.macomnet.net/pub/CTAN/help/Catalogue/entries/ascii-chart.html> (дата обращения: 07.07.2017). 2009.
58. GCC and File Extensions [Электронный ресурс]. URL: <http://labor-liber.org/en/gnu-linux/development/extensions> (дата обращения: 07.07.2017).
59. Guyver M. The Trouble With FSUB [Электронный ресурс]. URL: <http://www.mindfruit.co.uk/2012/03/trouble-with-fsub.html> (дата обращения: 07.07.2017). 2012.
60. History of operating systems [Электронный ресурс]. URL: <http://www.osdata.com/kind/history.htm> (дата обращения: 07.07.2017).
61. How to use RIP Relative Addressing in a 64-bit assembly program? [Электронный ресурс].
62. Lonesome TSH/Digital Daemons. Inline Assembler в GCC [Электронный ресурс]. URL: [http://sysbin.com/files/lowlevel/gcc\\_inline\\_assembly](http://sysbin.com/files/lowlevel/gcc_inline_assembly).

htm (дата обращения: 07.07.2017).

63. Padua S. The Marvellous Analytical Engine — How It Works [Электронный ресурс]. URL: <http://sydneyepadua.com/2dgoggles/the-marvellous-analytical-engine-how-it-works/> (дата обращения: 07.07.2017). 2015.
64. Ramankutty H. От C к Ассемблеру [Электронный ресурс]. URL: [http://opennet.ru/base/dev/from\\_c\\_to\\_asm.txt.html](http://opennet.ru/base/dev/from_c_to_asm.txt.html) (дата обращения: 07.07.2017). перевод Андрей Киселёв.
65. Rojas R. Наследие Конрада Цузе: Архитектура Z1 и Z3 [Электронный ресурс]. URL: <https://geektimes.ru/post/210412/> (дата обращения: 07.07.2017).
66. Sandeep.S. GCC-Inline-Assembly-HOWTO [Электронный ресурс]. URL: <http://ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html> (дата обращения: 07.07.2017).
67. Архитектура системы [Электронный ресурс]. URL: <http://infosys2006.narod.ru/struct/architec.htm> (дата обращения: 07.07.2017).
68. Гончаров В. Управление памятью в сетевой подсистеме и ядре FreeBSD в целом [Электронный ресурс]. URL: <https://nuclight.livejournal.com/129544.html?nojs=1> (дата обращения: 07.07.2017).
69. Ильин Е. История UNIX-систем [Электронный ресурс]. URL: <http://jenuay.net/blog/2012/02/04/history-unix-systems/> (дата обращения: 07.07.2017). 2012.
70. Караваев Д. Ю. Об исключенных командах или за что «списали» инструкцию INTO? [Электронный ресурс]. URL: <http://rdsn.org/article/pl1/pl1ex12/pl1ex12.xml> (дата обращения: 07.07.2017).
71. Клуб 155. Архитектура и система команд микропроцессоров x86 [Электронный ресурс]. URL: <http://www.club155.ru/programming> (дата обращения: 07.07.2017).

72. Михайличенко А. Приведение типов [Электронный ресурс]. URL: <https://habrahabr.ru/post/106294/> (дата обращения: 07.07.2017).
73. Поляков А. В. Нормализованная запись числа [Электронный ресурс]. URL: <http://av-assembler.ru/asm/afd/asm-real-normalization.htm> (дата обращения: 07.07.2017).
74. Потёмкин А. Основы компьютера для маньяков [Электронный ресурс]. URL: <http://phg.su/basis2/base.htm> (дата обращения: 07.07.2017).
75. Ревич Ю. В. ЕС ЭВМ. Крупнейший промах или всеобщее счастье? [Электронный ресурс]. URL: [http://nnm.me/blogs/2bytes/es\\_evm\\_krupneyshiy\\_promah\\_ili\\_vseobshee\\_schaste/](http://nnm.me/blogs/2bytes/es_evm_krupneyshiy_promah_ili_vseobshee_schaste/) (дата обращения: 07.07.2017).
76. Руководство по Qt Creator [Электронный ресурс]. URL: <http://doc.crossplatform.ru/qtcreator/2.0.1/> (дата обращения: 07.07.2017).
77. Сагалаева Е. Приведение типов в C++ [Электронный ресурс]. URL: <http://alenaspp.blogspot.ru/2005/08/c.html> (дата обращения: 07.07.2017).
78. Семенко А. Smart ASM [Электронный ресурс]. URL: <http://sasm.narod.ru> (дата обращения: 07.07.2017).
79. Холодилов С. А. Код Грея [Электронный ресурс]. URL: <http://rsdn.org/article/alg/gray.xml> (дата обращения: 07.07.2017).
80. Яшкардин В. IEEE 754 — стандарт двоичной арифметики с плавающей точкой [Электронный ресурс]. URL: <http://www.softelectro.ru/ieee754.html> (дата обращения: 07.07.2017). 2012.

## Список таблиц

2.1	Перевод $0,135_{10}$ в двоичную систему счисления . . . . .	83
2.2	Перевод $0,0010001010_2$ в десятичную систему счисления . . . . .	84
2.3	Соответствие двоичных триад восьмеричным цифрам . . . . .	88
2.4	Соответствие двоичных тетрад шестнадцатеричным цифрам . . . . .	89
2.5	Различные способы представления знаковых чисел (кодирование)	98
2.6	Различные способы представления знаковых чисел (декодирование)	99
2.7	Логические операции над разрядами . . . . .	115
2.8	Знаковое и беззнаковое расширение . . . . .	118
2.9	Стандартные двоичные форматы с плавающей запятой . . . . .	130
3.1	Регистр флагов <i>flags</i> . . . . .	166
3.2	Загрузка состояния FPU в регистр флагов . . . . .	169
3.3	Адресация операнда при помощи полей <i>Mod</i> и <i>R/M</i> . . . . .	174
3.4	Номера (коды) регистров . . . . .	177
3.5	Номера (коды) регистров общего назначения при использовании <i>REX</i> . . . . .	179
4.1	Модификаторы параметров ассемблерных вставок GCC . . . . .	220
5.1	Суффиксы размера операндов . . . . .	234
5.2	Двойные суффиксы размера для копирования целых чисел с расширением . . . . .	235
5.3	Основные общие команды . . . . .	239
5.4	Команды передачи управления, вызова и возврата . . . . .	244
5.5	Команды целочисленной арифметики . . . . .	250
5.6	Команды умножения и деления неявного аргумента <i>A</i> . . . . .	251
5.7	Команды расширения (увеличения разрядности) . . . . .	253
5.8	Мнемоники команд знакового расширения <i>A</i> . . . . .	254

5.9	Основные битовые операции . . . . .	256
5.10	Команды обработки флагов . . . . .	259
5.11	Условия и их связь с флагами состояния <i>flags</i> . . . . .	261
5.12	Команды передачи управления . . . . .	263
5.13	Команды пересылки данных . . . . .	264
5.14	Команды загрузки данных в стек FPU . . . . .	274
5.15	Команды выгрузки данных из стека FPU . . . . .	275
5.16	Команды пересылки данных FPU . . . . .	278
5.17	Условия <i>fstovCC</i> и их связь с флагами состояния <i>flags</i> . . . . .	278
5.18	Команды загрузки и выгрузки управляющих регистров FPU . . .	279
5.19	Основные арифметические операции FPU . . . . .	280
5.20	Шесть форм основных арифметических команд FPU . . . . .	282
5.21	Дополнительные арифметические и трансцендентные команды FPU	286
5.22	Команды сравнения FPU . . . . .	289
5.23	Значение флагов при сравнении . . . . .	290
5.24	Значение флагов при определении вида <i>st(0)</i> . . . . .	292
6.1	Тридцатидвухбитные соглашения о вызовах . . . . .	308
6.2	Шестидесятичетырёхбитные соглашения о вызовах . . . . .	309
6.3	Размер выравнивания для данных различных типов . . . . .	343
7.1	Минимальная разрядность стандартных целых типов . . . . .	353
7.2	Основные форматы вывода <i>printf()</i> . . . . .	373
7.3	Основные флаги вывода <i>printf()</i> . . . . .	374

## Список иллюстраций

1.1	Схема системной платы . . . . .	19
1.2	Структура системной шины . . . . .	20
1.3	Двухбайтовое число: а) биты старшего и младшего байтов, б) прямой порядок байтов в памяти, в) обратный порядок байтов в памяти . . . . .	24
1.4	Четырёхбайтовое число: а) байты и биты числа, б) прямой порядок байтов в памяти, в) обратный порядок байтов в памяти . . .	25
1.5	Цикл выполнения команды . . . . .	28
1.6	Расположение программ и данных в фон-неймановской (а) и гарвардской (б) архитектурах . . . . .	32
1.7	Иерархия запоминающих устройств. Сверху вниз увеличивается объём и уменьшается скорость обмена . . . . .	34
2.1	Эффективность систем счисления . . . . .	74
2.2	Геометрическая интерпретация позиционного представления дробной части в различных системах счисления . . . . .	79
2.3	Беззнаковый (а) и знаковый (б) сдвиги вправо . . . . .	119
2.4	Знаковый (беззнаковый) сдвиг влево . . . . .	121
2.5	Простой циклический сдвиг: а) влево, б) вправо . . . . .	121
2.6	Циклический сдвиг через флаг переноса: а) влево, б) вправо . . .	122
2.7	Структура числа с плавающей запятой согласно стандарту IEEE 754: а) нормализованное число, б) денормализованное, в) ноль, г) бесконечность, д) неопределённость или нечисло . . .	128

2.8	Структура внутреннего представления чисел в FPU x87: а) нормализованное число, б) денормализованное, в) ноль, г) бесконечность, д) вещественная неопределённость, е) тихое нечисло, ж) сигнальное нечисло . . . . .	131
2.9	Недопустимые значения в FPU x87: а) с нулевым порядком, б) с порядком $p_{min} \leq p \leq p_{max}$ , в) с порядком, состоящим из единиц . . . . .	132
3.1	Режимы работы современных процессоров . . . . .	141
3.2	Распределение памяти процесса в тридцатидвухбитной операционной системе GNU/Linux . . . . .	144
3.3	Стек . . . . .	148
3.4	Изменение указателя стека при вызове и возврате из функций . .	149
3.5	Удаление данных из стека — изменение указателя . . . . .	150
3.6	Регистры общего назначения в тридцатидвухбитном режиме . .	153
3.7	Регистры общего назначения в шестидесятичетырехбитном режиме	155
3.8	Регистры FPU . . . . .	158
3.9	Слово тегов FPU . . . . .	159
3.10	Слово состояния и управляющее слово FPU . . . . .	159
3.11	Стек FPU . . . . .	160
3.12	Структура команды в архитектуре x86 . . . . .	172
3.13	Префикс расширения регистров <i>REX</i> в структуре команды x86-64	178
4.1	Этапы компиляции программы на C++ . . . . .	190
4.2	Совместная компиляция нескольких модулей . . . . .	195
4.3	Совместная компиляция модулей на разных языках . . . . .	196
6.1	Изменение указателя стека командами вызова и возврата . . . .	304



6.2	Размещение локальных переменных в стеке оптимизирующим компилятором . . . . .	305
6.3	Размещение локальных переменных в стеке (устаревший вариант)	306
6.4	Параметры и адрес возврата в соглашении cdecl . . . . .	311
6.5	Компиляция C-функции без искажения имён (а) и с искажением, принятым в большинстве версий Microsoft Windows (б) . . . . .	320
6.6	Компиляция C-функции без искажения имён (а) и с компенсацией искажения (б) . . . . .	322
6.7	Алгоритм и реализация ветвления с операторами в одной ветви .	328
6.8	Алгоритм и реализация ветвления . . . . .	331
6.9	Алгоритм и реализация цикла . . . . .	333
Л1.1	Отладка с помощью Qt Creator . . . . .	392
Л1.2	Точка останова в коде . . . . .	393
Л1.3	Локальные и наблюдаемые переменные Qt Creator . . . . .	394
Л1.4	Отладочные окна Memory и Watches в Code::Blocks . . . . .	394
Л1.5	Обозреватель и редактор памяти Qt Creator . . . . .	395
Л1.6	Создание 64-битной конфигурации для проекта в Microsoft Visual Studio . . . . .	396
Л2.1	Дизассемблированный код программы (Qt Creator) . . . . .	400
Л2.2	Дизассемблированный код программы (Code::Blocks) . . . . .	401
Л2.3	Регистры (Qt Creator) . . . . .	402

# Предметный указатель

- 0x66, префикс изменения размера операнда, 181, 190
- 0x67, префикс изменения размера адреса, 181, 190
- 1C, 48
- APL, 48, 57
- Apple II, 62
- ARM, 62, 63
- ASCII, 67, 357, 373, 374, 382, 416, 442
- Bash, 47, 370, 398
- Bell Model V, 45, 55
- BINAC, 46, 56
- BIOS, 21
- C#, 47, 61
- CDC 6600, 57
- CISC, 31, 41, 179
- Colossus Mark I, 56
- Colossus Mark II, 56
- Commodore PET, 62
- Cray-1, 61
- Cray-X1E, 63
- CSIRAC, 56
- EDSAC, 46, 56
- EDSAC-2, 57
- EDVAC, 56
- Elliot-803, 57
- ENIAC, 56
- EVEX, префикс, 188
- Ferranti, 56
- FPU, 145, 165, 178, 280
- Harvard Mark I, 55
- Harvard Mark II, 55
- Harvard Mark III, 55, 56
- IBM 360, 58–60
- IBM 370, 59, 60
- IBM 7030, 57, 58
- IBM 704, 45
- IBM 709, 45
- IBM PC, 62, 145
- IDE Code::Blocks
  - дамп памяти, 416
  - окно дизассемблера, 421
  - синтаксис дизассемблера, 423
- Intel 4004, 61, 143
- Intel 4040, 143
- Intel 8008, 143
- Intel 80186, 145
- Intel 80286, 145
- Intel 80386, 145, 146
- Intel 8080, 143

- Intel 8086, 143, 145
- Intel 8087, 145
- Intel 8088, 62, 145
- Intel Itanium, 147
- Java, 47, 61
- Lorenz SZ, 54
- main(), 311, 369
- PDP-1, 57, 58
- PDP-11, 59
- PDP-4, 57, 58
- PDP-5, 57
- PDP-7, 63
- PDP-8, 57, 59
- Perl, 48
- PHP, 47
- POSIX, 64
- Python, 47, 48
- REX, префикс, 147, 162, 187, 190, 191, 233
- RISC, 31, 32, 41
- Ruby, 48
- Simens-2002, 57
- SUS, 64
- Tandy TRS-80, 62
- TRADIC, 58
- TX-0, 57
- TX-2, 57
- UNIVAC, 56
- Unix, 63
- VEX, префикс, 188
- void, специальный тип C++, 377
- Xerox Alto, 62
- Z1, 53, 71
- Z2, 54, 71
- Z22, 56
- Z3, 54
- Адресация
  - виды, 179, 242
  - косвенная, 180, 183, 243, 254, 260, 353
  - непосредственная, 179, 182, 242
  - неявная, 179
  - прямая, 180, 183, 242, 253
  - прямая относительная, 180, 185, 242, 254, 257, 259
  - регистровая, 180, 183, 186, 243, 260
- Айкен, Говард, 54
- Акация, 62
- АЛУ, 21, 41, 52, 59
- Алгол, 48, 57

- Алмаз, 59
- Альтаир, 62
- Арифметика
- беззнаковая, 93, 177, 264–266
    - вычитание, 96
    - деление, 99
    - сложение, 94
    - умножение, 97
  - двоично-десятичная, 115, 252
  - знаковая, 108, 177, 264–266
    - вычитание, 109
    - сложение, 108
    - умножение, 111
  - модулярная, 117
  - с насыщением, 118
- Арифмометр, 43, 49, 52
- Арка, 59
- Арфа, 59
- Архитектура
- гарвардская, 33
  - фон-неймановская  
(принстонская), 32
- Ассемблирование, 202
- Атака, 59
- Байт, 22, 42, 58
- порядок байтов, 23
- Бардин, Джон, 57
- Бейсик, 48
- Бит, 22, 39, 53
- Битовые операции, 119
- Битовые сдвиги, 125, 263, 266, 269
- Бонч-Бруевич, Михаил Александрович, 55
- Браттейн, Уолтер, 57
- БЭСМ-1, 56
- БЭСМ-2, 56
- БЭСМ-4, 57
- БЭСМ-6, 57, 59, 61
- Бэббидж, Чарльз, 33, 43, 51–55, 71
- Бэкус, Джон, 48
- Вещественные типы C++, 375
- Виберг, Мартин, 52
- Гранит, 58
- Декорирование, 202, 337
- Диана, 58
- Днепр-2, 59
- да Винчи, Леонардо, 52
- де Кольмар, Тома, 52
- ЕС ЭВМ, 60
- Жаккар, Жозеф, 43, 51
- Защищённый режим, 145, 147, 149
- К-340А, 72

Калькулятор, 40, 43, 49, 51, 52

Карат, 59

Карат-КМ-Е, 62

Качественные данные, 65

Керниган, Брайан, 63

Кеш-память, 22, 34

Килби, Джек, 59

Клён, 58

Кобол, 48, 57

Количественные данные, 65

Команды

adc, 263

add, 223–225, 255, 262, 263, 271,  
313, 335, 340

and, 269, 272

bound, 252

bt, 221, 269

btc, 269

btr, 269

bts, 269

call, 190, 257–259, 312, 320, 335,  
340, 341

cbw/cbtw, 249, 267

cdqe/cltq, 249, 267

clc, 273

cld, 273

cli, 273

cmc, 273

cmovCC, 274, 278, 345, 348

cmp, 263, 272, 345–349, 352, 354

cqo/cqto, 249, 267

cwd/cwtd, 249, 267

cwde/cwtdl, 249, 267

dec, 263, 271, 276, 278

div, 263, 265

f2xmi, 170, 301, 302

fabs, 302

fadd[p], 288, 294, 295

fbld, 289

fbst[p], 170, 290

fcmovCC, 274, 278, 293

fcom, 272

fcom[p[p]], 305

fcomi[p], 305, 306

fcos, 170, 301, 302

fdiv[p], 250, 294, 295

fdivr[p], 250, 294, 295

fiadd[p], 294, 295

ficom[p], 305

fidiv[p], 294, 295

fidivr[p], 294, 295

fild, 289

fimul[p], 294, 295

finit, 280, 286

fist[p], 170, 290

fisub[p], 294, 295

fisubr[p], 294, 295

fld, 288, 289

fld1, 289

fldcw, 294

fldl2e, 289

fldl2t, 289

fldlg2, 289

fldln2, 289

fldpi, 289

fldz, 289

fmul[p], 294, 295

fnstcw, 294

fnstsw, 294, 307

fpatan, 170, 301–303

fprem1, 302

fptan, 170, 301, 302

FPU

арифметические, 294

выгрузки, 290

дополнительные, 301

загрузки, 288

пересылки, 290

сравнения, 304

трансцендентные, 170, 301

fscale, 302

fsin, 170, 301, 302

fsincos, 170, 301, 302

fsqrt, 302

fst[p], 170, 288, 290

fsub[p], 250, 294, 295

fsubr[p], 250, 294, 295

ftst, 305

fucom[p[p]], 305, 306

fucomi[p], 305, 306

fwait/wait, 286

fxam, 304

fxch, 293

fyl2x, 170, 301, 302

fyl2xp1, 170, 301, 302

idiv, 263, 265

imul, 225, 263, 265, 272, 333, 340

inc, 263, 271, 276, 352, 354

int, 257, 258, 260, 316, 342, 343

into, 252

iret, 257, 258, 260

jCC, 256, 257, 274, 277, 278, 346,  
349, 350, 352, 354

jcxz, 278

jecxz, 278

jmp, 256, 257, 277, 349, 352, 354

lahf, 273

lea, 231, 251, 252, 254, 264

loop, 278, 344

loopCC, 278

mov, 46, 159, 182, 191, 198,  
223–225, 241, 247, 251, 252,  
254, 278, 330, 333, 340  
movabs, 191, 250, 252, 254  
movs, 247, 249, 266, 278  
movz, 249, 266, 278  
mul, 263, 265, 272  
neg, 263  
nop, 252, 287  
not, 269, 272  
or, 269, 272  
pop, 155, 190, 252, 255, 312, 320  
popf/popfd/popfq, 273  
push, 155, 190, 252, 255, 312, 320,  
335, 340  
pushf/pushfd/pushfq, 273  
rcl, 269  
rcr, 269  
ret, 190, 257–259, 320, 333, 341  
rol, 269  
ror, 269  
sahf, 273, 307  
sal, 263, 269  
sar, 263, 269  
sbb, 263  
setCC, 274, 279, 347  
shl, 263, 269  
shr, 263, 269, 352

stc, 273  
std, 273  
sti, 273  
sub, 255, 263, 271  
syscall, 252, 257, 259, 261, 342, 343  
sysenter, 252, 257, 259, 260, 342,  
343  
sysexit, 257, 259, 260  
sysret, 257, 259, 261  
test, 269, 272  
xor, 269, 272, 312, 352, 354  
арифметические, 262, 271, 276  
обнуления регистра, 261, 312  
передачи управления, 256, 277  
пересылки, 252  
расширения, 249, 266  
сравнения, 272  
удвоения разрядности А, 249, 266  
условной пересылки, 278, 292  
установки байта, 279  
Компилятор, 32, 44–47, 195, 196, 199,  
201–203, 205  
Компиляция  
модули, 205  
этапы, 200  
Компоновка, 202, 205, 207, 337  
Компьютер, 43, 49, 51  
Конвейер, 30, 32

- Куча, 154, 237
- Лада-2, 62
- Лейбниц, Готфрид, 52, 71
- Лисп, 48
- Литералы, 237, 382
  - вещественные, 384
  - строковые, 385
  - целочисленные, 382
- Логический вентиль, 40, 59
- М-1, 56
- М-220, 57
- М-222, 57
- Мантисса, 132, 134–140, 166, 167, 281, 283
- Массивы
  - динамические, 360
  - матрицы, 360
  - многомерные, 359
  - одномерные, 353
- МИР, 57
- МИР-2, 59
- Минск-1, 56
- Минск-2, 57
- Минск-32, 57
- Мокли, Джон, 32, 46
- МЭСМ, 56
- Н-1, 57
- Наири-1, 57
- Наири-2, 57
- Наири-3, 59
- Наири-4, 59
- Наур, Питер, 48
- Неопределённость
  - вещественная, 134–136, 138, 171, 283, 284, 304, 306, 308
- Нецифровые символы, 78
  - знак, 79
  - простые дроби, 79
  - разделитель дробной части, 79
- Нойс, Роберт, 59
- Однер, Вильгодт, 52
- Октет, 22
- Осокин, Юрий Валентинович, 59
- Память
  - виды, 34
  - внешняя, 18
  - оперативная, 18
  - плоская модель, 35, 145, 150
  - просмотр содержимого, 395, 415
  - распределение адресов, 150
  - сегментная модель, 144
- Параметр, 59
- Параметры командной строки, 45, 369, 370



Паскаль, 47, 48, 194, 210

Паскаль, Блез, 52

Перезаписываемые элементы, 220,  
222, 224–226, 288, 354, 429

память, 226

флаги, 225

ПЗУ, 20

Планкалкюль, 47

Подпрограммы, 258, 317

вызов, 320

пролог, 321

соглашения о вызовах, 156, 324

стек вызовов, 155

эпилог, 321

Поразрядные операции, 120, 269

Порядок, 132, 134, 136, 141, 166, 167,  
281, 283

ПП-1, 47

Представление вещественных чисел

двойной точности, 136, 376

одинарной точности, 136, 376

позиционное, 80

простые дроби, 79

расширенной точности, 137, 281,  
376

с плавающей запятой, 106, 132,  
281, 284, 376

с фиксированной запятой, 129

Представление знаковых целых чи-  
сел, 100, 285, 373

двоично-десятичное, 284, 285

дополнительный код, 106

код с избытком, 105

код со знаком, 104

Представление натуральных чисел,  
88, 373

восьмеричное, 90

двоично-десятичное, 113

двоичное, 88

модулярное, 72, 116

перевод, 75

позиционное, 73

троичное, 72

шестнадцатеричное, 92

экономичность, 71, 76

Препроцессинг, 201

Препроцессор, 201

включение файла, 209

макросы, 211, 389

условия, 210

Проминь, 57

Процессор, 18, 21, 59

Радон, 58

Раздан-2, 57

Расширение целых чисел, 123, 266

- Реальный режим, 145, 147
- Регистры, 21, 158
- общего назначения, 159, 160, 162
  - расширений, 164
  - специальные, 164
  - флагов, 173
- Реле, 54, 55, 71
- Ритчи, Денис, 63
- СAB-500, 58
- Сегмент, 144, 150, 237
- Сетунь, 39, 58, 72
- Синтаксис
- AT&T, 197, 234, 235, 296
  - Intel, 197, 235
- Системная (материнская) плата, 17, 18
- Системные вызовы, 44, 256, 315–317, 341–343
- Слово
- x86, 23, 143
  - двойное, 23
  - машинное, 22, 42, 53, 320
  - четверное, 23
- Стандарты C/C++, 368
- Стек, 144, 151, 155, 161, 187, 190, 219, 237, 245, 255, 259, 312–314, 320–325, 327, 329, 330, 336
- Стрела, 56
- Суперскалярность, 31, 32
- Табулятор, 49
- Тактовый генератор, 20
- Томпсон, Кен, 63
- Транзистор, 57, 59, 71
- Триггер, 55, 59
- Триод, 55, 57
- Трит, 39
- Уилер, Дэвид Джон, 46
- Урал-1, 56
- Урал-11М, 59
- Урал-14, 57
- Урал-25, 59
- УУ, 21, 41, 52, 59
- Фаулер, Томас, 52
- Феликс, 52
- Флаги, 28, 29
- AF (коррекции), 175, 176, 271, 272, 275
  - CF (переноса), 95, 110, 125, 173, 175–178, 269, 271, 272, 275, 279, 304–307
  - DF (направления), 174, 175
  - FPU, 171
  - C0, 178, 294, 305–308
  - C1, 178, 294, 307, 308

- C2, 178, 294, 305–308
- C3, 178, 294, 305–308
- OF (переполнения), 110, 173–175, 177, 271, 272, 275
- PF (чётности), 175, 176, 178, 271, 272, 275, 279, 304–307
- SF (знака), 175, 176, 271, 272, 275
- ZF (нуля), 175, 176, 178, 271, 272, 275, 279, 304–307
- проверка, 274, 277–279, 292
- состояния, 175
- установка, 177, 271
- Фортран, 45, 48, 57, 195
- фон Нейман, Джон, 32, 77
- Холлерит, Герман, 54
- Хоппер, Грейс, 47, 48
- Целые типы C++, 371
- Цикл выполнения команды, 27
- Цузе, Конрад, 32, 43, 47, 51, 52, 54, 55, 71
- Шаблоны C++, 387, 399
- Шестидесятичетырёхбитный режим, 148, 149, 153, 162, 164, 184–192, 243, 252, 254, 259, 261, 326
- Шиккард, Вильгельм, 52
- Штибитц, Джордж, 54
- Шутц, Георг, 52
- Экерт, Джон Преспер, 32, 46
- Электроника ИЦ-8010, 62
- Эльбрус, 61, 146
- ЭНИАК, 71
- Энигма, 54
- Эпос, 72